From Application to Technology OpenCL Application Processors

Chung-Ho Chen

Computer Architecture and System Laboratory (CASLab)
Department of Electrical Engineering and
Institute of Computer and Communication Engineering
National Cheng-Kung University
Tainan, Taiwan

Welcome to this talk

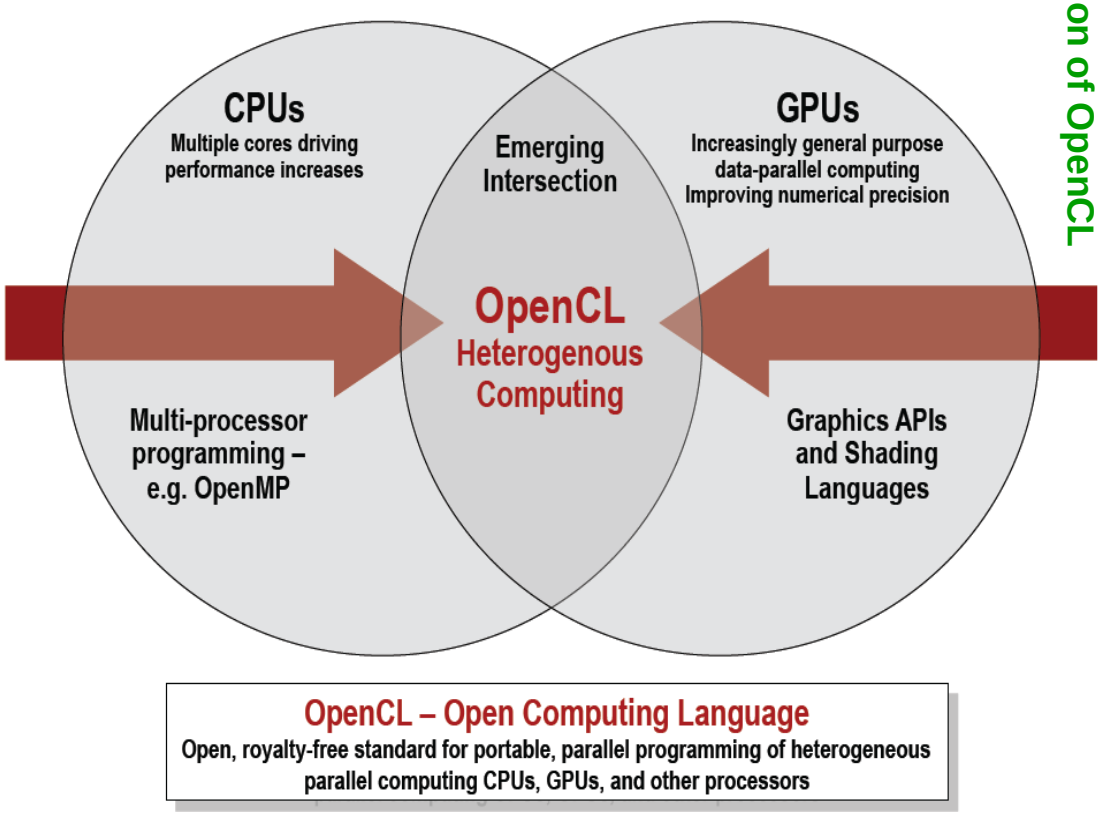
- From Application
  - OpenCL Platform Model
- To Technology
  - Micro-architecture implications
  - An OpenCL runtime on a many-core system
- Summary

Welcome to this talk

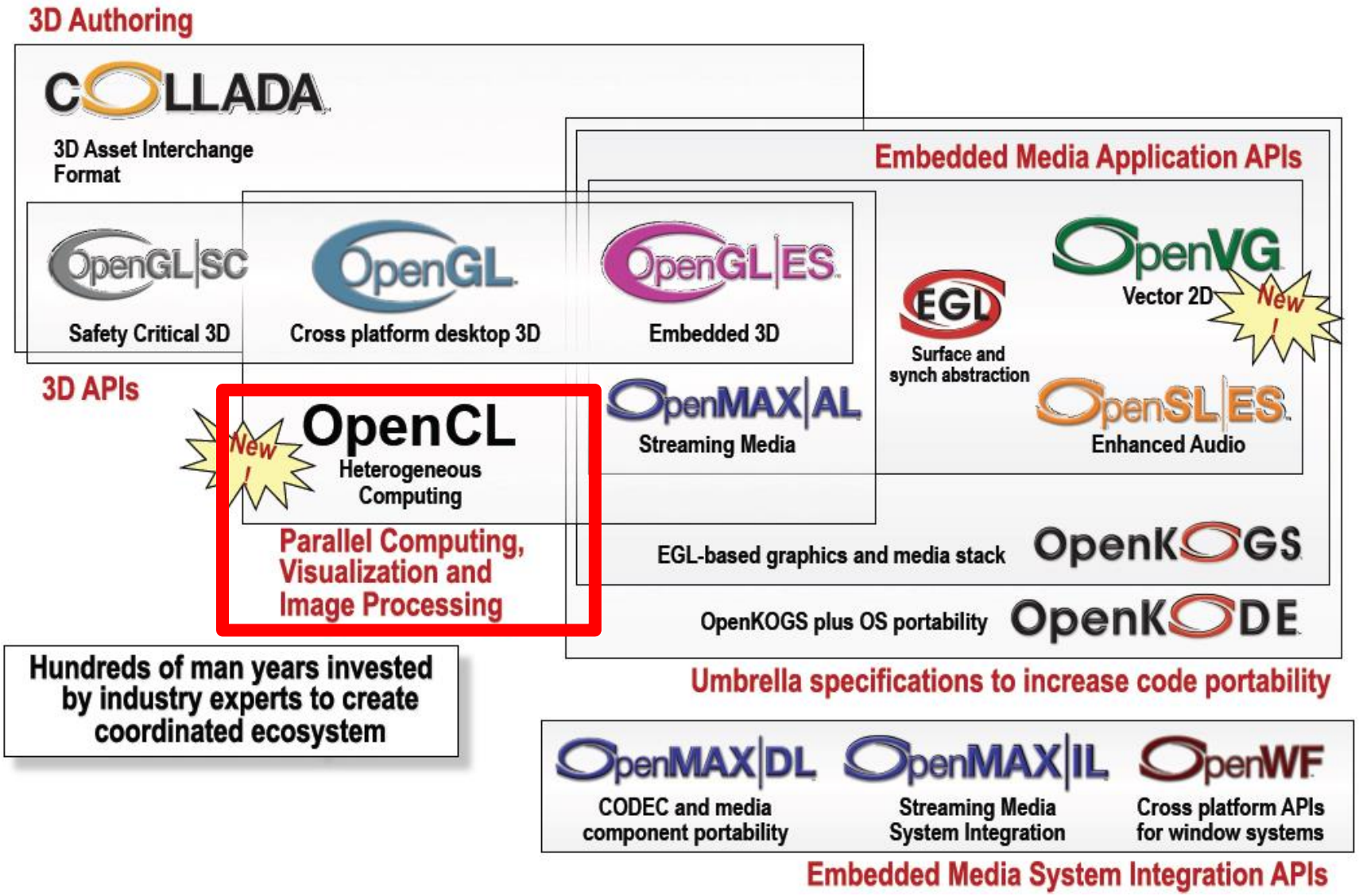
- **OpenCL Platform Model**
- **Micro-architecture implications**
- **An OpenCL RunTime on a Many-core system**
- **Summary**

Parallel Processing Platform

- Multi-core CPU
- DSPs
- GPU
  - Data parallelism
- Portability
  - OpenCL is a framework for building parallel applications that are portable across heterogeneous platforms.

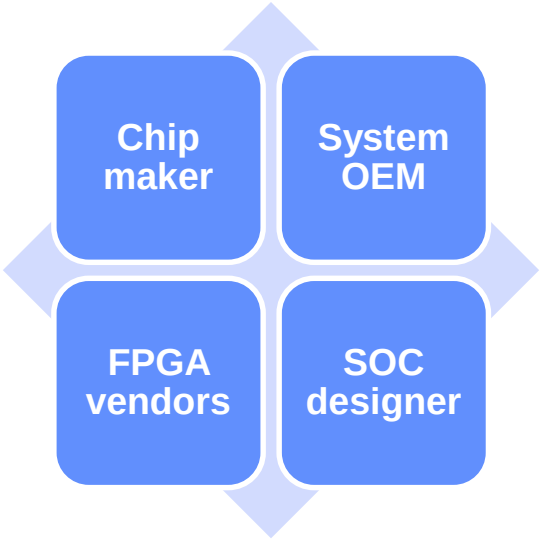


Khronos Standards Ecosystem



Key Players

- NVIDIA
- Apple
- ARM
- IBM
- Intel
- AMD
- BROADCOM
- SAMSUNG
- ……



OpenCL Specification

OpenCL

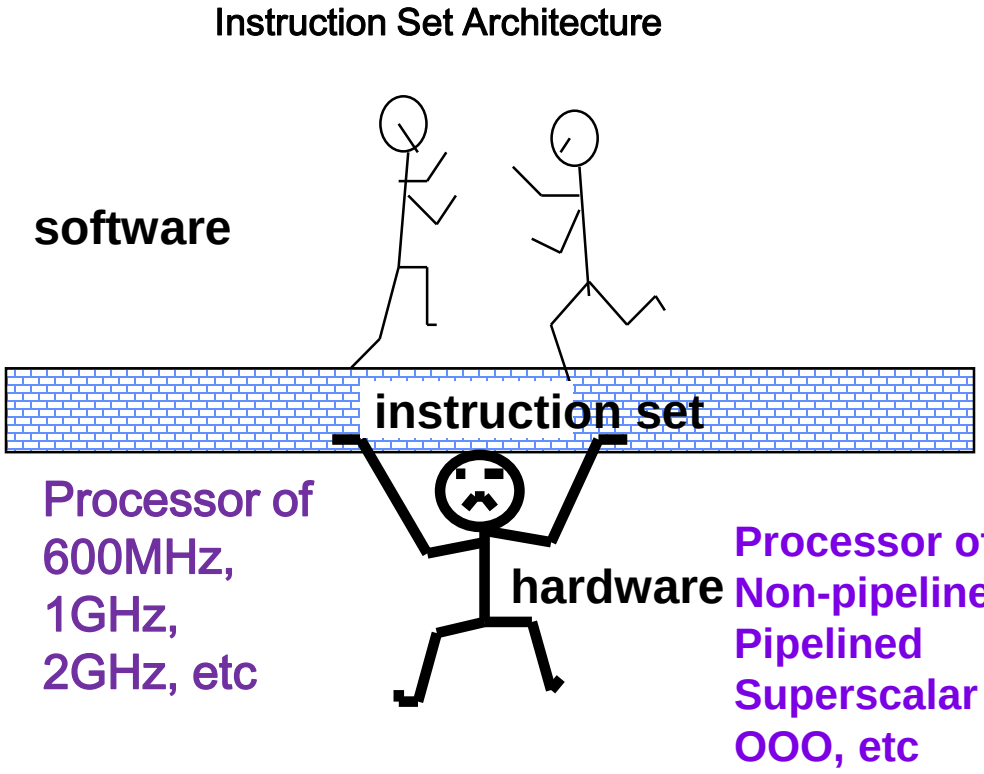
- **Open Computing Language** (**OpenCL**) is a framework for writing programs that execute across heterogeneous platforms consisting of central processing unit (CPUs), graphics processing unit (GPUs), and other processors.

- **Language:** OpenCL defines an OpenCL C language for writing *kernels* (functions that execute on OpenCL devices) also define many built-in functions.

- OpenCL defines application programming interfaces (APIs
  - **Platform Layer API: hardware abstraction layer; query, select and initialize compute devices; create compute contexts and work queues**
  - **Runtime API: execute kernels, manage thread scheduling and memory resources**

Common Hardware Abstraction

- **Abstraction makes life easier for those above**
- **An OpenCL device is viewed by the OpenCL programmer as a single virtual processor.**



From Sequential to Parallel

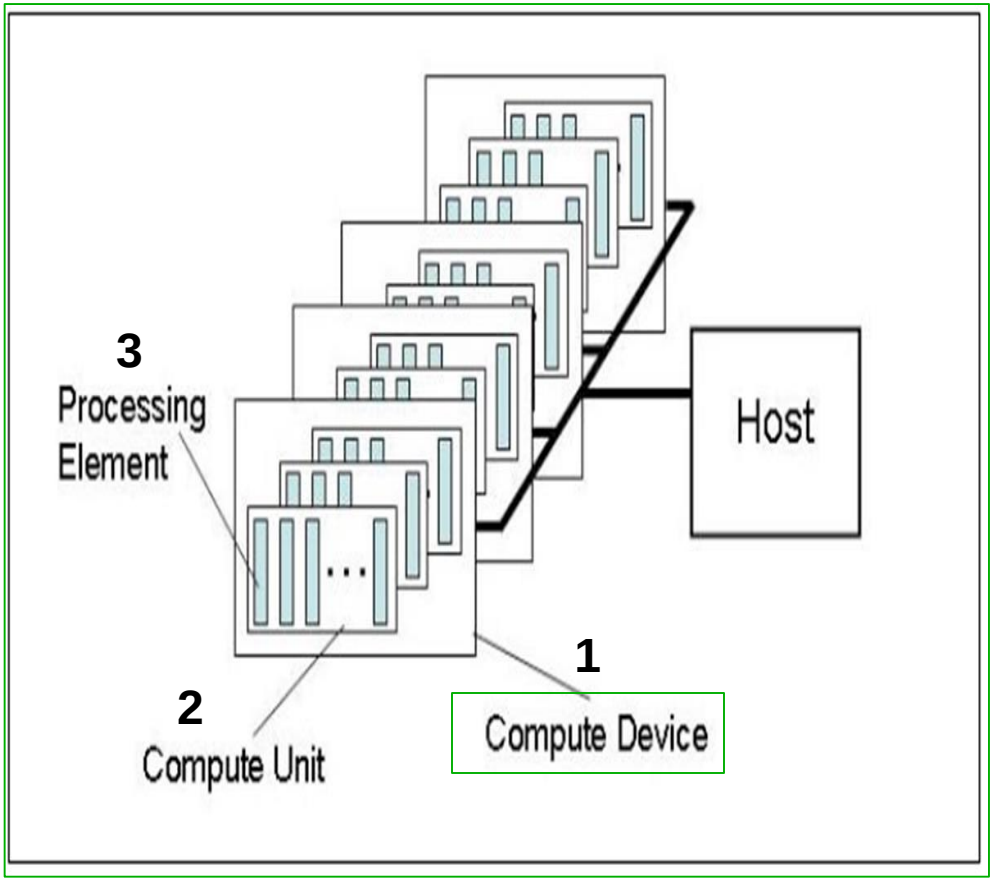
```c
/* C function */
void array_add(int* a, int* b, int* c)
{
    for(i=0;i<array_size;i++)
    {
        c[i] = a[i] + b[i];
    }
}
```

Define N-dimensional computation system (N=, 1, 2, 3)
Execute a kernel code for each point in the system

```c
/* OpenCL */
_kernel void add(_global int* a, _global int* b, _global int* c)
{
    int i = get_global_id(0);
    c[i] = a[i] + b[i];
}
```

OpenCL Platform Model

- **Hierarchical System**
- **A host with one or more compute devices**
  - **A compute device: one or more compute units**
  - **A compute unit: one or more processing elements**



OpenCL Execution Model

Application runs on a host

Host submits the work to the compute devices through enqueue operation

Work item: basic unit of work (the work or data a thread to work on)

Kernel: the code for a work item

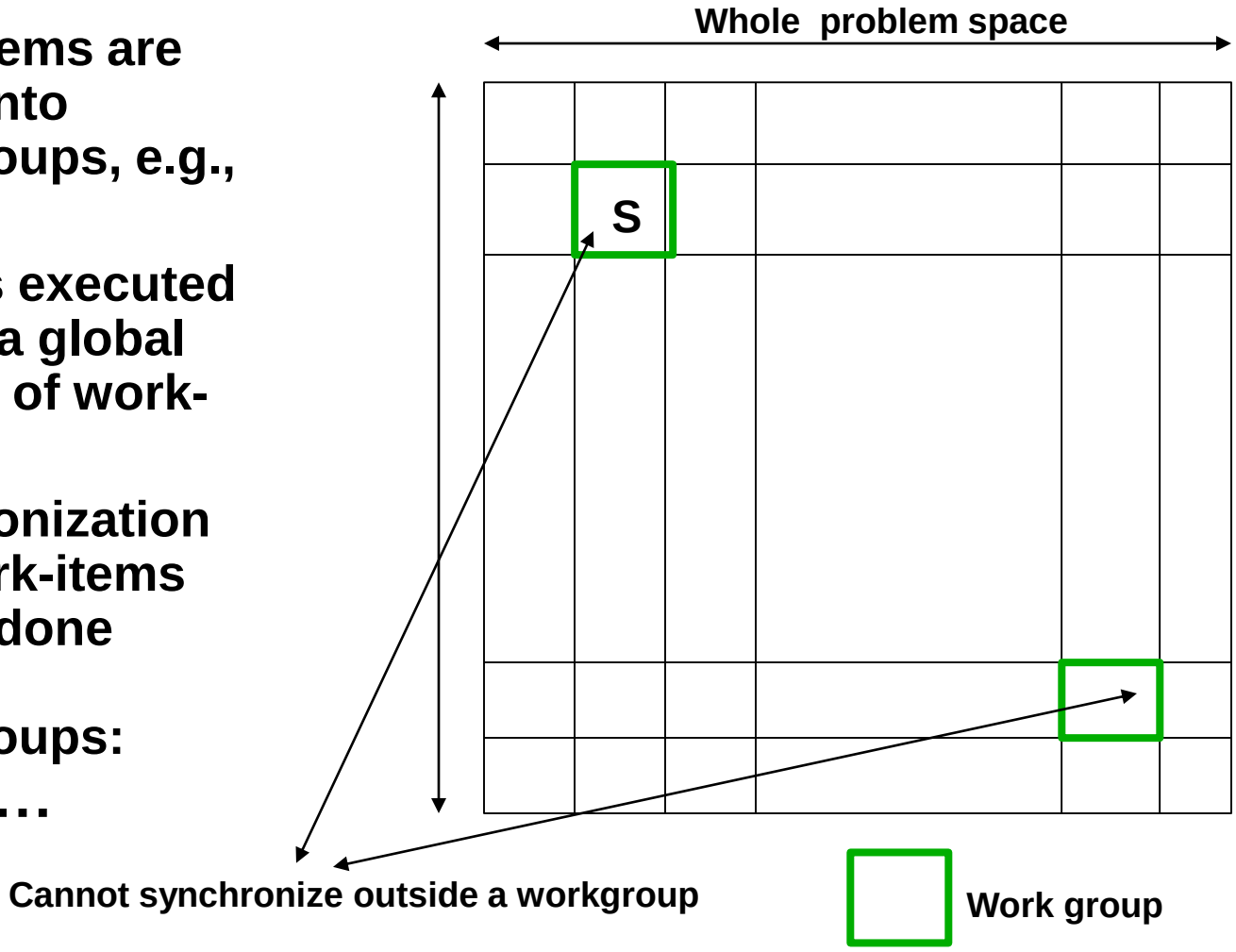
- An invocation of the kernel is a thread

Program: Collection of kernels + lib functions

Context: the environment in which work-items execute, including command queues, memories used, and devices

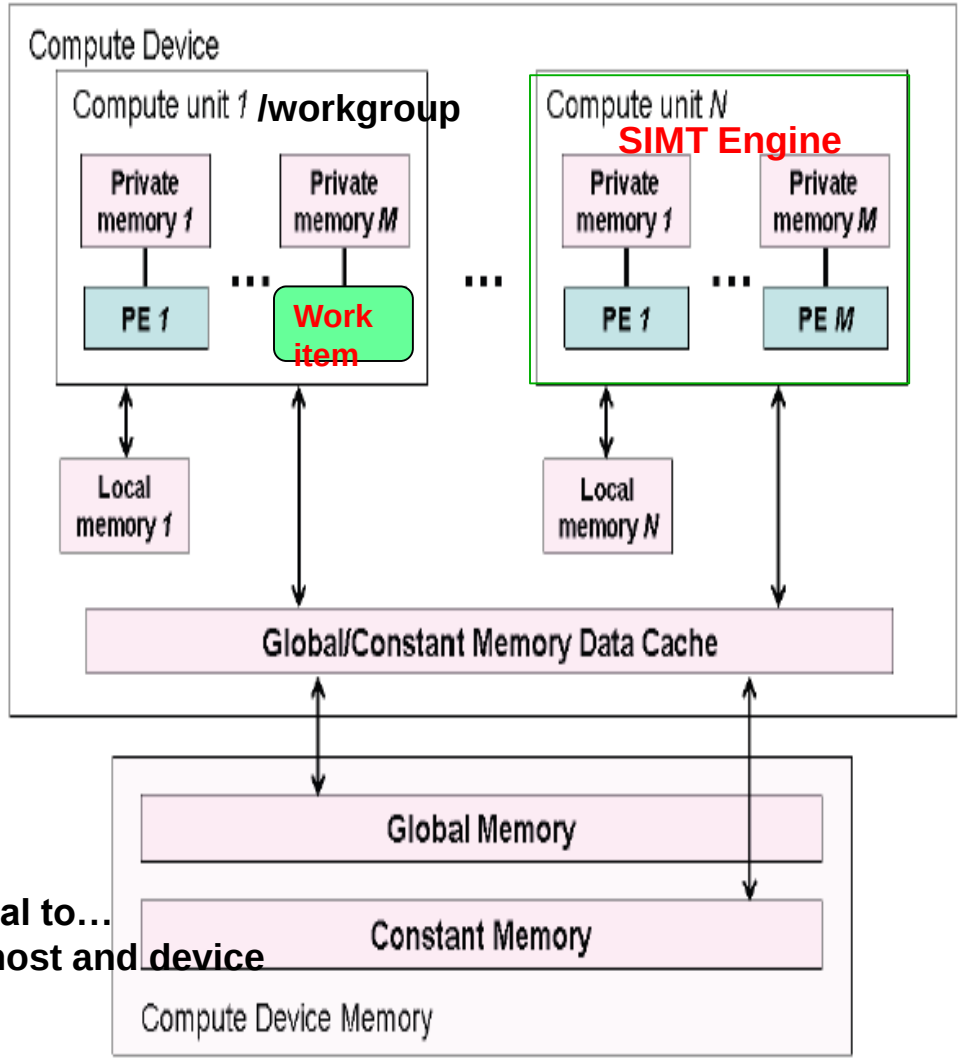
Work-items specified in N-dimension: N=2

- Work-items are group into workgroups, e.g., 64x64
- Kernels executed across a global domain of work-items
- Synchronization btw work-items can be done within workgroups: barrier …



OpenCL Memory Model

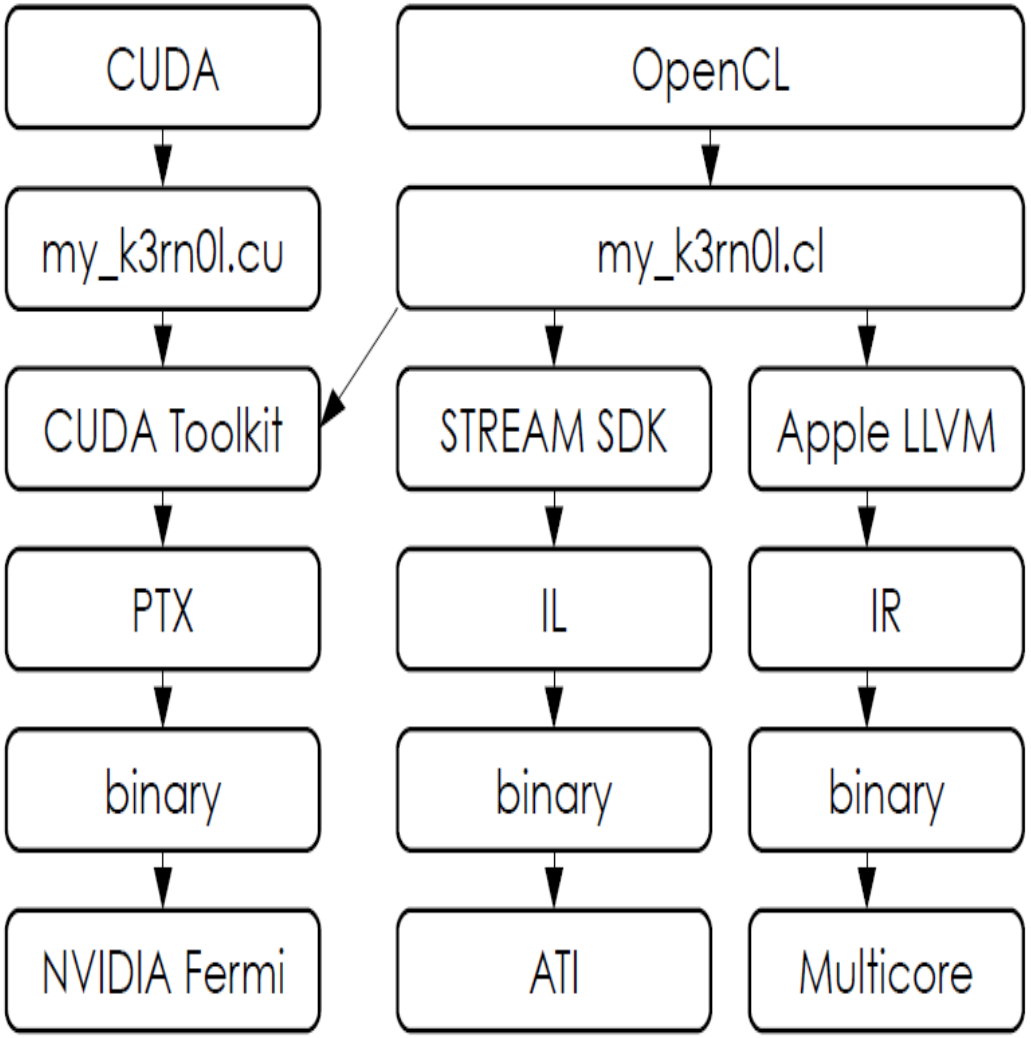
- **Private memory: per work-item**
- **Local memory: shared within a workgroup**
- **Global/constant memory: visible to all workgroups**
- **Host memory : on the host CPU**



OpenCL 1.2: move data from host to global to…
OpenCL 2.0: shared virtual memory btw host and device

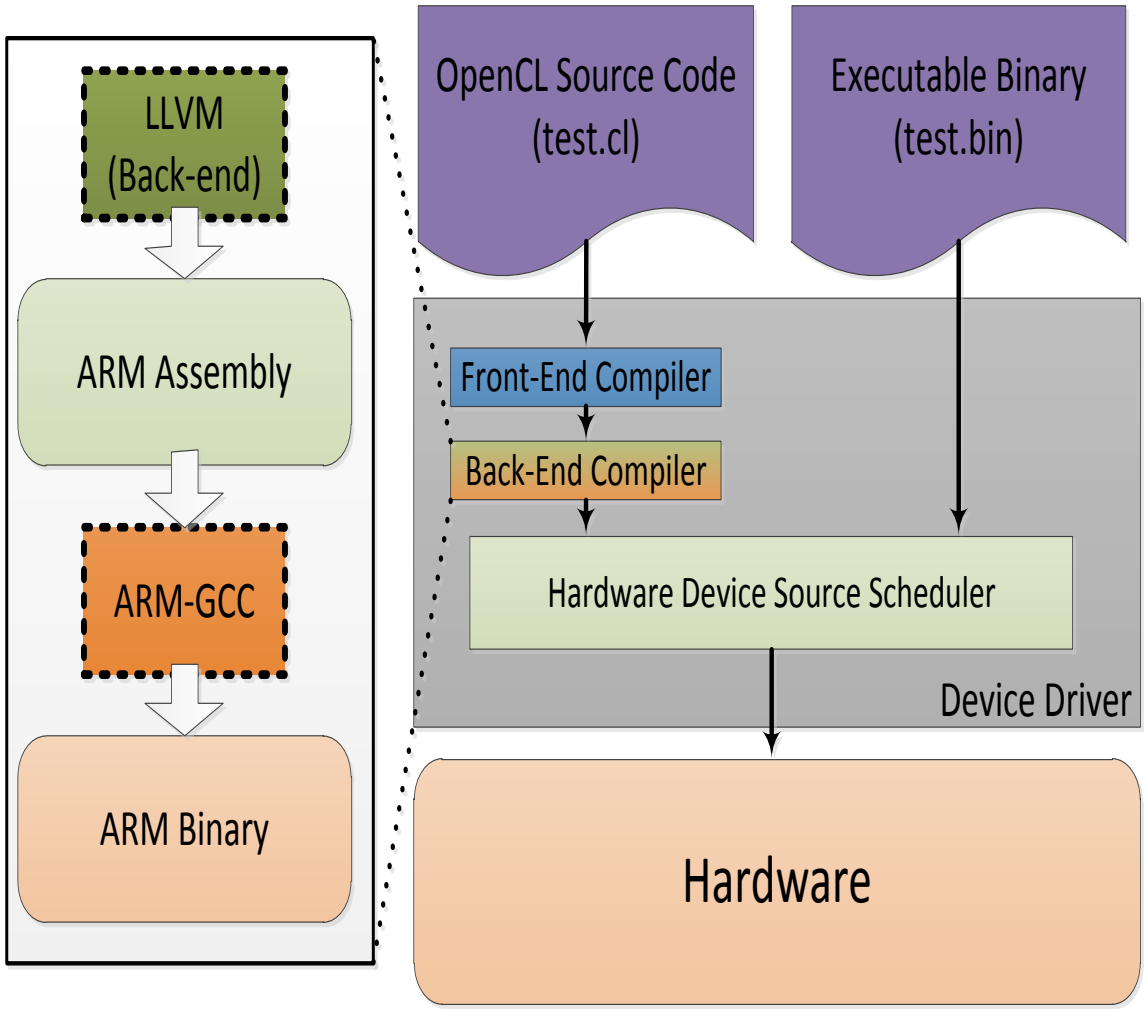
Compilation

- **Dynamic runtime compilation model**
- **Intermediate Representation (IR)**
  - **Assembly based on a virtual machine (done once)**
- **IR is compiled into machine code**
  - **App loads IR and compiles it.**



LLVM example

- **Low Level Virtual Machine**
  - **Front-end compiler: source code to IR**
  - **Back-end compiler: IR to target code**
  - **IR to CPU ISA**
  - **IR (PTX) to GPU ISA**



A Simple Example

- **Built-in function get_global_id(0) returns the thread id.**
- **Each PE executes the same kernel code and uses its thread id to access its data. Hence, SIMT.**

.CL code

```
/* OpenCL */
_kernel void add(_global int* a, _global int* b, _global int* c)
{
    int i = get_global_id(0);
    c[i] = a[i] + b[i];
}
```

Get processor ID. This is one dimension example.

host C code: Run control thread

```
void array_add()
{
    /* create memory object a,b,c;*/
    global_size[0] = array_size;
    dim = 1;
    clEnqueueNDRangeKernel(queue, add, dim, NULL, &global_size, &local_size, 0, NULL, NULL);
    /* Copy the result from device */
}
```

memory objects created using clCreateMemObj

Create this number of threads

Workgroup size

Host control thread example

Setup Execution Environment

Create Memory/Program Objects

Prepare Parameters and Copy Data for Execution

Enqueue for Execution and Read Back

Setup Execution Environment: prepare environment

Get Available Devices

- clGetPlatformIDs
  - Return the number of available platforms and the pointers of the platforms
- clGetDeviceIDs
  - Return the list of devices available on a platform

Create Context

- clCreateContext
  - Return the pointer of context structure used by runtime for managing objects such as memory, command-queue, program and kernels

Create Command Queue

- clCreateCommandQueue
  - Return the pointer of command-queue that programmer uses to queue a set of operations

Create Memory/Program Objects

Create Memory objects

- clCreateBuffer
  - Return the pointer of the memory object which contains the relationship between host memory and device memory region

Create Program and Kernel Objects

- clCreateProgramWithSource
  - Use the CL source code to generate a program object
- clBuildProgram
  - Compile the source code of the target program object; each program has more than one kernel source
- clCreateKernel
  - Designate the kernel that is going to run

Prepare Parameters and Copy Data for Execution

Setup Kernel Parameters

- clSetKernelArg
  - Set up the parameters of the kernel function

Copy Data from Main Memory to Target Device

- clEnqueueWriteBuffer
  - Write the data from main memory to target device

Enqueue for Execution and Read Back

Execute the Kernel in N-dimension

- clEnqueueNDRangeKernel
  - Declare a N-dimensional work-space (global_size) for executing
  - Subdivide the work-space into work-group by means of setting local_size

Read Back Results from Target Device

- clEnqueueReadBuffer
  - Read the data from target device to main memory

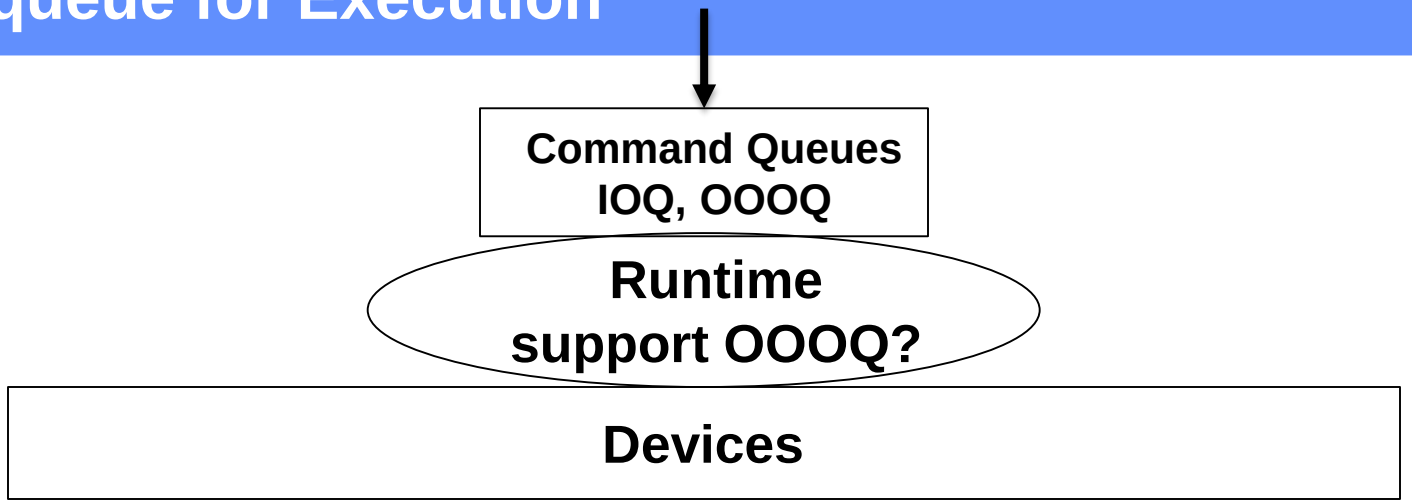
Executed in-order or out-of-order

Setup Execution Environment

Create Memory/Program Objects

Prepare Parameters and Copy Data for Execution

Enqueue for Execution



Welcome to this talk

- OpenCL Platform Model
- Micro-architecture implications
- An OpenCL RunTime on a Many-core system
- Summary

Now Device

- **Architecture implication for OpenCL Program Model**
  - **SIMT ISA**
  - **SIMT instruction scheduling**

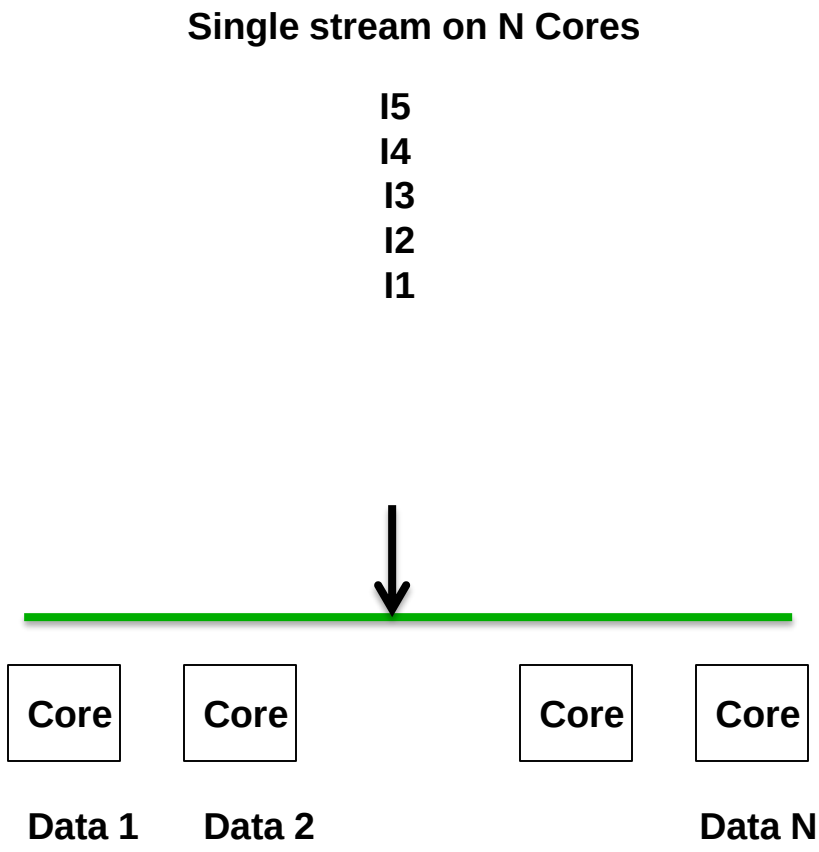
SIMT Machine

- **What architecture features are useful/critical for OpenCL-based computation?**
- **SIMT: single instruction multiple threading**
- **Same instruction stream works on different data**

```
/* OpenCL */
_kernel void add(_global int* a, _global int* b, _global int* c)
{
    int i = get_global_id(0);
    c[i] = a[i] + b[i];
}
```

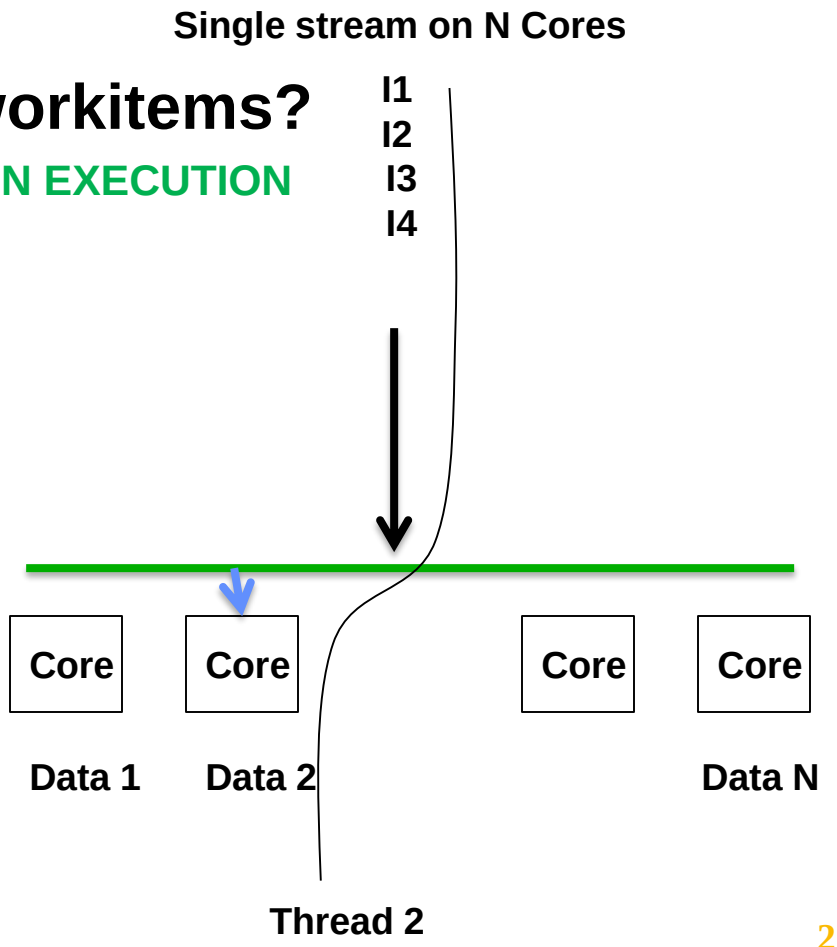
Single Instruction Multiple Threading

- **Single Instruction Multi-Threading**
- **A thread == a workitem**
- **Get one instruction and dispatch to every processor units.**
- **Fetch one stream -> N threads (of the same code) in execution**
- **Each thread is independent to each other.**
- **All cores execute instructions in lockstep mode.**



SIMT: Single Instruction Multiple Threading

- Clarify what is what
- What is S?
- What are threads or workitems?
  - AN INSTRUCTION STREAM IN EXECUTION



Fetching Mechanism for SIMT

Instruction Fetching

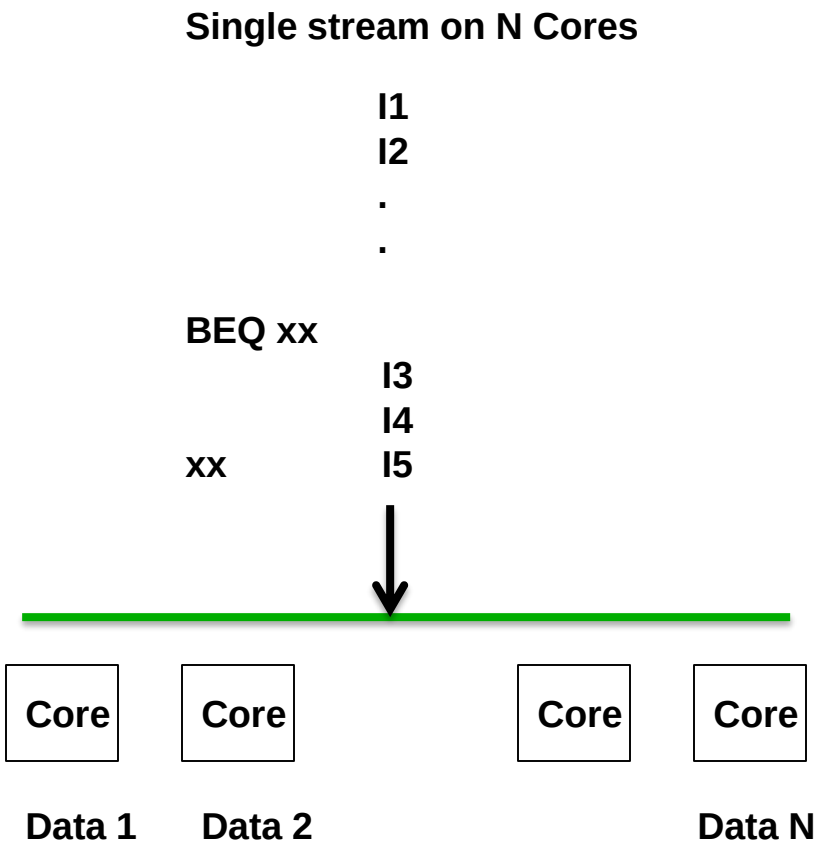
- Need an instruction fetcher to let each core or PE get their instruction
- Each PE may free run also.

Data Fetching

- Need an efficient way to get per-PE's data from global memory (DRAM).

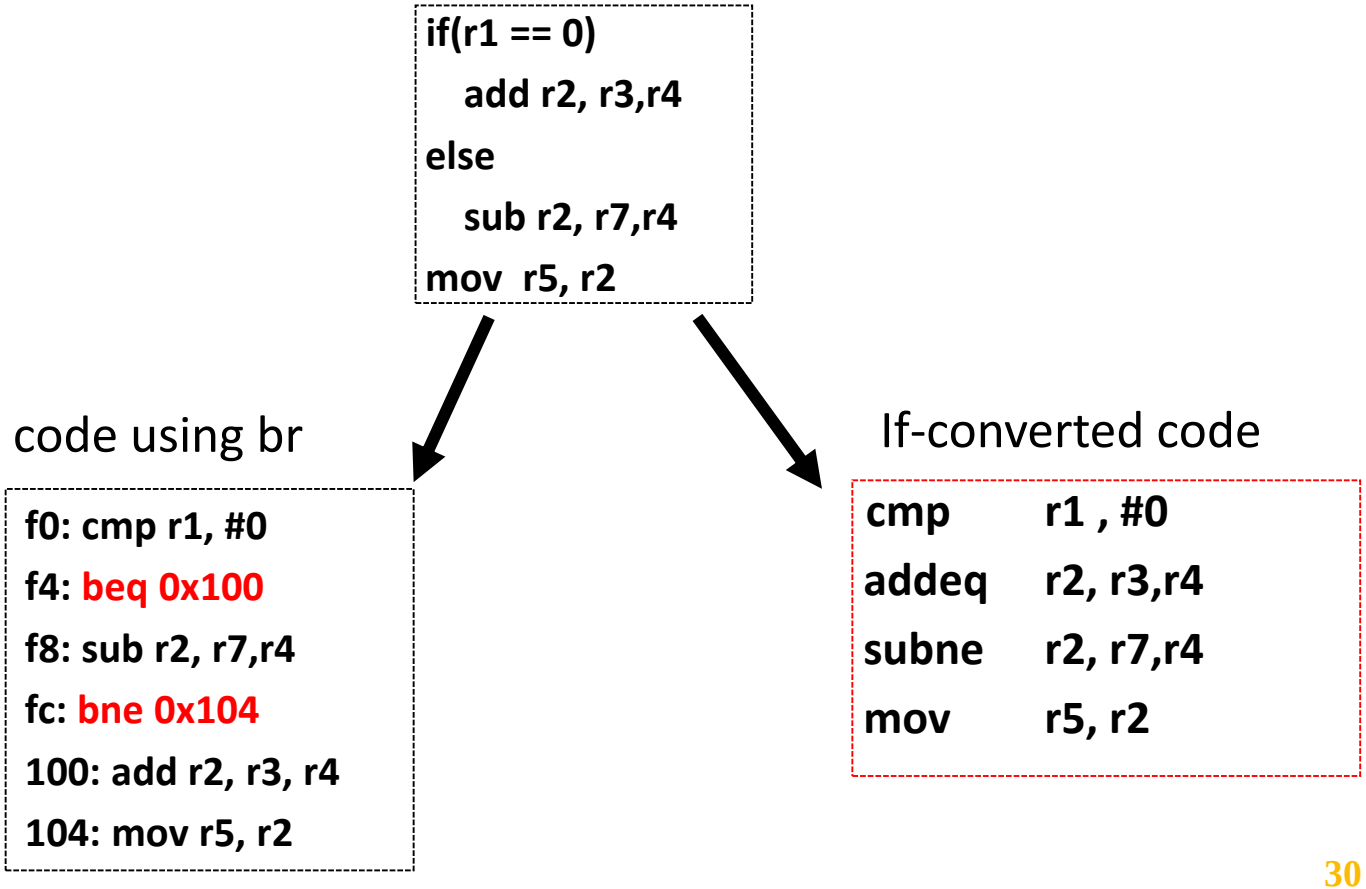
ISA issues for SIMT PE

- **Branch problem in SIMT**
  - **Can not use "regular branches" in SIMT because**
  - **If some PE gets I3 etc and some PE get I5,**
  - **then there is no single instruction stream anymore.**



Conditional Execution for SIMT

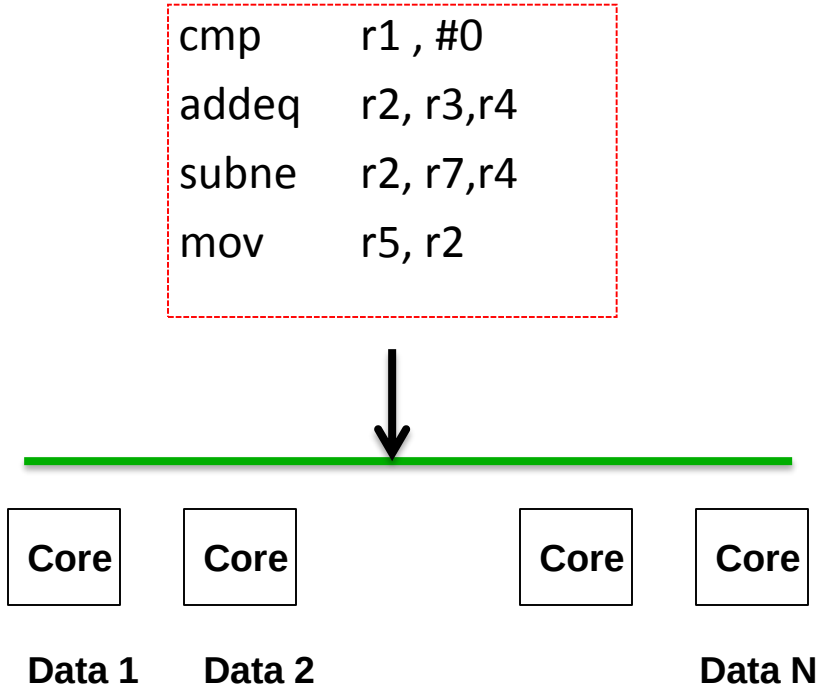
- If-conversion uses predicates to transform a conditional branch into a single control stream code.



ISA issues for SIMT

- **No branch in SIMT.**
- **Each PE simply executes the same instruction stream**
- **If the condition is met, commit the result otherwise nop.**
- **Problem:**
  - **Low Utilization of PE**
  - **Poor performance for branch rich App.**
  - **Poor performance in SISD: clEnqueueTask:** the kernel is executed using a single work-item.

Single instruction stream on N Cores

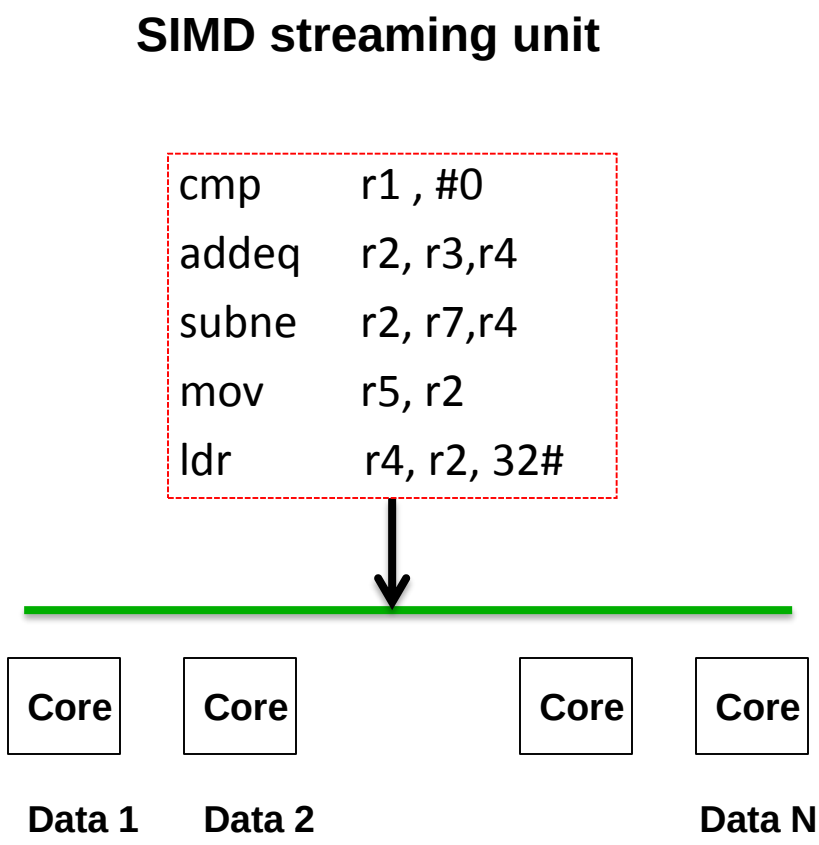


Now Device

- **Architecture implication of OpenCL Program Model**
  - **SIMT ISA**
  - **SIMT instruction scheduling**

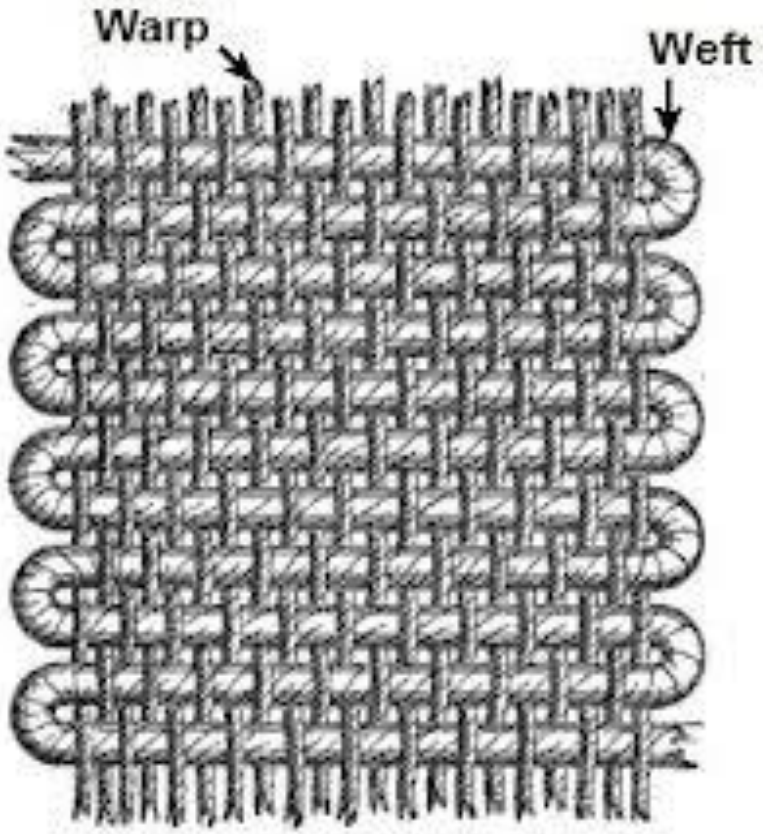
SIMT: SIMD Streaming Machine

- **Pipelined PE/Core**
- **How to tolerate long latency instructions?**
  - **Cache miss**
  - **Complex integer instructions**
  - **Expensive floating point operations**



Multithreaded Categories

Warp



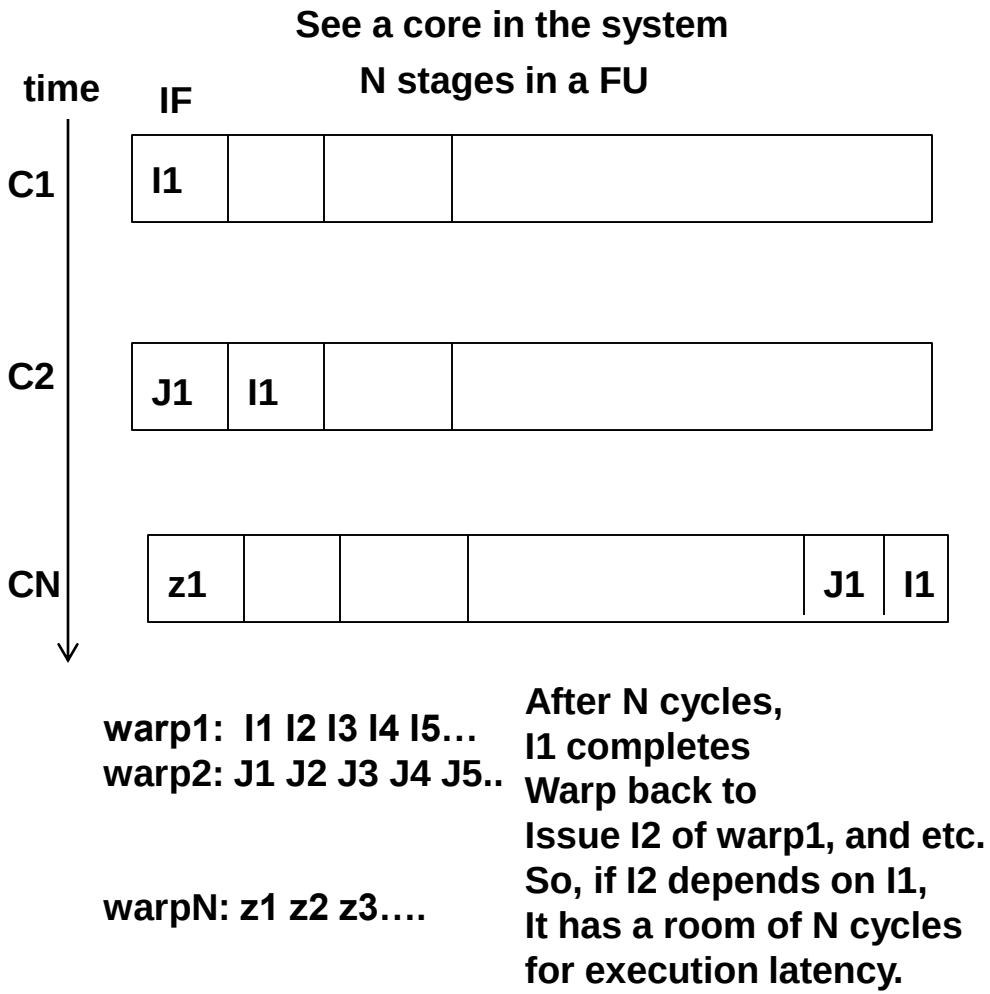
Terminology: Barrel threading

- **Interleaved multi-threading**
- Cycle i+1: an instruction from instruction stream (warp) A is issued
- Cycle i+2: an instruction from instruction stream (warp) B is issued
- The purpose of this type of multithreading is to remove all data dependency stalls from the execution pipeline. Since one warp is independent from other warps.

- **Terminology**
- This type of multithreading was first called *Barrel processing*, in which the staves of a barrel represent the pipeline stages and their executing threads. *Interleaved* or *Pre-emptive* or ***Fine-grained*** or *time-sliced* multithreading are more modern terminology.

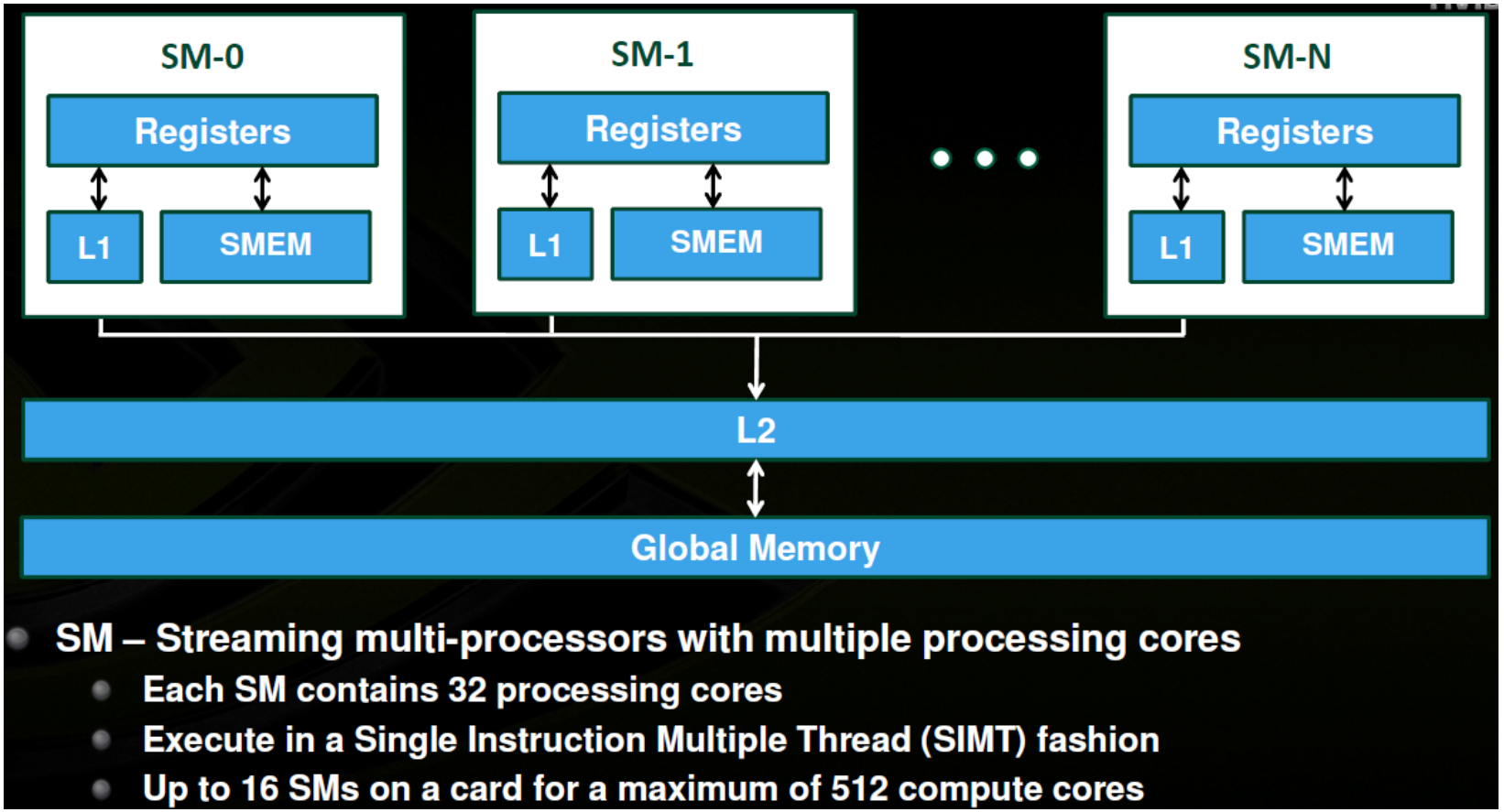
Warp scheduler

- SIMT machine fetcher fetches warps of instructions and store them into a warp queue.
- Warp scheduler issues (broadcasts) one instruction from a ready warp to the PEs in the SIMT machine.



Example: Fermi GPU Architecture

SMEM: shared memory in Fermi term, but this is actually a private local scratchpad memory for a thread block communication (workgroup)
Data memory hierarchy: register, L1, L2, global memory
L1 + Local Scratchpad = 64KB configurable



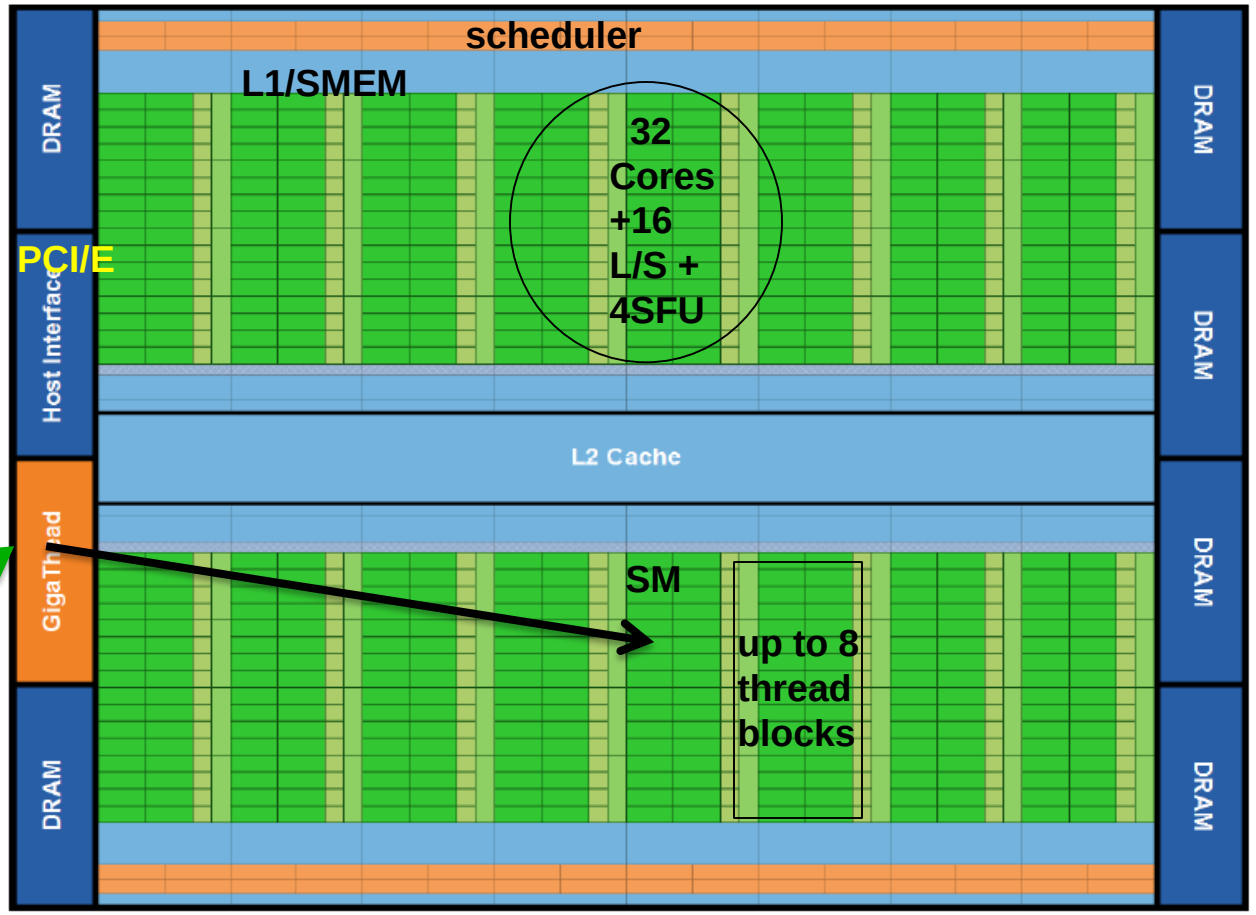
Example: Fermi Floor Plan

40-bit address space
40 nm TSMC
3 x 10^9 T
> Nehalem-Ex(2.3)
1.x GHz

16 multithreaded SIMD processors

Thread Block Distributor: Workgroup distributor

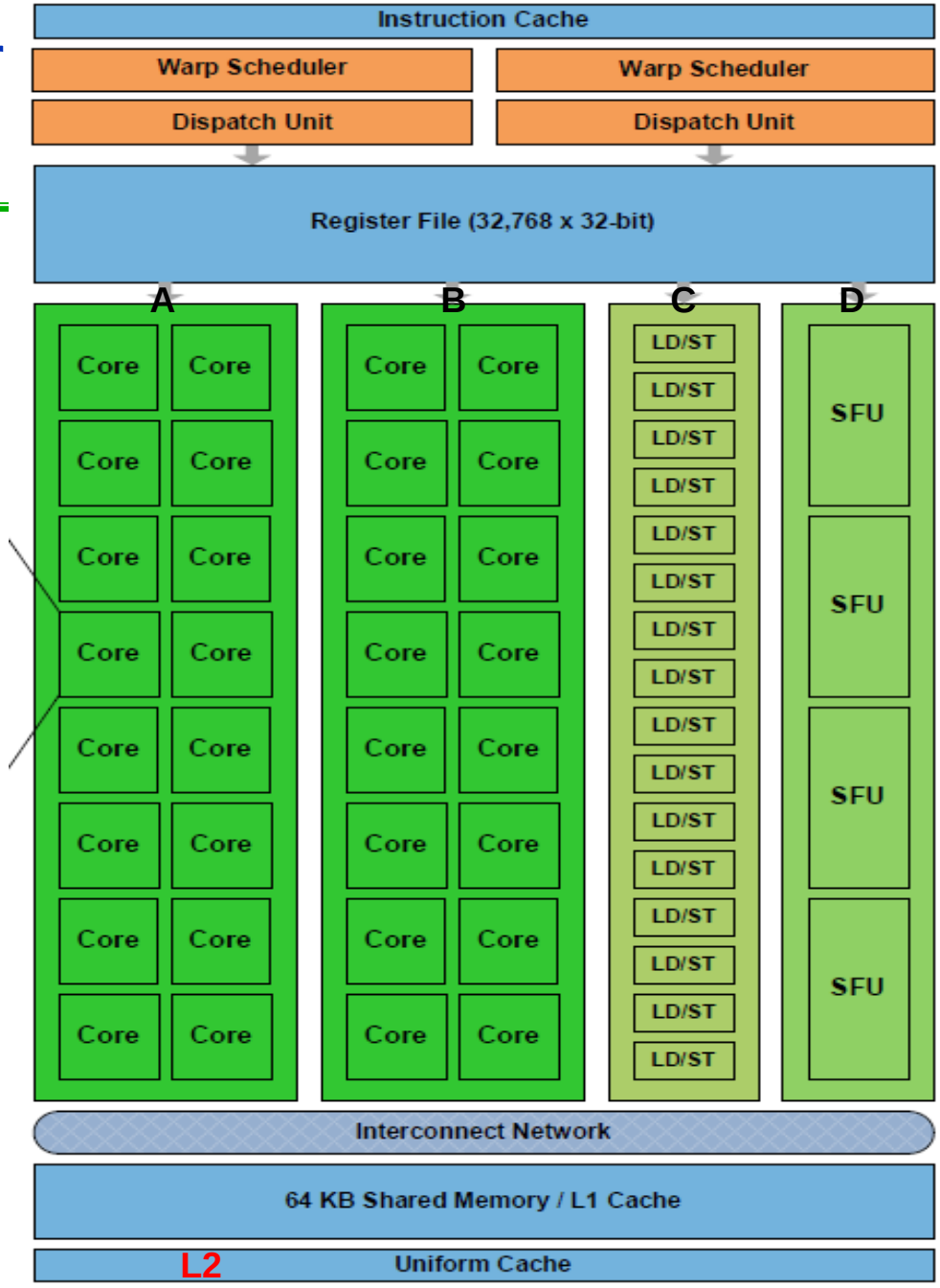
This GPU is a multiprocessor composed of multithreaded SIMD processors



Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).

A Streaming Multiprocessor ie., a Multithreaded SIMD Processor

- An SM consists of 32 CUDA cores + some 16 Load/Store unit + 4 special functional units
- Registers: 32K x words
- L1 data cache private to each SM
- L1 Instruction cache
- L2 unified for data and texture, instruction(?), shared globally, coherent for all SMs.
- Instruction dispatch
  - (A, B) fs
  - (A+B) fd
  - (A, C)
  - (B, C)
  - (A, D)
  - (B, D), (C, D), etc



Fermi Streaming Multiprocessor (SM)

Warp Scheduler in Fermi

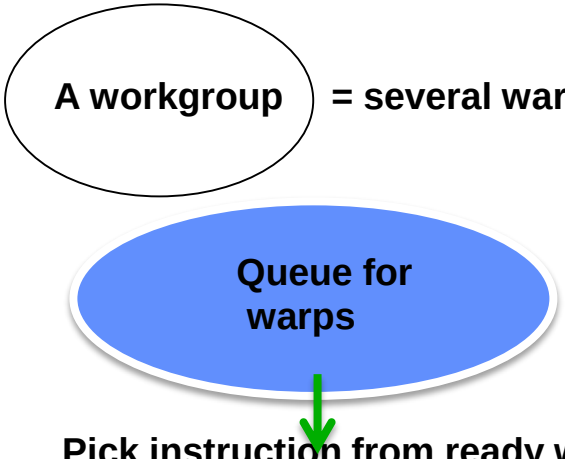
32 threads form a warp

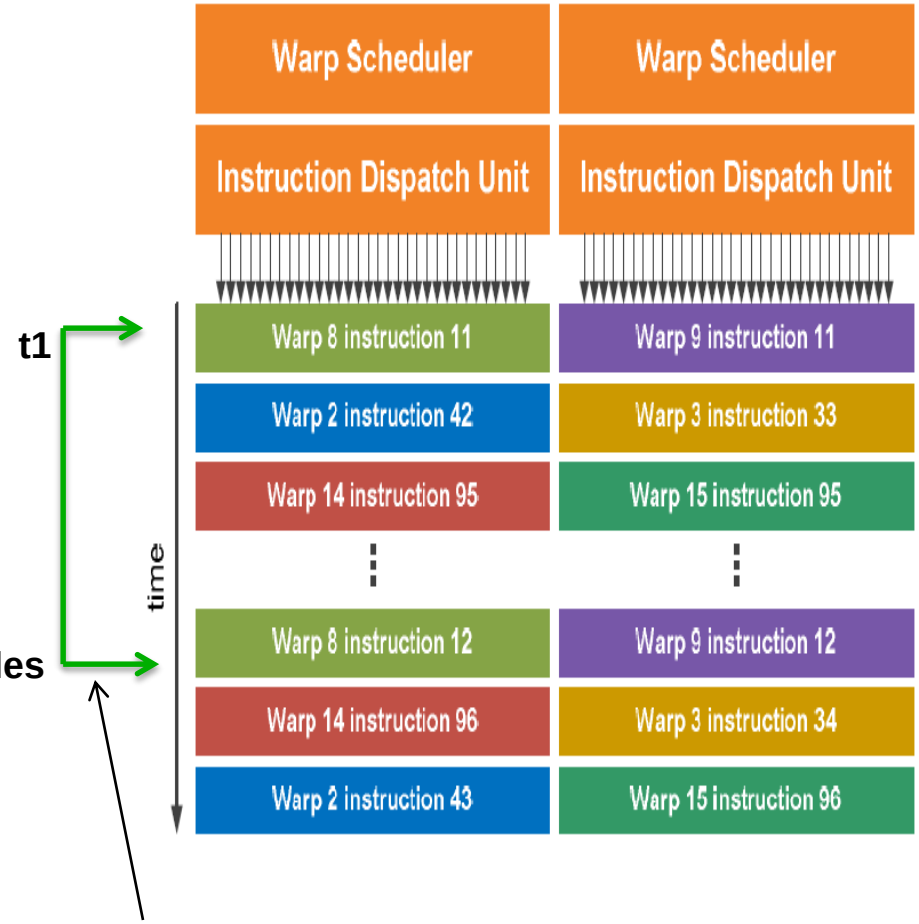
Instructions are issued per warp

If an operand is not ready the warp will stall

- Context switch between warps when stalled
- Context switch must be very fast

- **A warp is simply an instruction stream of SIMD instructions.**





Welcome to this talk

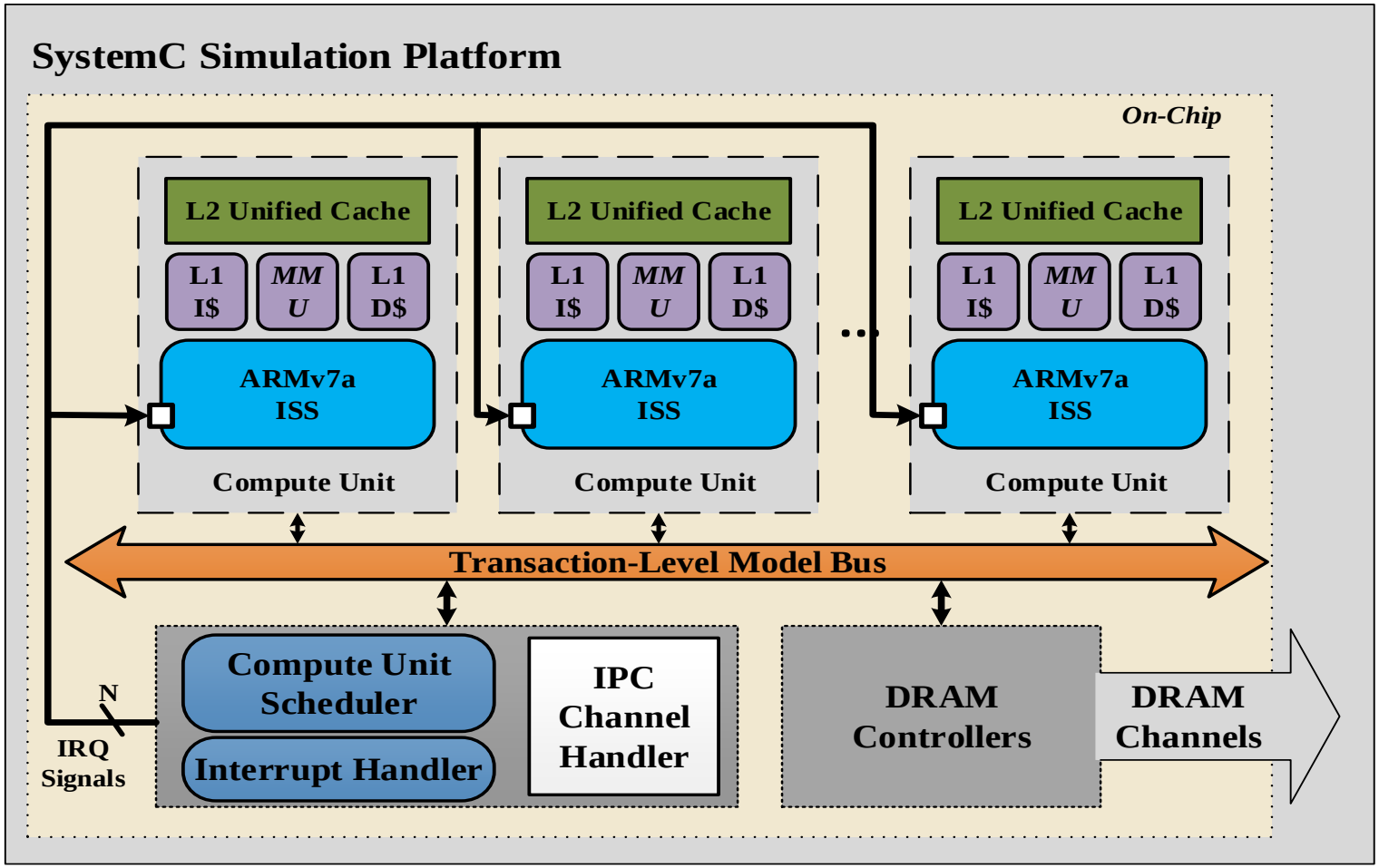
- OpenCL Platform Model
- Micro-architecture implications
- An OpenCL RunTime on a Many-core system
- Summary

Runtime Implementation Example

- On an 1-to-32 ARM-core system,
- Build an OpenCL runtime system
  - Resource management + On-the-fly compiling
- To evaluate
  - Work-item execution methods
  - Memory management for OpenCL memory models

Target Platform – ARM multi-core virtual platform

- **Homogeneous many-core with shared main memory**



OpenCL Runtime System – Software Stack

OpenCL Application Programming Interface

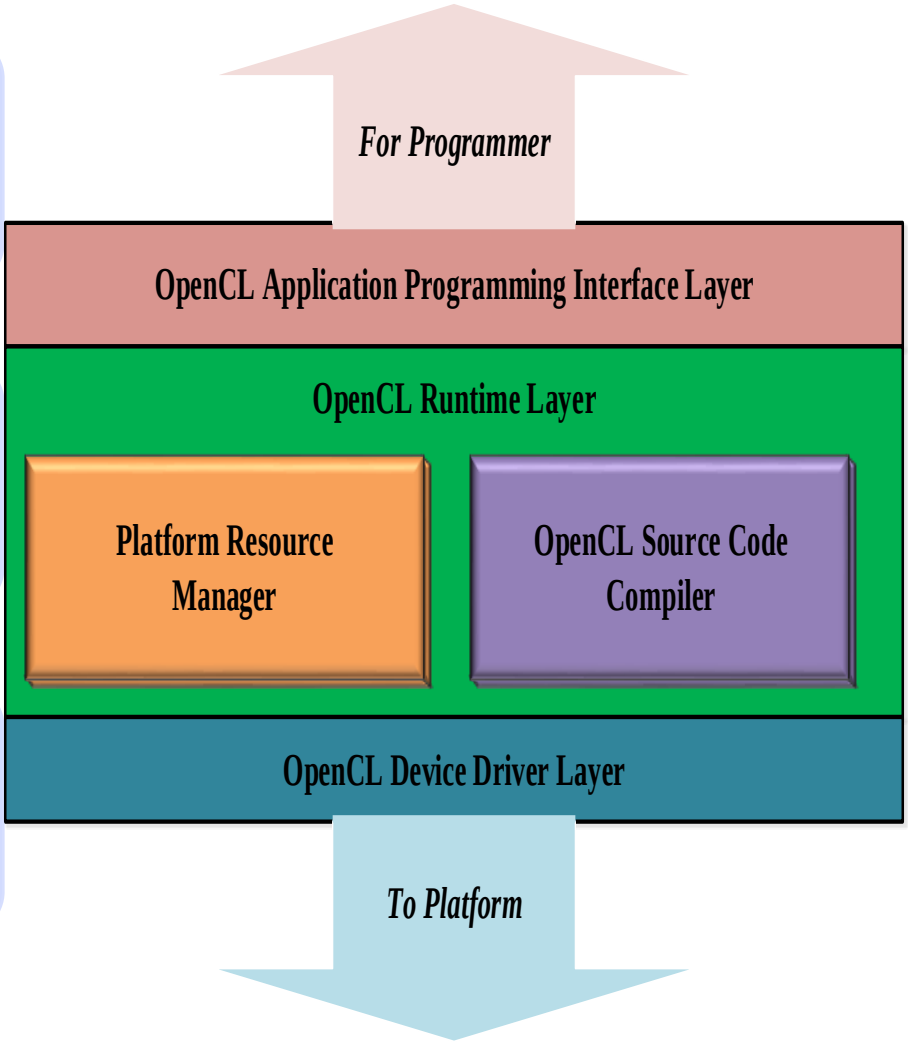
- A unified programming interface for various platforms

OpenCL Runtime System

- Resource manager
  - Device, memory, execution kernel … etc.
- CL source code compiling for target device

OpenCL Device Driver

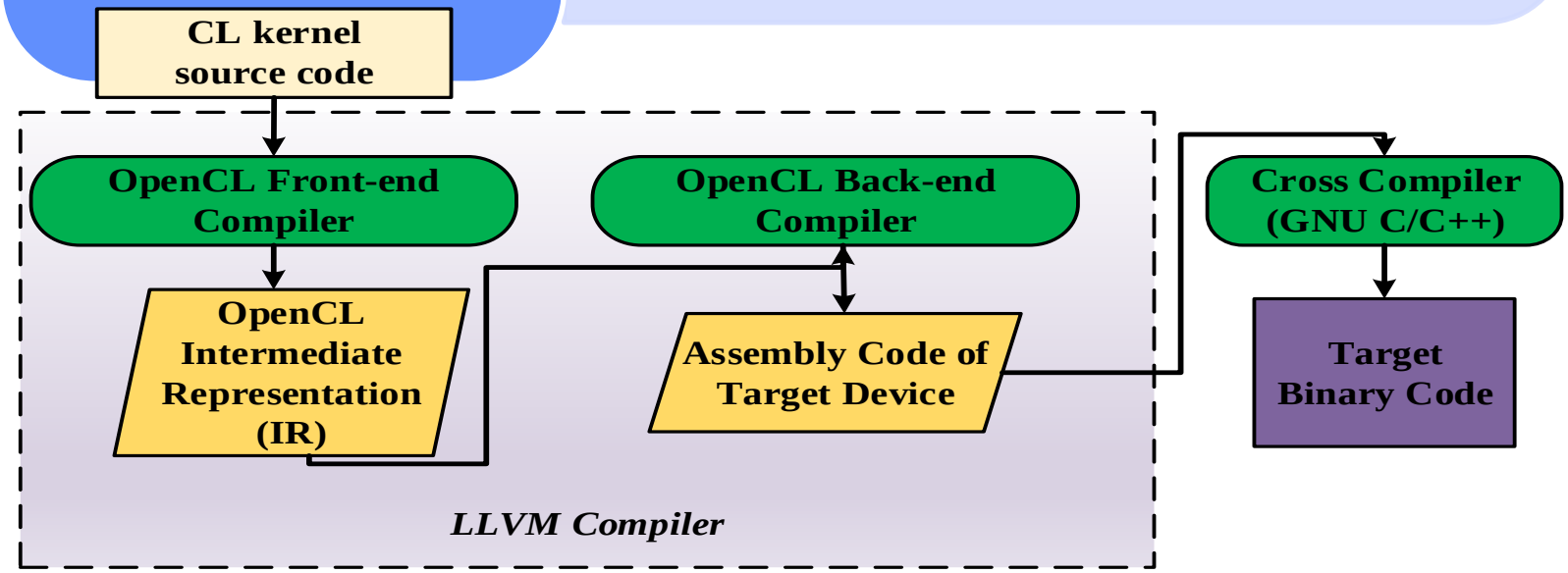
- Provide driver API to runtime



OpenCL Source Code Compilation

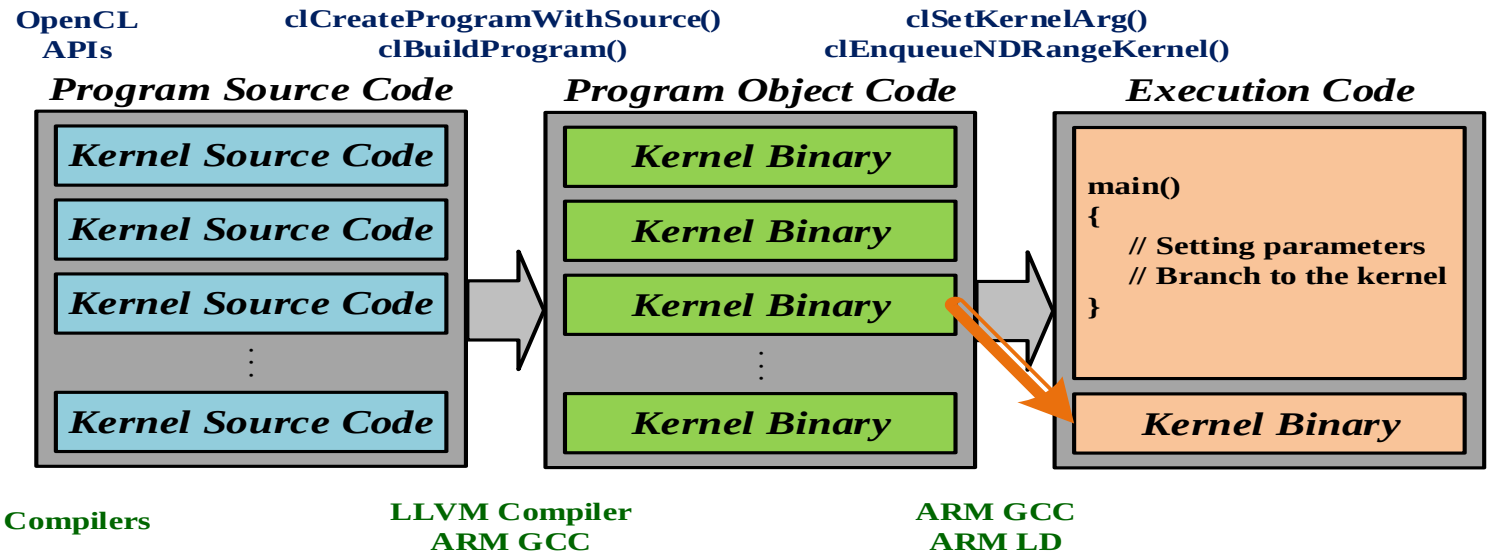
Translate the kernel code to binary code

- LLVM compiler framework
  - Front-end compiler: Intermediate representation
  - Back-end compiler: Assembly code of target device
- ARM Cross Compiler
  - Target Binary



Runtime: Program & Kernel Management

- More than one kernel in a program
  - clCreateProgramWithSource/clBuildProgram
    - » Use LLVM compiler and ARM cross compiler to build the object code by the program source code
  - clEnqueueNDRangeKernel
    - » This API decides the kernel which is going to run.
      - Object code linking

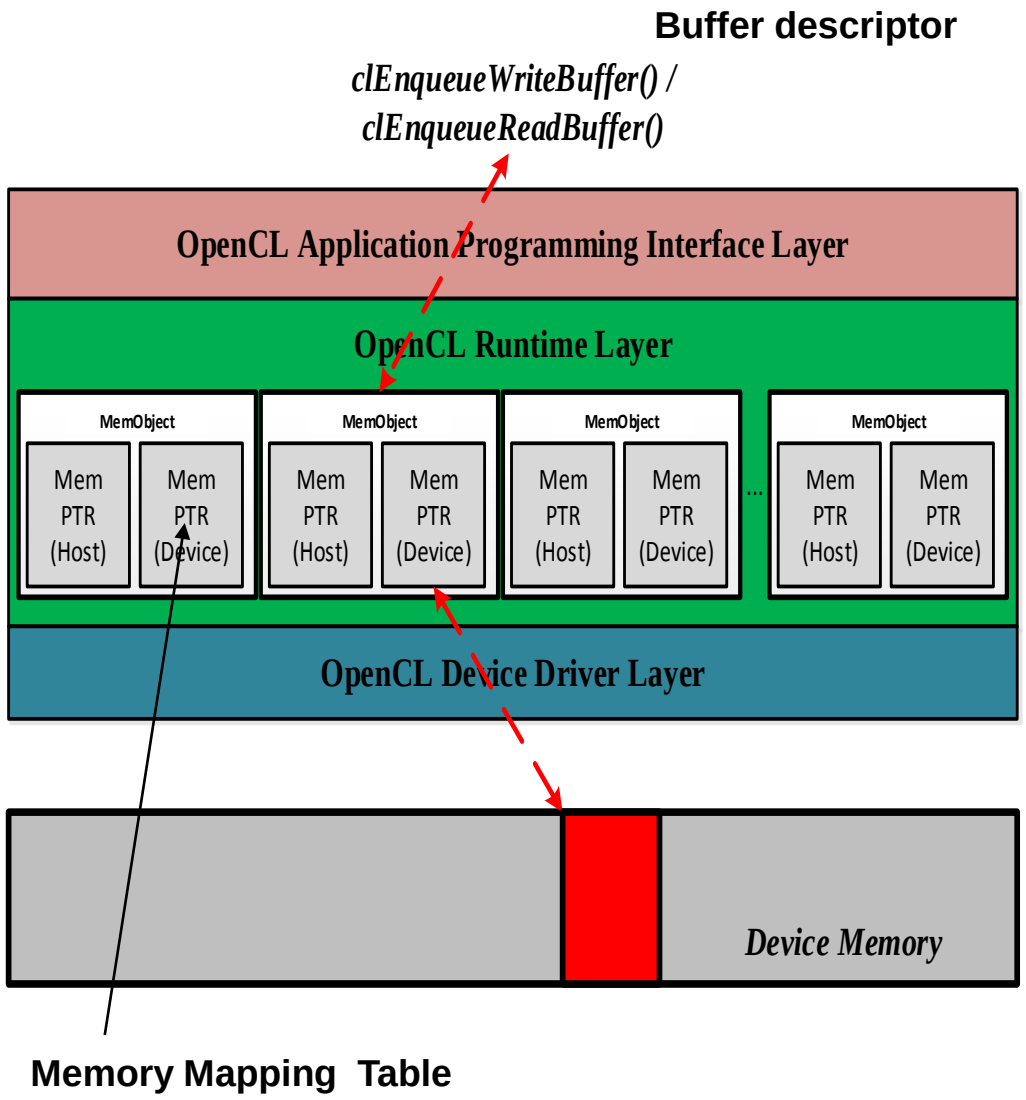


Runtime: Memory Mapping

- Mapping OpenCL Progam Memory to Physical Memories
  - Created by clCreateBuffer (Global, constant, local)
    - » Runtime system creates a memory object through memory allocation function provided by device driver. (Map physical to OpenCL memories)
    - » This API returns a pointer of the buffer descriptor for the mapping table. RunTime keeps this table.
  - Local memory can be also declared by kernel source
    - » LLVM compiler uses .bss section for variables declared with __local key word.
    - » Memory mapping in MMU set by work-item management thread per CPU core.
  - Kernel's private memory
    - » Use stack memory
    - » Stack set by work-item management thread

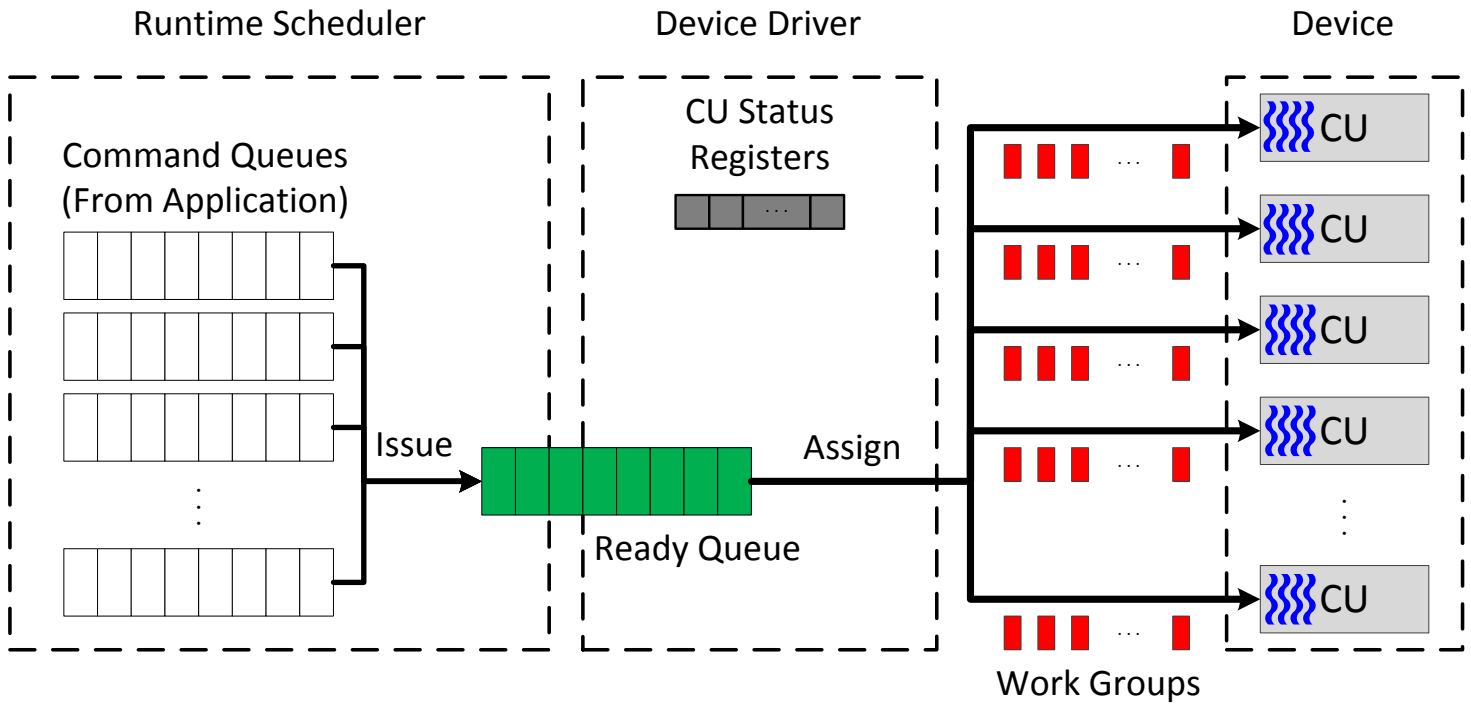
Runtime: Data transfer

- **Data transfer between host and target device by:**
  - **clEnqueueWriteBuffer**
  - **clEnqueueReadBuffer**
  - **For these API calls, the runtime system copies the data between host memory and target device memory through the mapping table kept in runtime.**



Runtime: Compute Unit Management

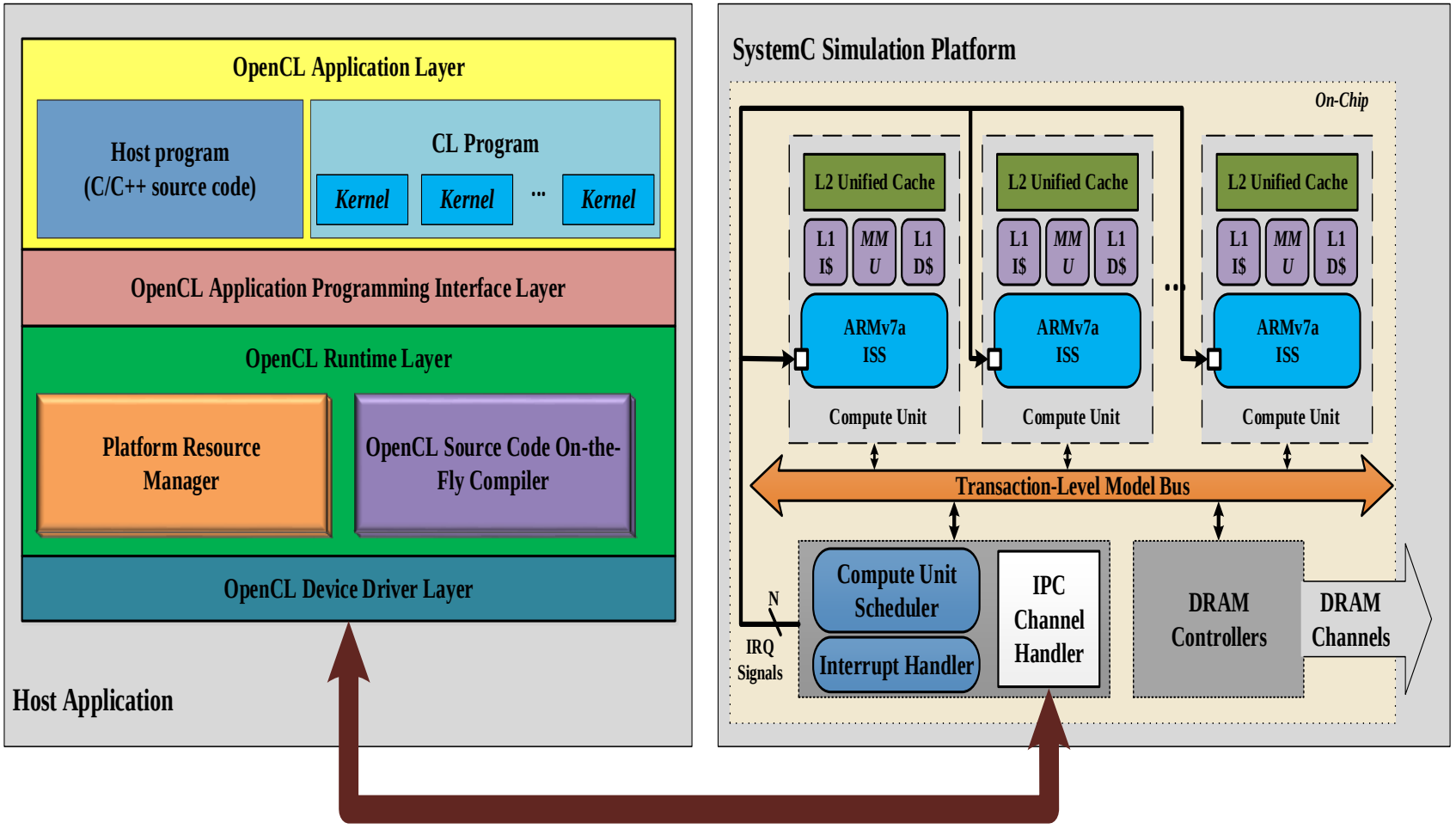
- **Each ARM core is mapped to a compute unit (CU).**
- **A CU executes a work-group at a time.**



Device and Memory Mapping

OpenCL Progam Model	Map onto CASLAB multi-core Platform
Host Processor	Host CPU (INTEL i7)
Host Memory	Host main memory
Compute Device	SystemC ARM Multicore (1 to 32 core)
Compute Unit	SystemC ARMv7a ISS
Process Element	Work-item coalescing to a thread running on an ARMv7a core
Global/Constant Memory	Mulit-core Shared Memory
Local Memory	Per ARM's memory (in shared memory)
Private Memory	Each Work-item's Stack Memory

Simulation Platform for OpenCL runtime development



Work-item coalescing

- **Work-items in a workgroup are emulated in a CPU core.**
- **Context switching overheads occur when switching work-item for execution.**
  - **Combine the work-items in a workgroup in to a single execution thread.**
  - **Need to translate the original CL code.**

New Features in OpenCL 2.0

- **OpenCL 1.0**
- **OpenCL 1.1**
- **OpenCL 1.2**
- **OpenCL 2.0 (July, 2013)**
  - Extended image support (2D/3D, depth, read/write on the same image, OpenGL)
  - Shared virtual memory
  - Pipes (transfer data btw multiple invocation of kernels, enable data flow operations)
  - Android Driver

Summary

- From Application
  - OpenCL
- To Technology
  - Architectural support
  - Runtime implementation