Lecture 12
OpenCL II

4190.414A
Multicore Computing
Fall 2017
© Jaejin Lee

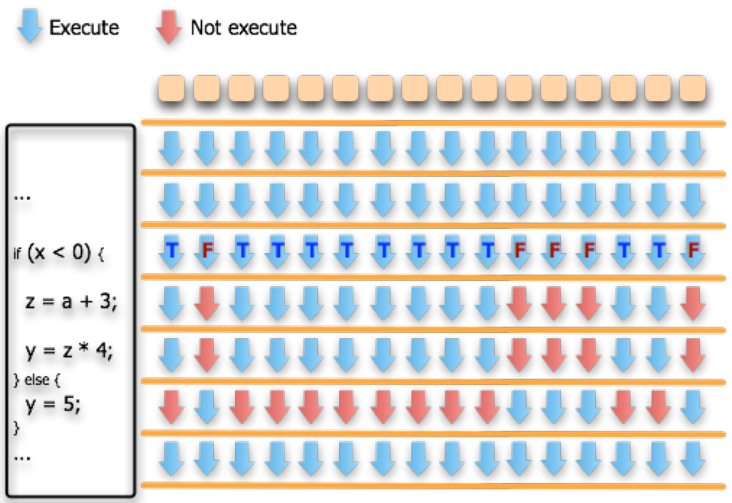
Optimization Issues in GPU Programming

Hardware Scheduling Unit in GPUs

- Basic unit of GPUs for scheduling
  - All threads in it processes a single instruction at the same time in SIMD fashion
  - Lock-step
- NVIDIA - warp
  - 32 hardware threads (work-items)
- AMD - wavefront
  - 64 hardware threads (work-items)
- Other vendors may have different names

Divergence

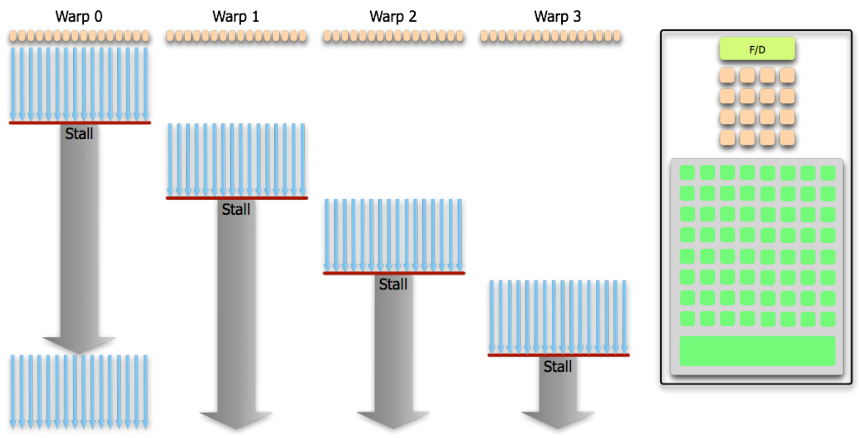
- When work-items in the same work-group follow different paths of control flow, they diverge in their execution
  - If - then - else
  - Loops with different loop bounds for different work-items
- Low degree of divergence will be better
- Pick a work-group size that is a multiple of the warp size



```
...
if (x < 0) {
  z = a + 3;
  y = z * 4;
} else {
  y = 5;
}
...
```

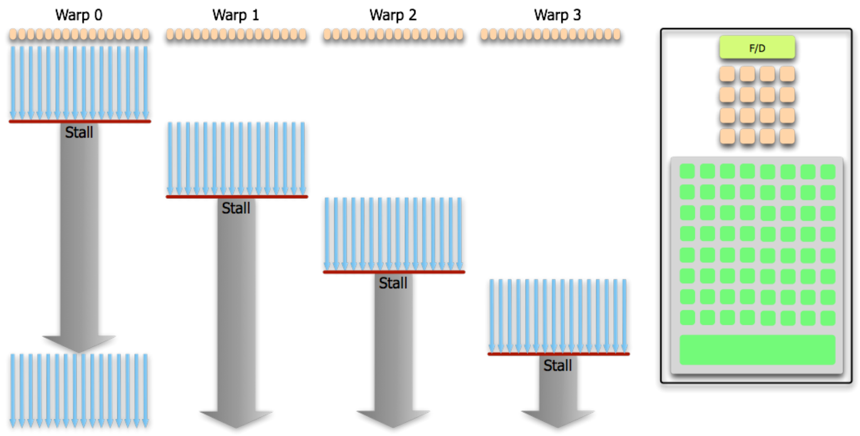
Occupancy

- The number of active warps per Streaming Multiprocessor (AMD calls it a Streaming Core)
  - Computed at compile time
- It describes how well the resources of the SM are being utilized



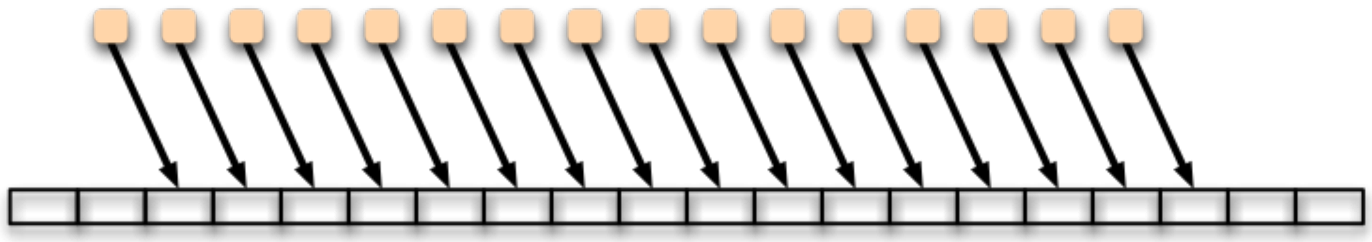
Occupancy (contd.)

- The maximum number of registers required by a kernel must be available for all threads in a warp
- The maximum size of local memory required by a kernel must be available for all threads in a warp
- The maximum number of active threads and warps per SM is limited
- Consider the above three factors



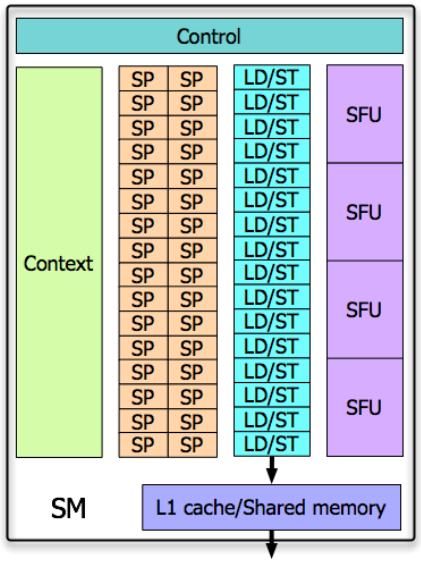
Memory Coalescing

- The same instruction for all work-items in a warp accesses consecutive global memory locations
  - The hardware coalesces all of these accesses to a consolidated access
  - To achieve the peak global memory bandwidth
- For example, work-item 0 accesses global memory location N, work-item 1 accesses N+1, etc.



Instruction Mix

- Each SM has limited instruction processing bandwidth
  - Due to limited number of functional units
- Loop unrolling will help
  - But increases register pressure



Thread Granularity

- Put more work into each work-item and use fewer work-items
  - May reduce the kernel launching overhead
  - May remove redundant computations between work-items
  - May increase the number of registers resulting in low occupancy
  - May reduce the number of work-groups resulting in making the SM underutilized

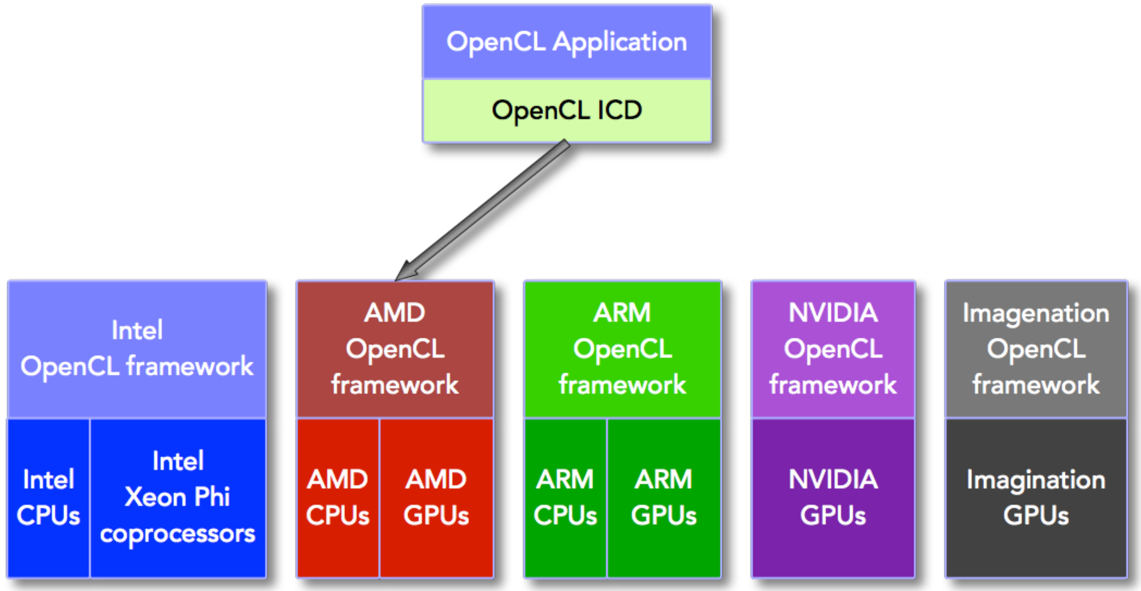
SnuCL

Limitations of OpenCL

- Current OpenCL implementations are targeting parallelism for multiple compute devices under a single OS instance
  - An application for a heterogeneous CPU/GPU cluster
    - MPI + OpenCL or MPI + CUDA
    - Complicated, less portable, and hard to maintain

OpenCL ICD (revisited)

- The OpenCL ICD enables multiple OpenCL implementations to coexist under the same system (an operating system instance)
- The user explicitly specifies which framework to use



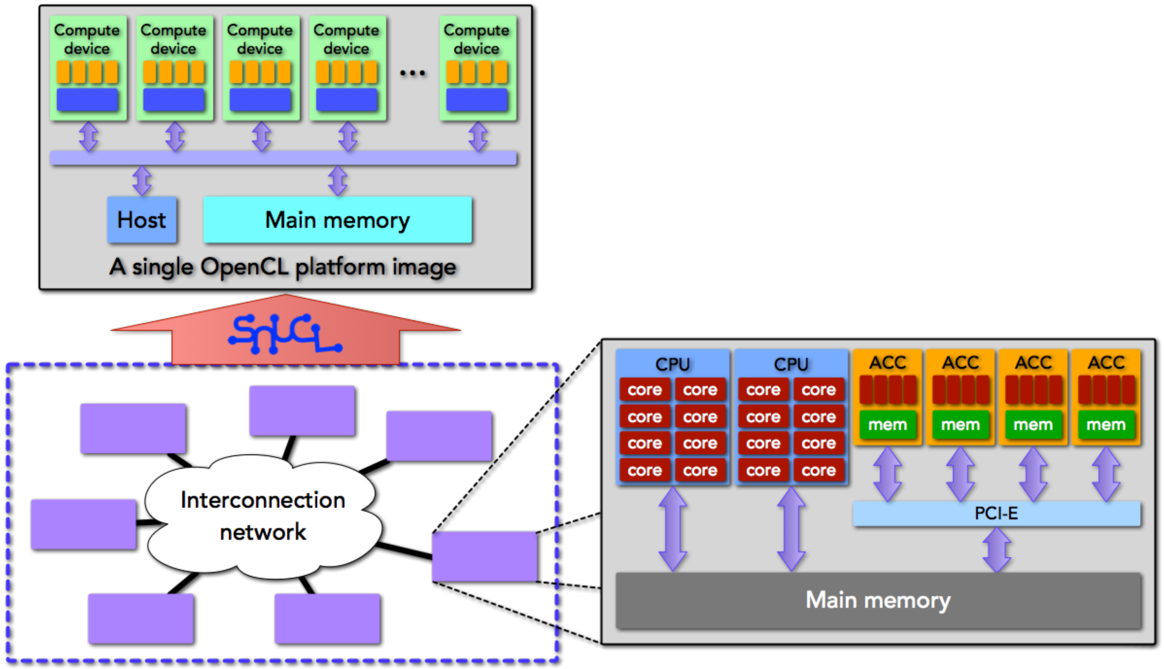
Limitations of the OpenCL ICD

- Users need to explicitly specify which framework is used in their applications

- Cannot share objects (buffers, events, etc.) across different frameworks in the same application

Illusion of a Single OpenCL Platform Image

- If the programmer can write applications for heterogeneous clusters using only OpenCL
  - Easy to program
  - More portable program

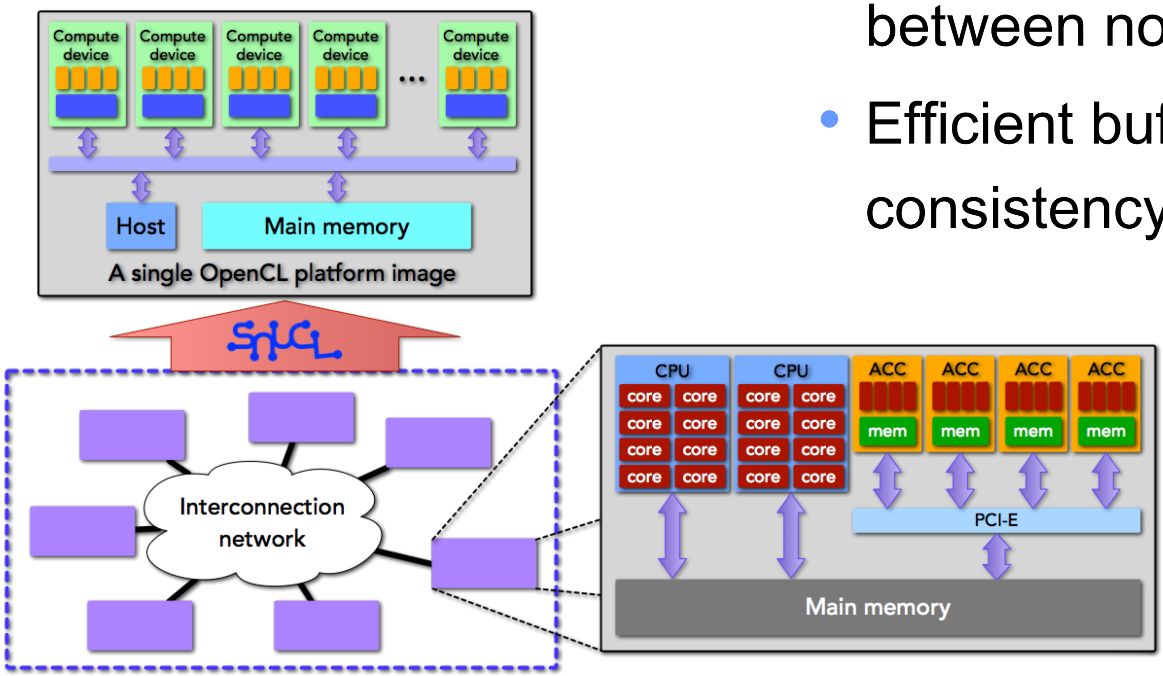


SnuCL

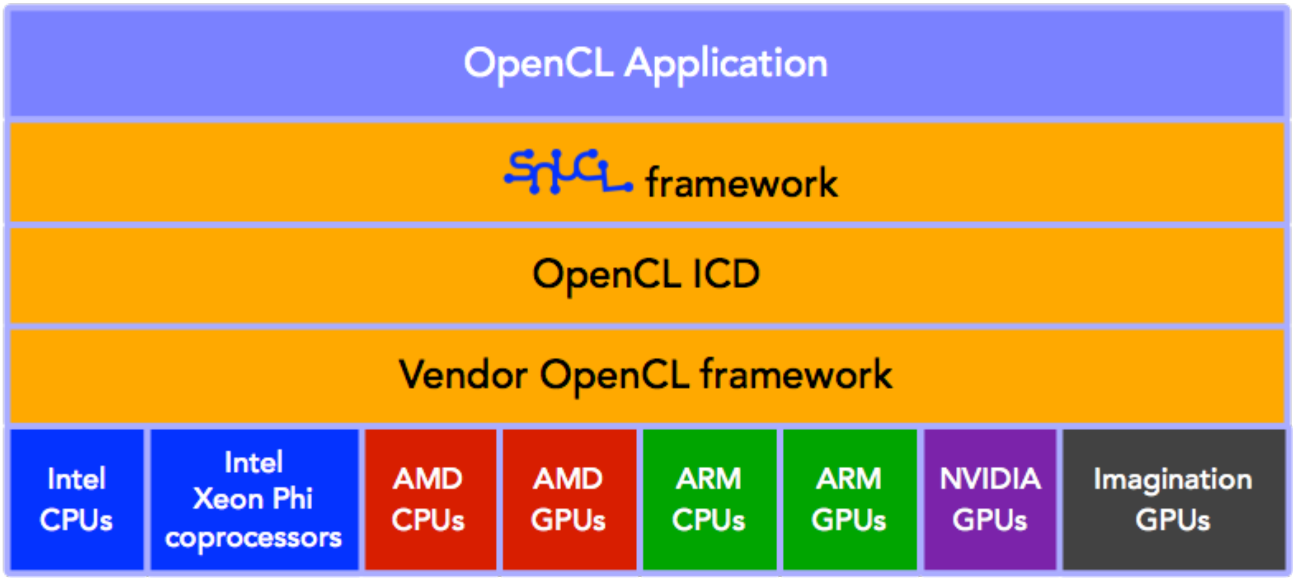
- An OpenCL framework
  - Platform layer + runtime + kernel compiler
- Freely available, open-source software developed at Seoul National University
  - http://aces.snu.ac.kr
  - Supports OpenCL 1.2
  - Passed most of OpenCL conformance tests
- Supports x86 CPUs, ARM CPUs, AMD GPUs, NVIDIA GPUs, Intel Xeon Phi coprocessors (from July, 2013)
- With SnuCL, an OpenCL application written for a single operating system instance runs on a heterogeneous cluster without any modification

How to Achieve the Illusion?

- SnuCL runtime provides the illusion
  - Handles communication between nodes
  - Efficient buffer and consistency management



Using ICD in SnuCL



SnuCL's Approach

- Exploits the OpenCL ICD
- However,
  - No need to explicitly specify a specific framework
  - Can share objects (buffers, events, etc.) between different frameworks in the same application
- Works for heterogeneous clusters, too

SnuCL's Approach (contd.)

- Naturally extends the original OpenCL semantics to the heterogeneous cluster environment
  - Provides an illusion of a heterogeneous system running a single OS instance

- With SnuCL, an OpenCL application written for a single OS instance runs on a heterogeneous cluster without any modification

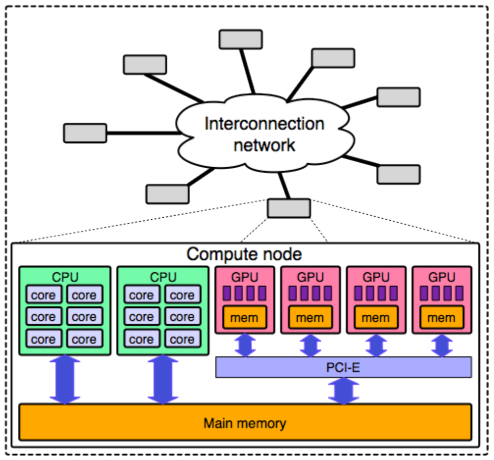
The Effect of Using SnuCL

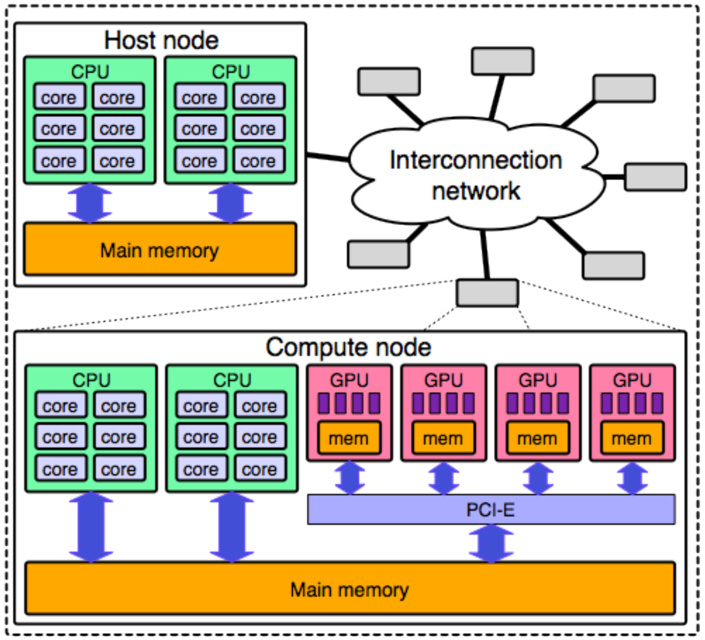
- Copy buffers between different nodes in the cluster environment (Buffer A → Buffer B)

Previous approach (Mixture of MPI and OpenCL)	SnuCL (OpenCL only)
MPI_Init(..);	
MPI_Comm_rank(MPI_COMM_WORLD,	
&rank);	…
…	cl_mem bufferA = clCreateBuffer(…);
cl_mem bufferA = clCreateBuffer(…);	cl_mem bufferB = clCreateBuffer(…);
cl_mem bufferB = clCreateBuffer(…);	…
…	clEnqueueCopyBuffer(cq, bufferA, bufferB, …);
void *temp = malloc(…);	…
if (rank == SRC_DEV) {	
clEnqueueReadBuffer(cq, bufferA, …, temp, …);	
MPI_Send(temp, …, DST_DEV, …);	
} else if (rank == DST_DEV) {	
MPI_Recv(temp, …, SRC_DEV, …);	
clEnqueueWriteBuffer(cq, bufferB, …, temp, …);	
}	
…	
MPI_Finalize();	

Target Cluster Architecture

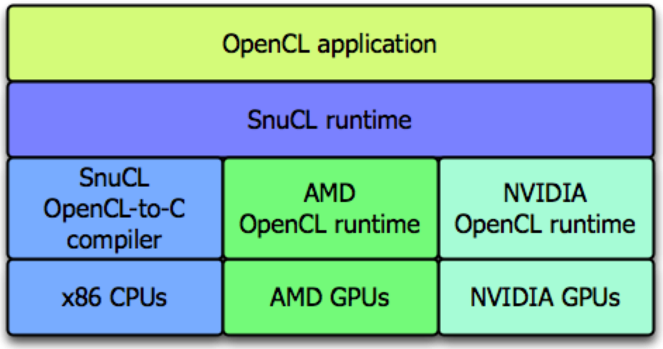
- A compute node is designated as the host node



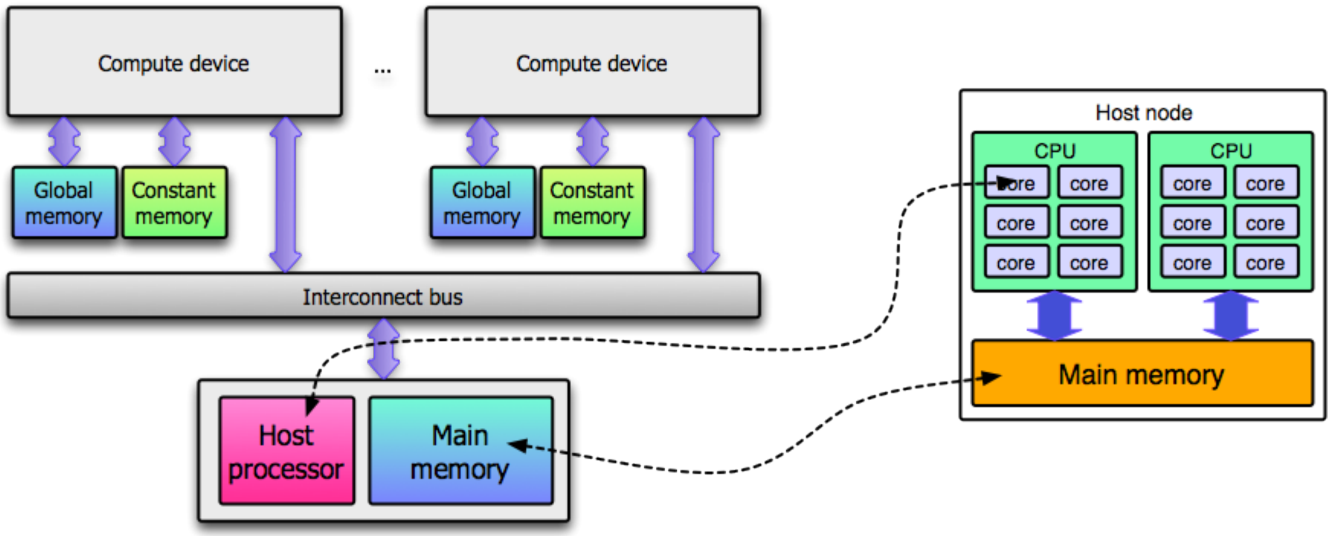


How to Achieve the Single System Image?

- SnuCL runtime provides the illusion
  - Mapping components between the OpenCL platform and underlying hardware resources
- Source-to-source kernel restructuring techniques
  - OpenCL C to C for CPUs
- Buffer management techniques
  - Efficient node to node data transfer
  - Consistency management



Mapping Components (the Host)



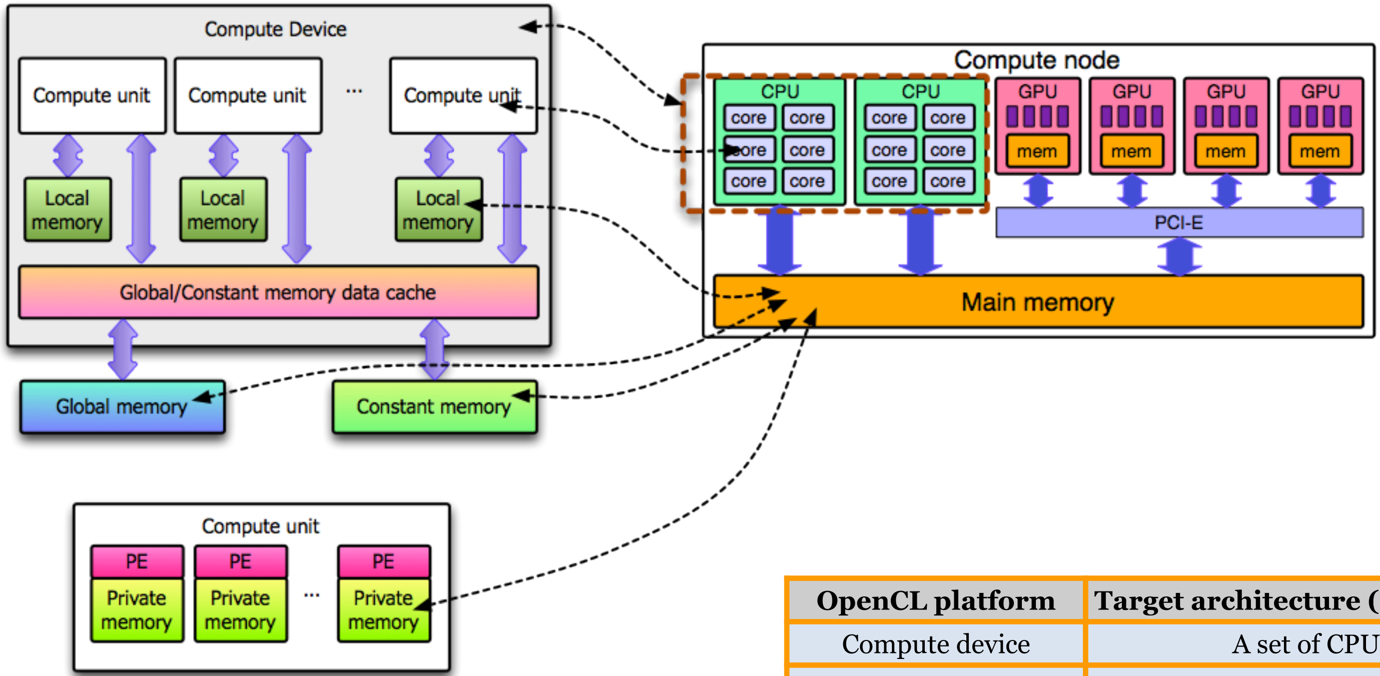
OpenCL platform	Target architecture (the host node)
Host processor	A CPU core
Main memory	Node main memory

Mapping Components (GPU Devices)

- Straightforward

OpenCL platform	Target architecture (compute node)
Compute device	GPU
Compute unit	Streaming multiprocessor
Processing element	Scalar processor
Global memory	Global memory
Constant memory	Constant memory
Local memory	Shared memory
Private Memory	Registers
Data cache	Data cache in the GPU

Mapping Components (CPU Devices)



OpenCL platform	Target architecture (compute node)
Compute device	A set of CPU cores
Compute unit	CPU core
Processing element	**Emulated by a CPU core**
Global memory	Node main memory
Constant memory	Node main memory
Local memory	Node main memory
Private Memory	Node main memory
Data cache	**Data caches and the coherence mechanism in CPUs**

Source-to-Source Translation Techniques

- For CPUs, implemented in clang (the LLVM front-end)
- For AMD GPUs, the SnuCL uses the AMD OpenCL runtime
- For NVIDIA GPUs, the SnuCL uses the NVIDIA OpenCL runtime

For CPUs

- Emulate PEs in a CU with a CPU core to execute a work-group
  - Using a triply nested loop to iterate over the index space
- But, barrier synchronization is problematic

```
__kernel void vec_add(__global float *A,
                      __global float *B,
                      __global float *C)
{
    int id = get_global_id(0);
    C[id] = A[id] + B[id];
}
```

C code

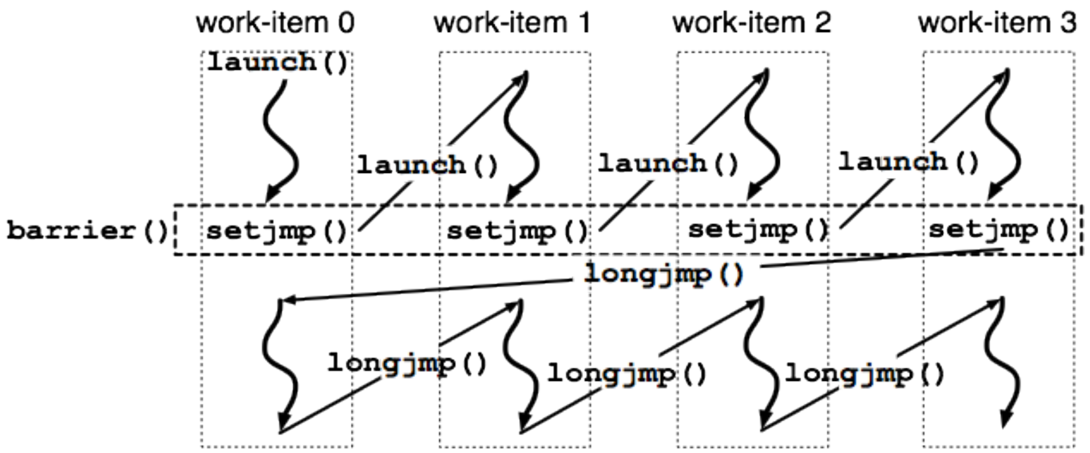
```
void vec_add(float *A, float *B, float *C)
{
  for (int __k = 0; __k < __local_size[2]; __k++) {
    for (int __j = 0; __j < __local_size[1]; __j++) {
      for (int __i = 0; __i < __local_size[0]; __i++) {
        int id = get_global_id(0);
        C[id] = A[id] + B[id];
      }
    }
  }
}
```

Handling Barrier Synchronization

- Two ways of emulating PEs [PACT '10]
  - Lightweight context switch between work-items
  - Work-item coalescing and variable expansion

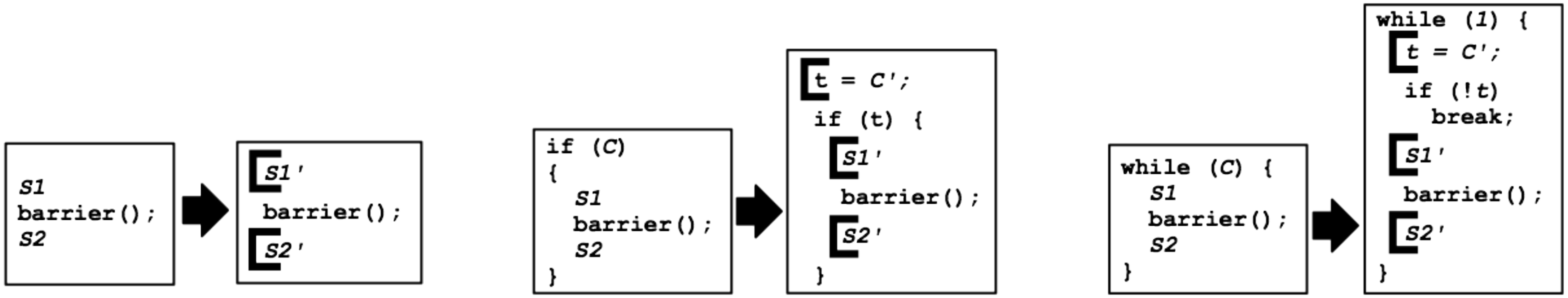
Context Switching between Work-items

- When a work-group is assigned to a CU (i.e., a CPU core), the associated thread executes each work-item in the work-group one by one
  - Context switching using setjmp and longjmp functions



Work-item Coalescing [PACT '10]

- To avoid context switch overhead
- The source-to-source translator encloses the kernel code with work-item coalescing loops (WCLs)
  - A WCL is the triply nested loop that iterates over the index space of a work-group
- A kernel code region that needs to be enclosed with a WCL is called a work-item coalescing region (WCR)



```
S1
barrier();
S2
```

```
[S1'
 barrier();
[S2'
```

```
if (C)
{
  S1
  barrier();
  S2
}
```

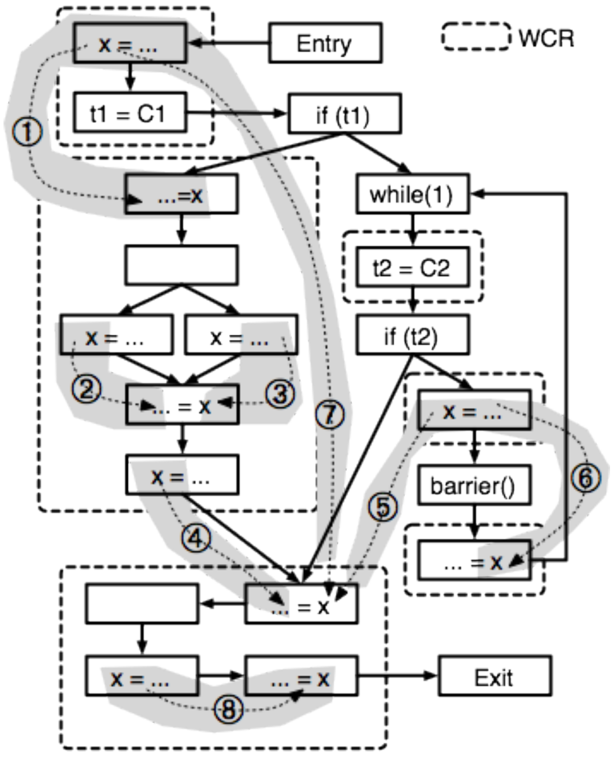
```
[t = C';
 if (t) {
  [S1'
   barrier();
  [S2'
 }
```

```
while (C) {
  S1
  barrier();
  S2
}
```

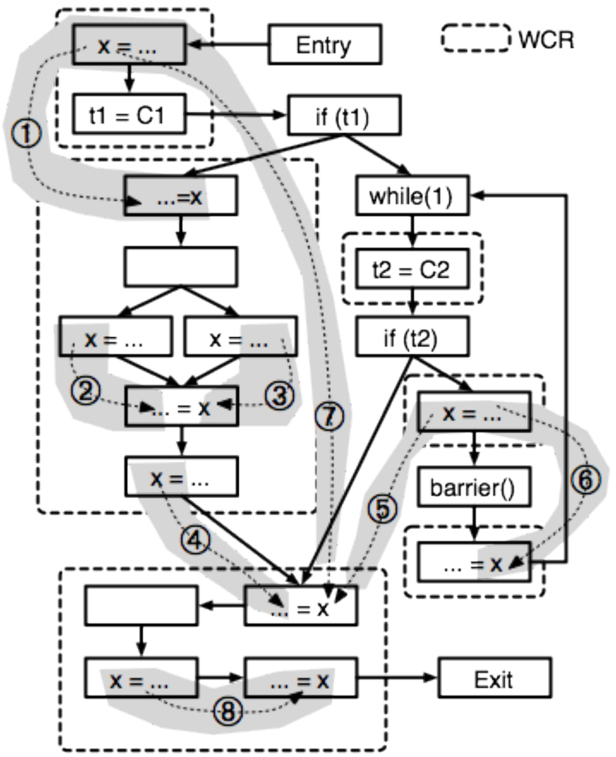
```
while (1) {
 [t = C';
  if (!t)
    break;
 [S1'
  barrier();
 [S2'
}
```

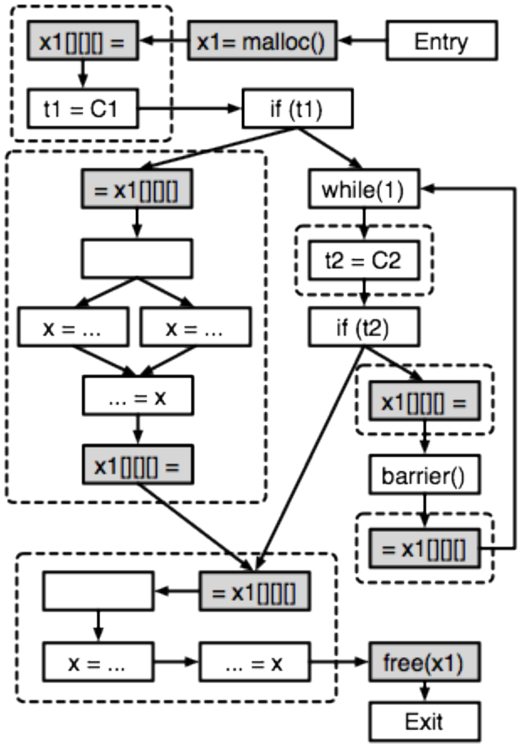
Web-based Variable Expansion [PACT '10]

- A work-item private variable that is defined in one WCR and used in another needs a separate location for a different work-item
    - To transfer the value from one to another
    - A variable is expanded to a three-dimensional array
- A web for a variable is all du-chains of the variable that contain a common use of the variable

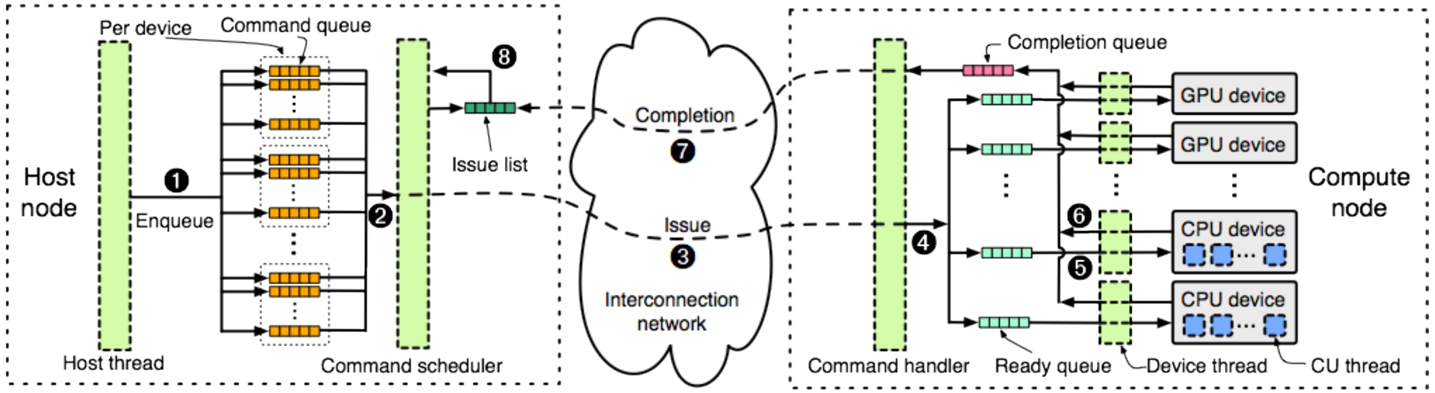


Web-based Variable Expansion (contd.)



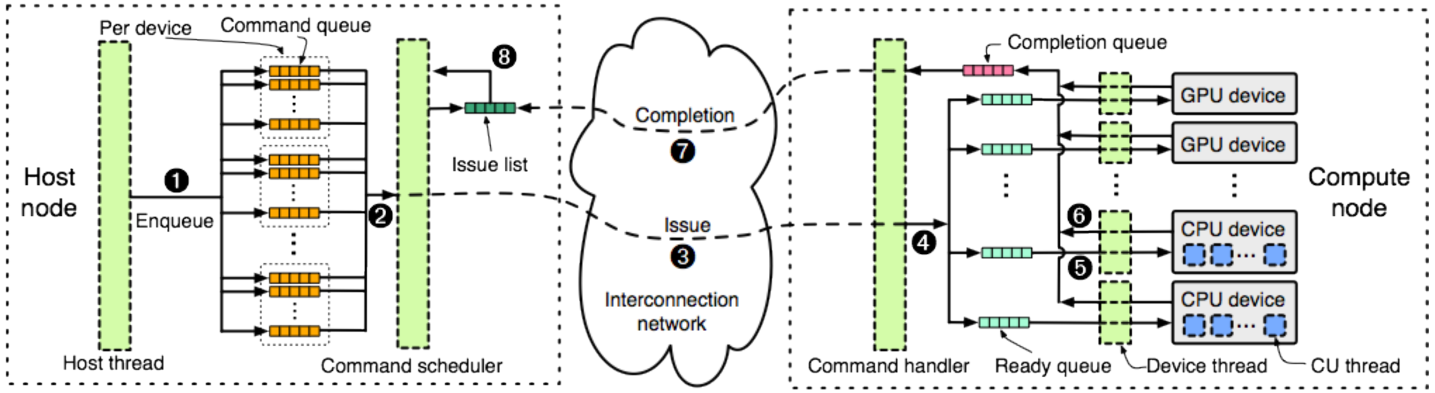


SnuCL Runtime



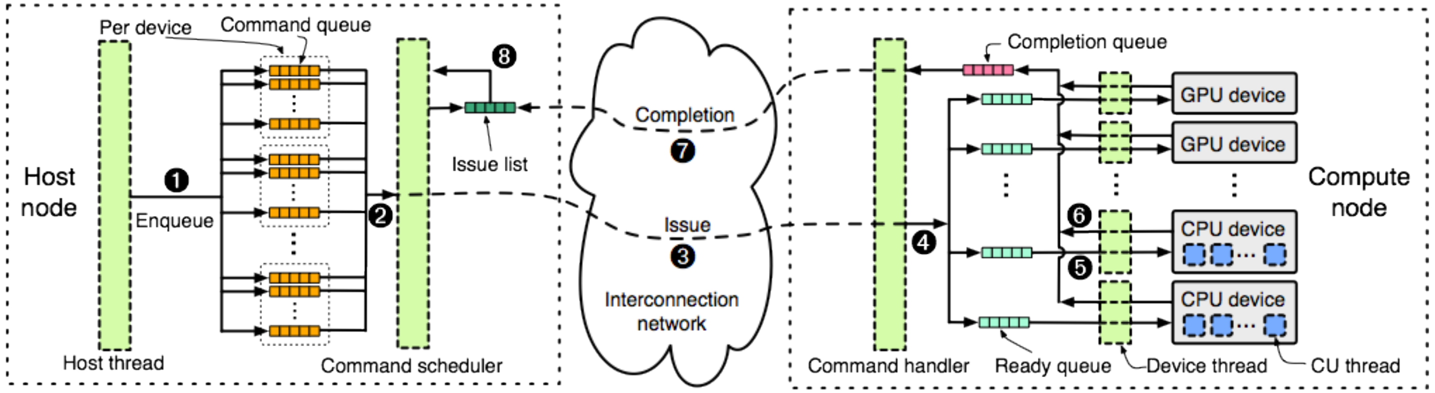
- Host node
  - Host thread
  - Command scheduler thread
- Compute node
  - Command handler thread
  - Device threads
  - CU threads (CPU only)

(1) Enqueuing a Command



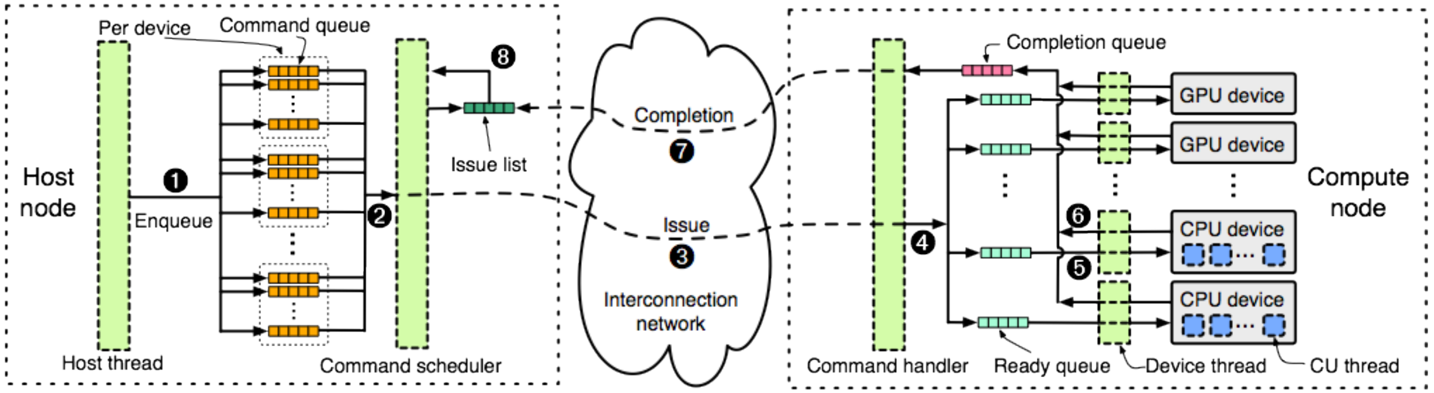
- The user executes an OpenCL application in the host node
    - The user only communicate with the host node
- The host thread in the host node executes the host program in the application
    - The host thread enqueues a command in a command-queue

(2) Scheduling the Command



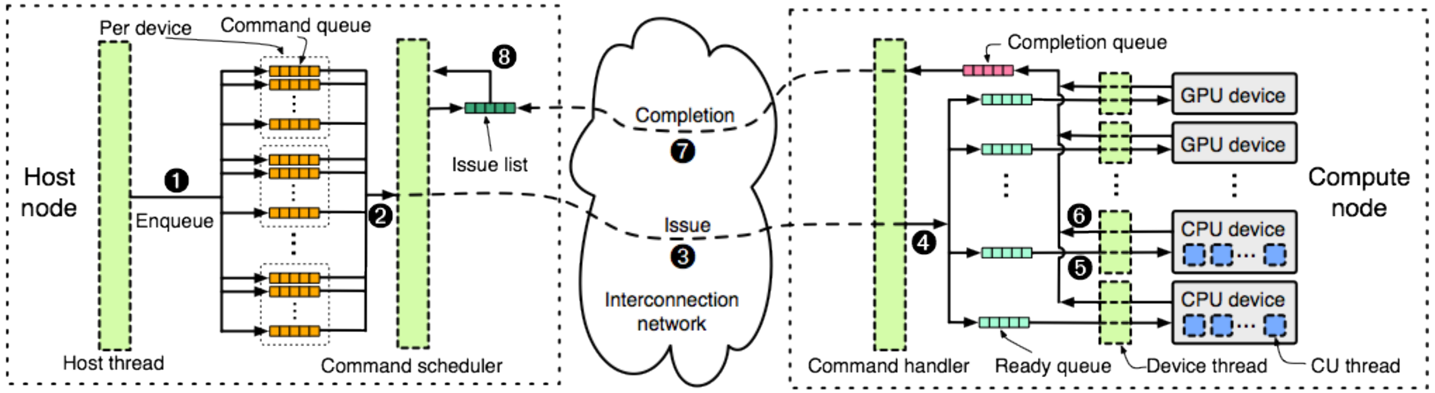
- The command scheduler in the host node schedules enqueued commands across compute nodes in the cluster
  - Honor the type (in-order or out-of-order) of each command-queue and synchronization enforced by the host program

(3) Issuing the Command



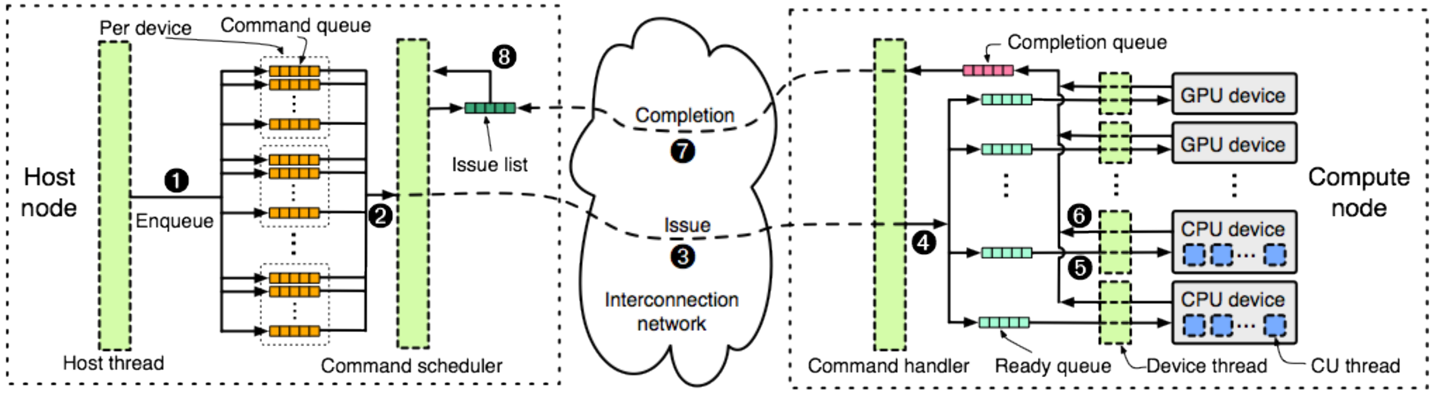
- The command scheduler issues the command
  - By sending a command message to the target compute node
- Issue list
  - Contain event objects associated with issued but not completed commands

(4) Enqueuing the Command in the Ready Queue



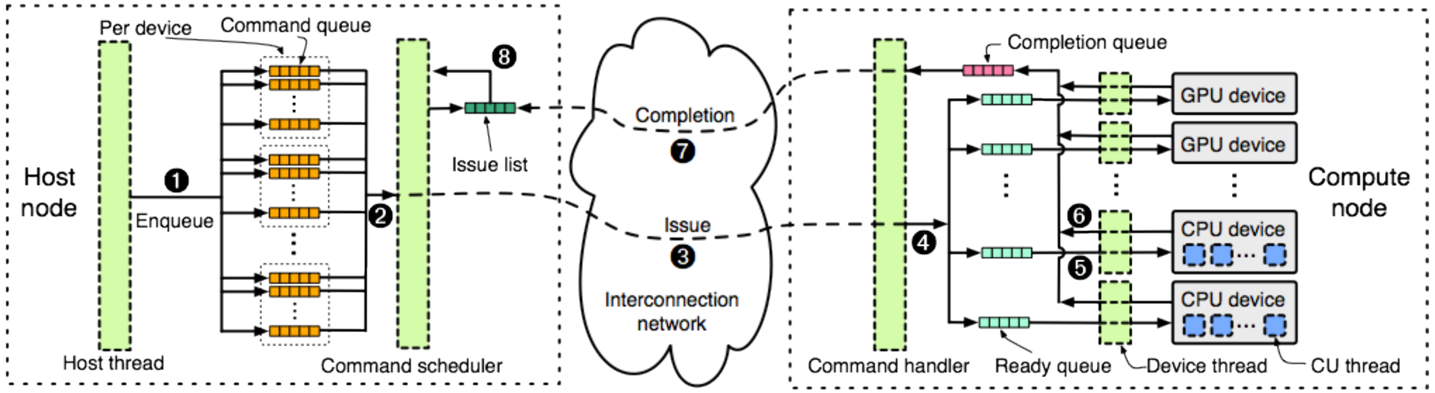
- The command handler in a compute node receives the message from the host node
  - Enqueue the command into a ready queue for the target compute device
- Ready queue
  - One ready queue for a compute device
  - Contain commands that are issued but not launched

(5) Launching the Command



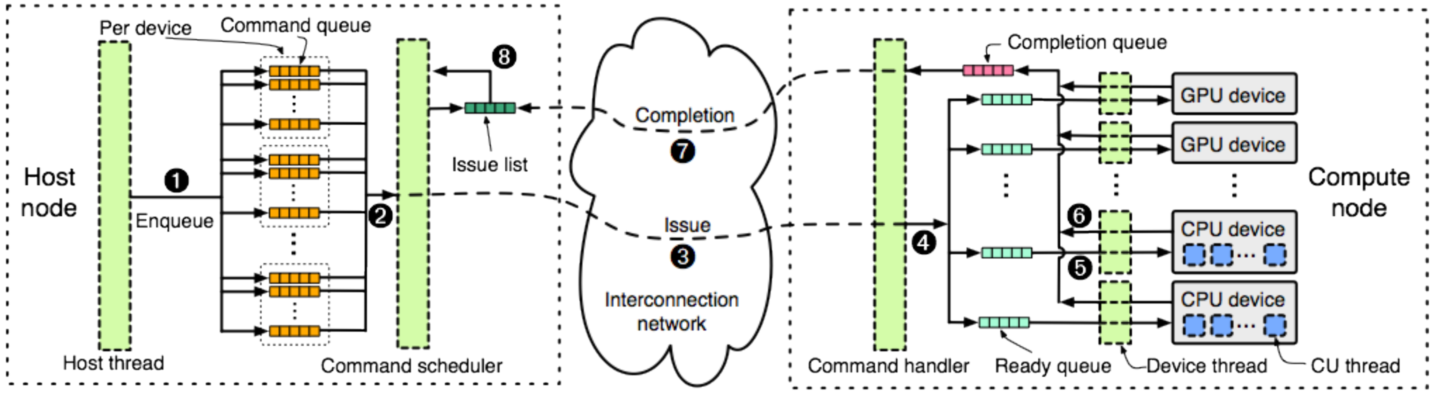
- The device thread dequeues a command from its ready queue and launches the command to the associated compute device

(6) Enqueueing the Completed Command



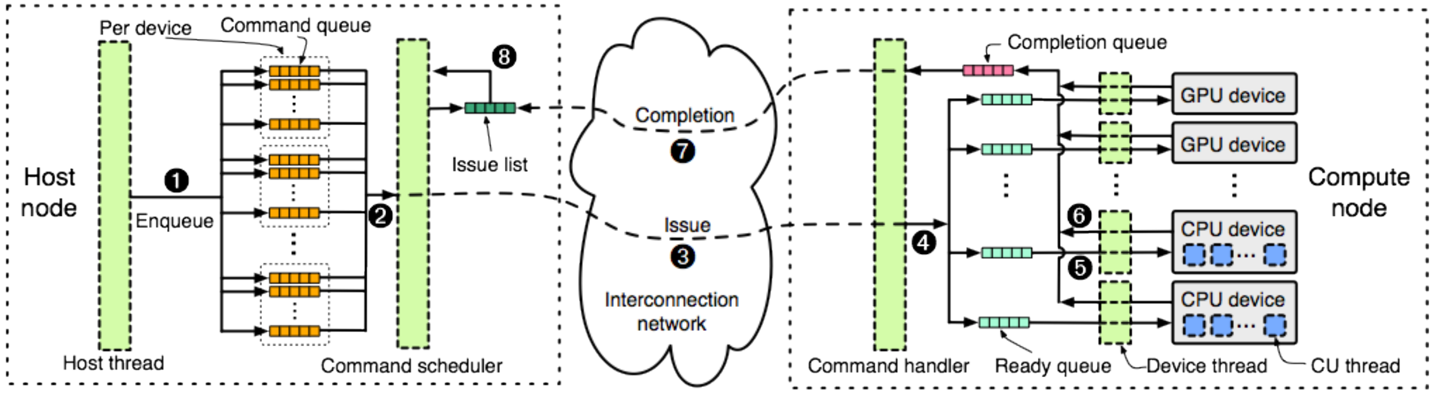
- When the compute device completes the command, the device thread insert the associated event to the completion queue
- Completion queue
  - Contains completed event objects in a compute node

(7) Notifying Completion



- The command handler in the compute node dequeues the event from the completion queue, and sends a completion message to the host node

(8) Completing the Execution of the Command



- The command scheduler in the host node receives the completion message from the compute node
    - Checks the issue list for completion
    - Resolves dependences between commands and schedules them

Dynamic Scheduling for CPU Devices

- From Li et al. [ICPP 1993]
- $S = \lceil N/(2P) \rceil$
  - S: the number of work-groups to be assigned to an idle CU thread in the CPU device
  - N: the number of remaining unscheduled work-groups
  - P: the number of all CU threads in the CPU device
- When the number of total work-groups is 64 and there are 4 CU threads in a CPU device
  - S: 8 7 7 6 5 4 4 3 3 3 2 2 2 1 1 1 1 1 1 1 1

Buffer Space Allocation

- An OpenCL buffer object is not associated with a specific compute device
  - Different compute devices can share buffers
  - Implementation dependent
- When clCreateBuffer(),
  - The SnuCL runtime does not allocate any memory space to the buffer
- When clEnqueue...(),
  - The runtime knows the target device
  - The SnuCL runtime allocates a memory space to the buffer when the host program enqueues a command that manipulates the buffer object

```
bufferA = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeA,
                         NULL, NULL);
...
clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
                       global, local, 0, NULL, NULL);
```

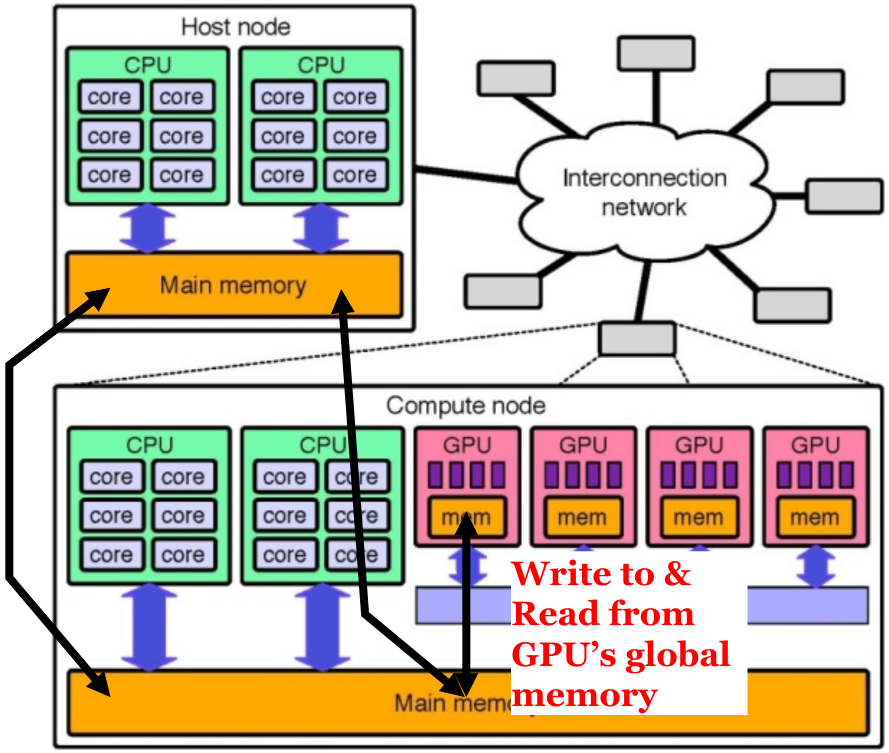
Latest Copy of a Buffer Object

- SnuCL runtime maintains a device list for each buffer
  - Compute devices that have the same latest copy of the buffer in their global memory
- When the target compute device does not have a latest copy of the buffer
  - Copies the buffer to the target device from the nearest compute device in the device list

Distance	Compute devices
0	Within a device
1	CPU device ⟺ CPU device in the same node
2	CPU device ⟺ GPU device in the same node
3	GPU device ⟺ GPU device in the same node
4	CPU device ⟺ CPU device in different nodes
5	CPU device ⟺ GPU device in different nodes
6	GPU device ⟺ GPU device in different nodes

Buffer Reads and Writes

Write to &
Read from
CPU's global
memory



Write to &
Read from
GPU's global
memory

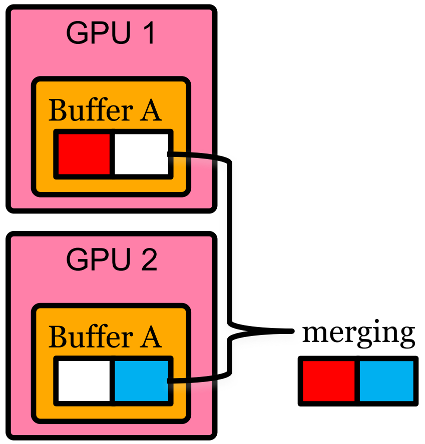
Buffer Copy Commands

```
clEnqueueCopyBuffer(command_queue, src_buffer, dst_buffer, …)
```

- Target device
  - The compute device that is associated with the command_queue
- Source device
  - The nearest compute device to the target device (among the device list in the src_buffer)
- Three cases
  - The source and target devices are the same
  - The source and target devices are different but they are in the same node
  - The source and target devices are in different nodes

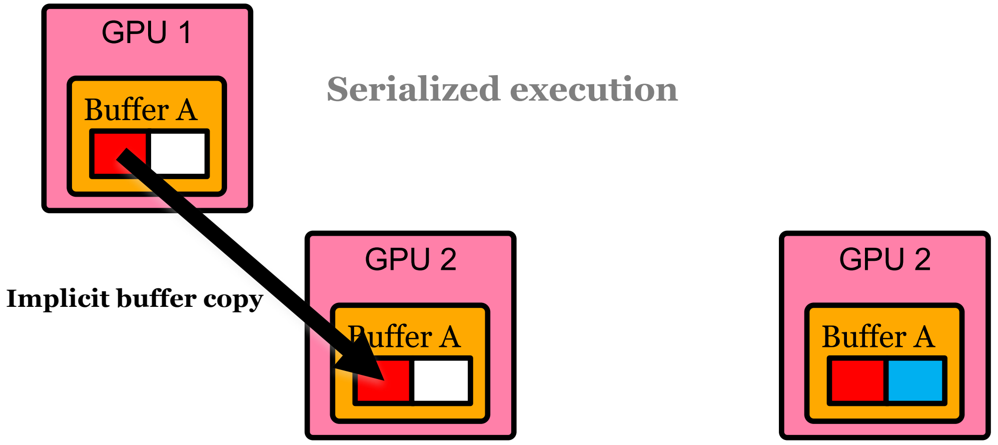
Consistency Management

- A buffer object can be shared between different compute devices (e.g., GPU1 and GPU2)
- Two different compute devices update the same buffer simultaneously
- A simple solution
  - Compare (diff) and merge
  - Need to maintain an original copy of the buffer
  - High overhead



Consistency Management (contd.)

- SnuCL serializes executions of the commands
  - To avoid the comparison and merging overhead


GPU 1
Buffer A
Serialized execution
Implicit buffer copy
GPU 2
Buffer A
GPU 2
Buffer A