OpenCL for Heterogeneous Clusters

Jaejin Lee

Center for Manycore Programming

School of Computer Science and Engineering

Seoul National University

jlee@cse.snu.ac.kr

http://aces.snu.ac.kr

http://www.isc13.org/tutorial_materials

Username: tutorials

Password: 13tut836

Outline

- Part I: Introduction to OpenCL
- Part II: Introduction to SnuCL

Part I

Introduction to OpenCL

Part I: Introduction to OpenCL

- Heterogeneous computing and OpenCL
- Background for parallel processing
- GPU architectures
- Intel Xeon Phi coprocessor
- Introduction to the OpenCL framework
- OpenCL for a single compute device
- OpenCL for multiple compute devices
- OpenCL+MPI for clusters
- HPL in OpenMP+MPI
- Limitations of the OpenCL programing model

Heterogeneous Computing Systems

- Computer systems that use more than one type of computational units for computing
  - Contain different types of processors
    - CPUs, DSPs, GPUs, FPGAs, and ASICs
  - For extra performance and power efficiency

- Heterogeneity in ISAs, processing power, power consumption, memory hierarchies, micro-architectures, etc.

Heterogeneous Computing Systems (contd.)

- General-purpose processors (resource management) + accelerators (compute intensive)
- Accelerators
  - Computer hardware that performs some task faster than the general-purpose CPU
  - AMD GPUs, Intel Xeon Phi coprocessors, NVidia GPUs, etc.

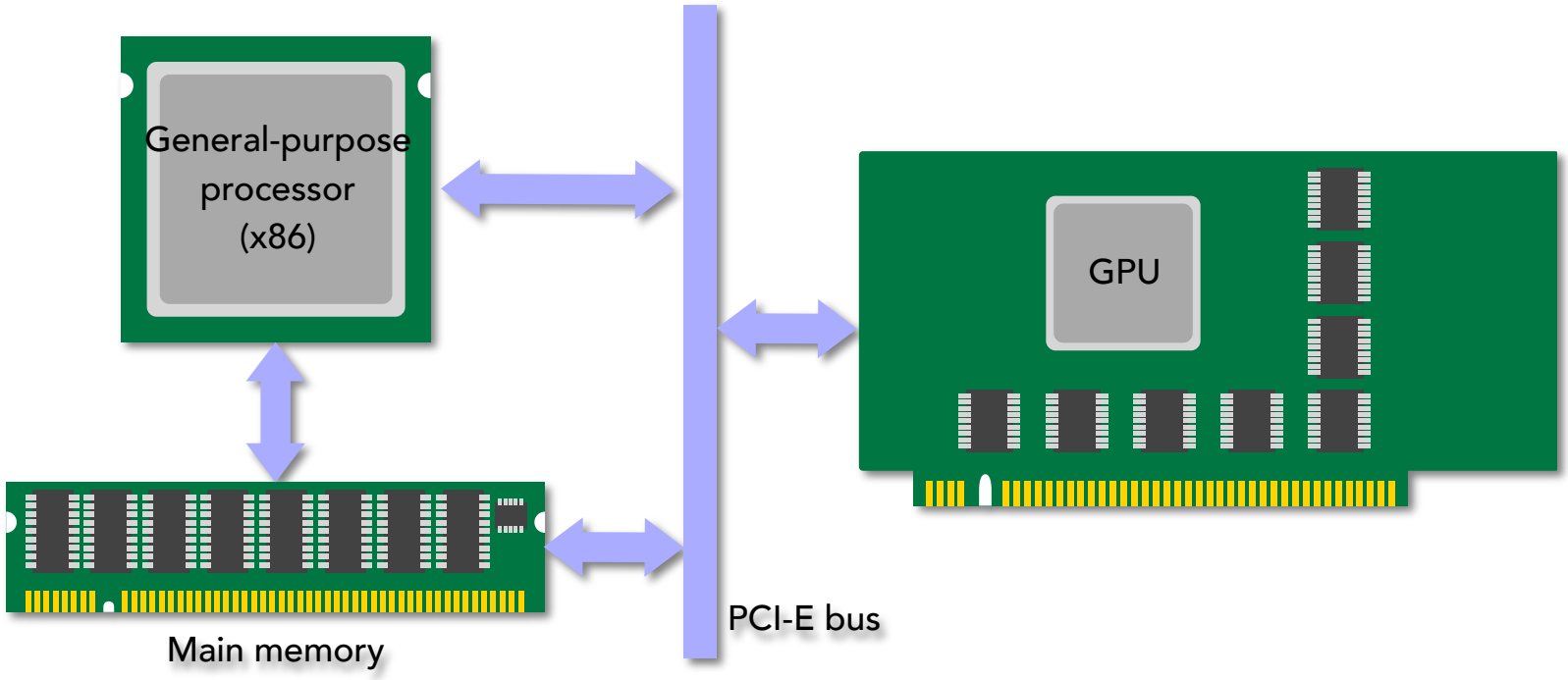
from www.nvidia.com

from www.intel.com

from www.amd.com

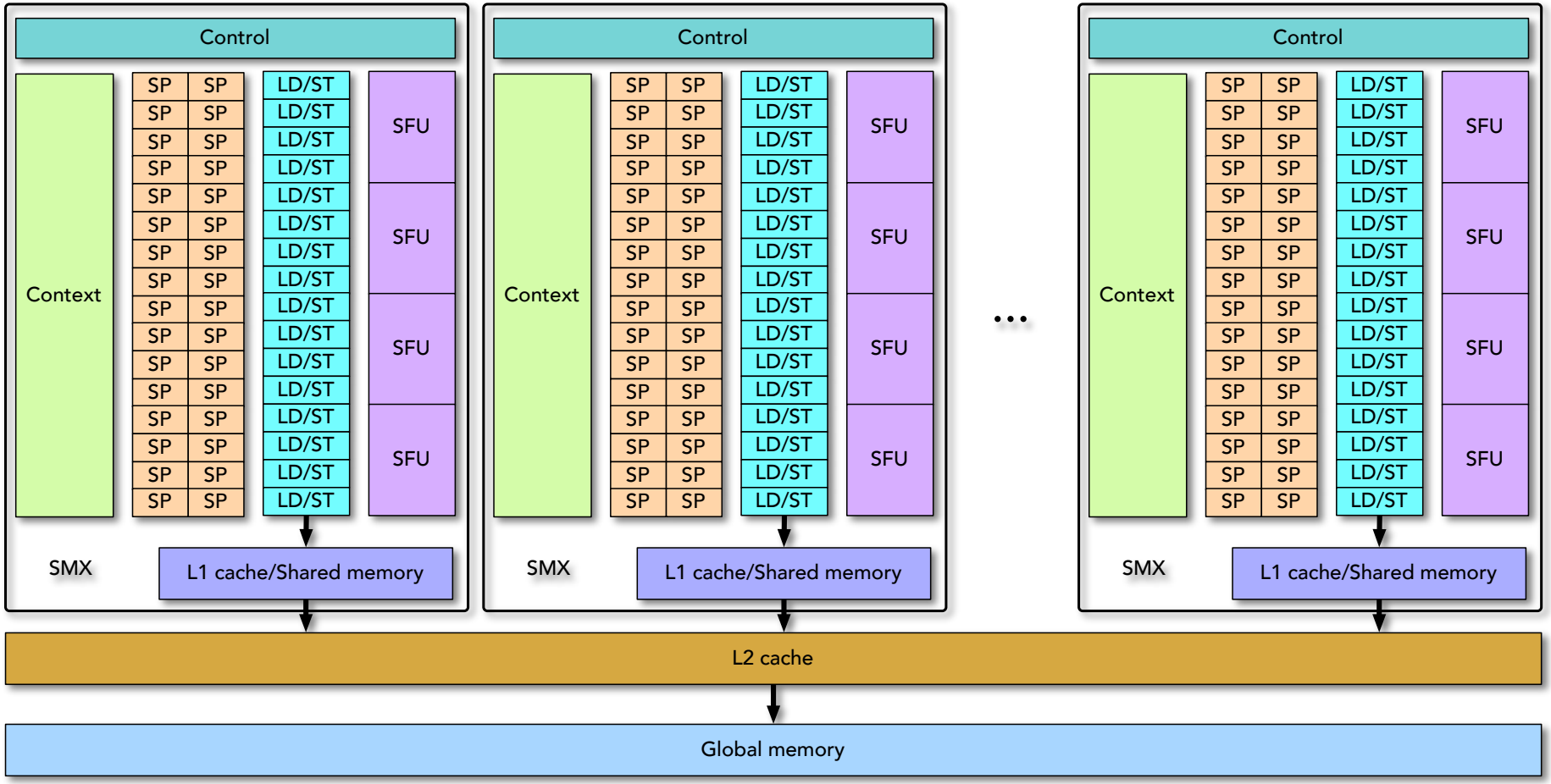
GPGPUs

- General-Purpose computing on Graphics Processing Units



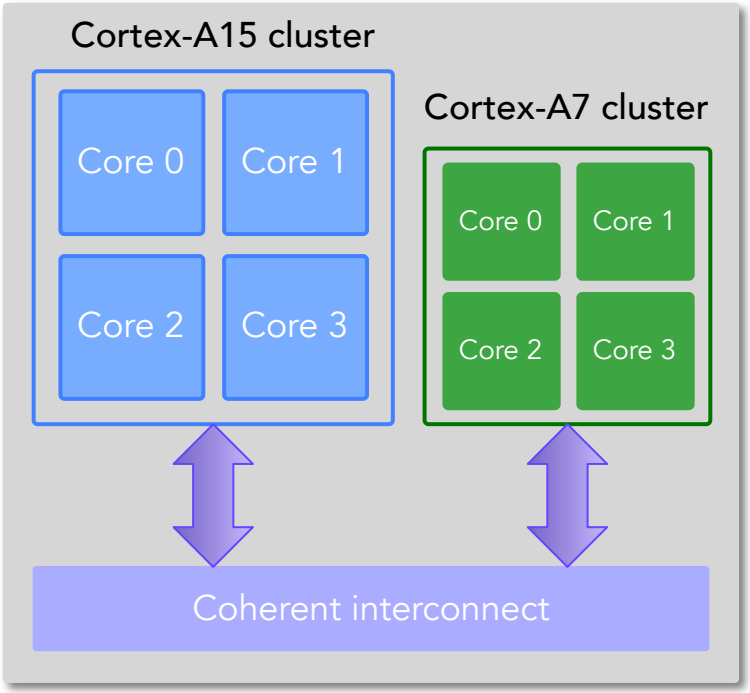
Typical GPU Architecture

- Thousands of simple cores (scalar processors) in a single GPU
  - Massive parallelism



ARM big.LITTLE

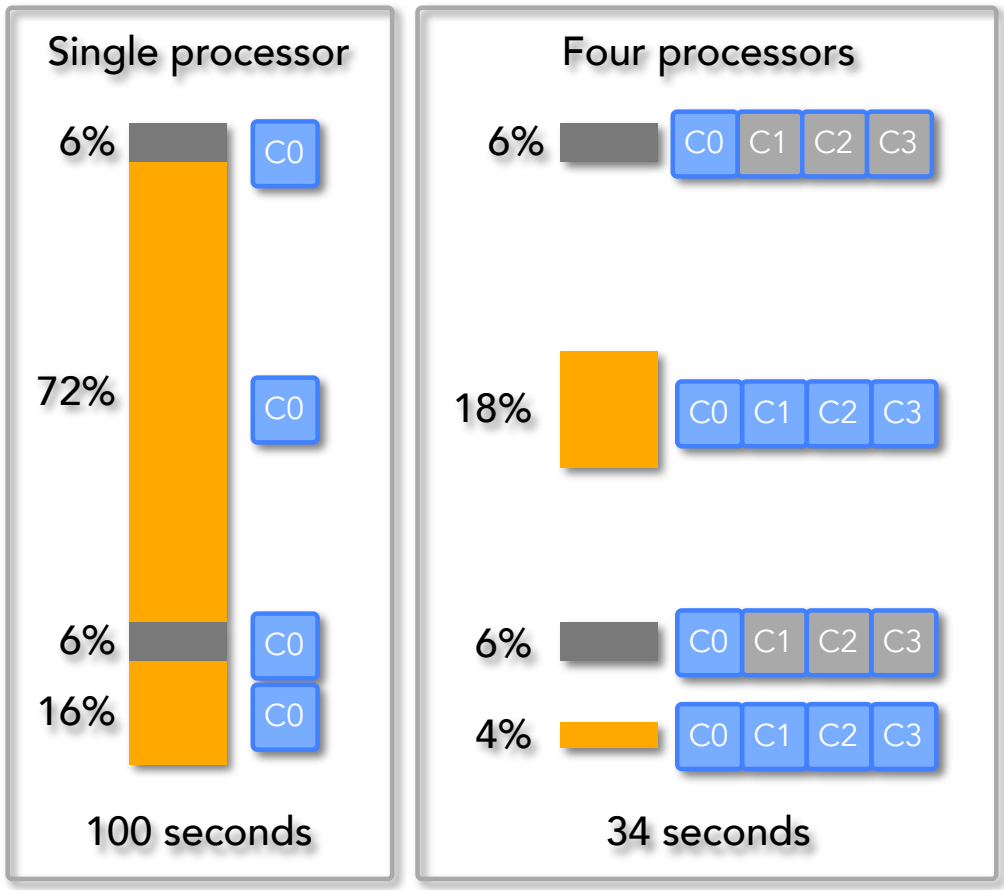
- A heterogeneous computing architecture proposed by ARM
- Combining relatively slower, low-power processor cores (e.g., Cortex-A7) with relatively more powerful and power-hungry cores (e.g., Coretex-A15) in a single chip



Amdahl's Law

- p: the proportion of a program that can be parallelized
- 1 - p: the proportion of a program that cannot be parallelized
- n: the number of processors

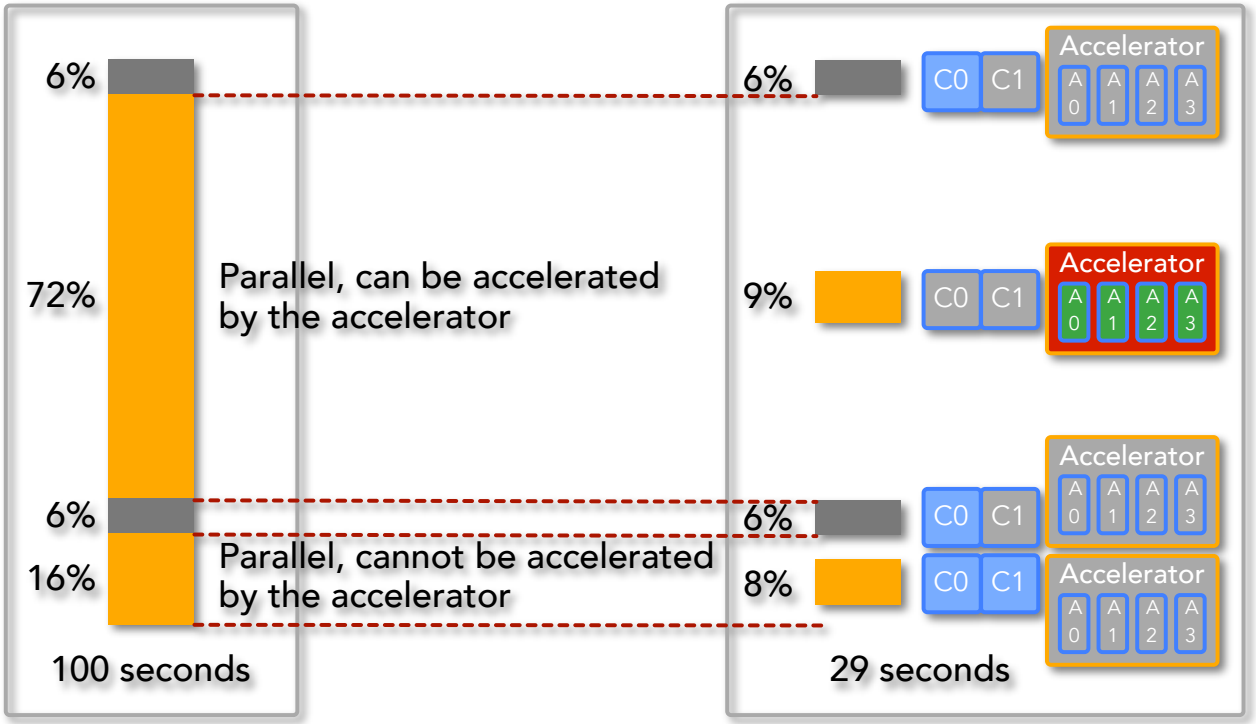
$$\text{speedup} = \frac{1}{(1-p) + \frac{p}{n}}$$



$$\begin{aligned}\text{speedup} &= \frac{1}{(1-0.88)+0.88/4} \\ &= 1/0.34 \\ &= 2.94\end{aligned}$$

Why Heterogeneous Systems?

- The accelerator costs the same as two general-purpose cores
- Each accelerator core executes the code fragment twice as fast as the general-purpose core



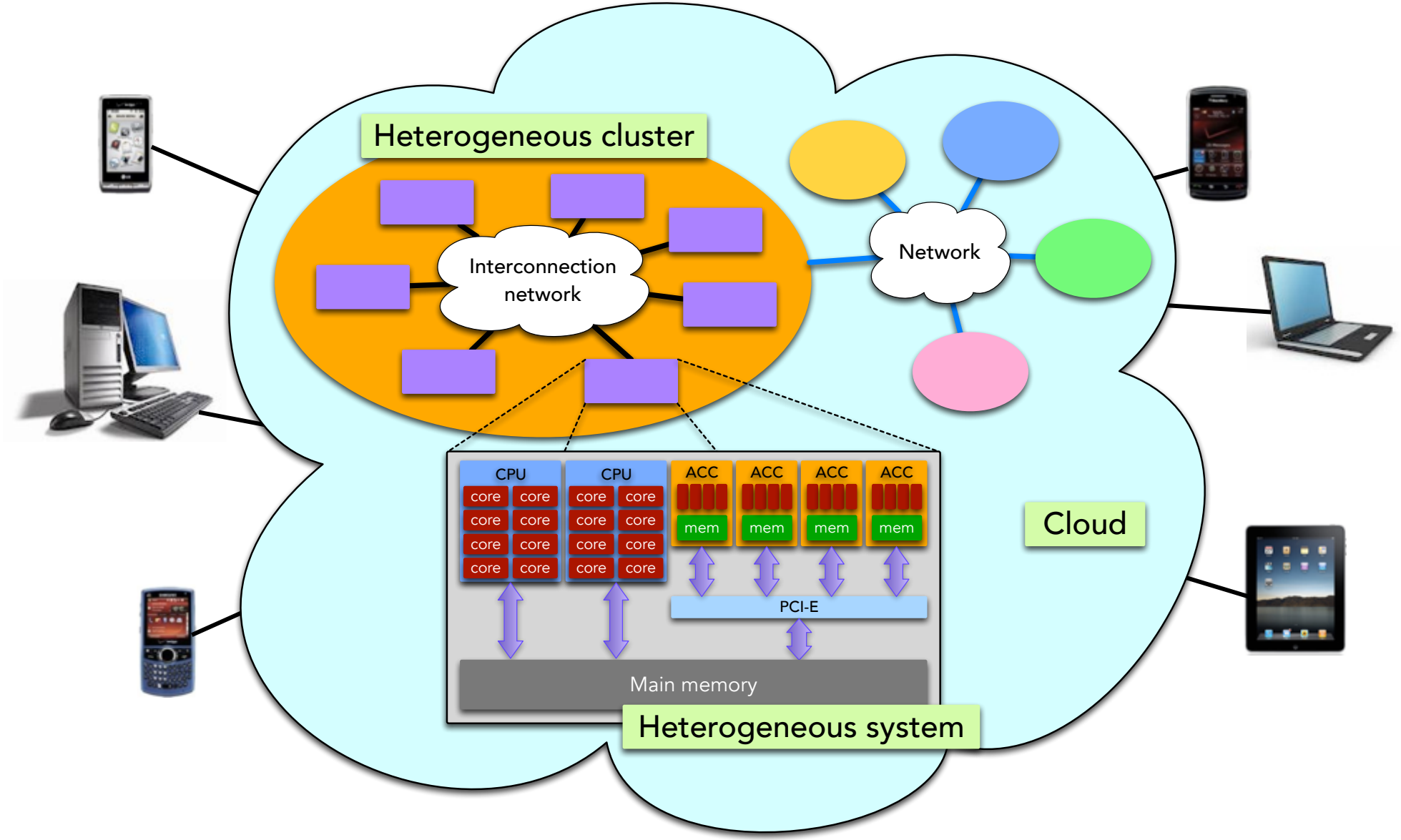
$$\text{speedup} = \frac{1}{(1-0.88) + \frac{(0.72/4)}{2} + \frac{0.16}{2}} = \frac{1}{0.29} = 3.45$$

Trend in Top500

- The number of heterogeneous supercomputers is continuously increasing in Top500

Top 500	Jun 2009	Nov 2009	Jun 2010	Nov 2010	Jun 2011	Nov 2011	Jun 2012	Nov 2012
Homogeneous system	495	493	491	483	481	461	442	438
Heterogeneous system	5	7	9	16	19	39	58	62 (12.4%)

Heterogeneous Parallel Computing



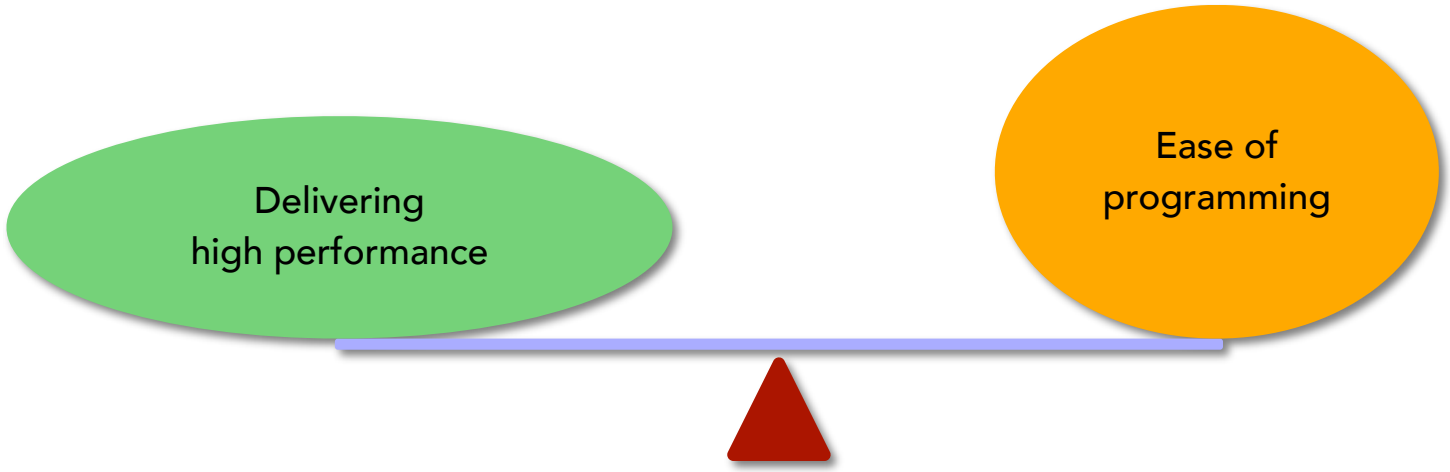
Ease of Programming

- How to handle the heterogeneity?
- Programmers need to make the best use of all the available heterogeneous devices ***with a single programming model***

Parallel Programming Models

- An interface between the programmer and the parallel machine when developing an application
  - Languages, libraries, language extensions, compiler directives, etc.

- Important to have balance between ***delivering high performance*** and ***ease of programming***



OpenCL

- ***Open*** ***C***omputing ***L***anguage

- Open, royalty-free standard
  - OpenCL 1.0 was released in late 2008, now OpenCL 1.2

- A framework (parallel programming model) for heterogeneous parallel computing
  - A language (based on C99), API, libraries, and a runtime system

- Supported by many vendors, such as Apple, AMD, ARM, IBM, Intel, Imagination, NVIDIA, Qualcomm, Samsung, etc.

OpenCL (contd.)

- From mobile devices to supercomputers
- Portable code across different architectures
  - CPUs, GPUs, Cell BE processors, Xeon Phi coprocessors, FPGAs, etc.
- Not yet portable performance across different architectures
  - Good research topic

Part I: Introduction to OpenCL

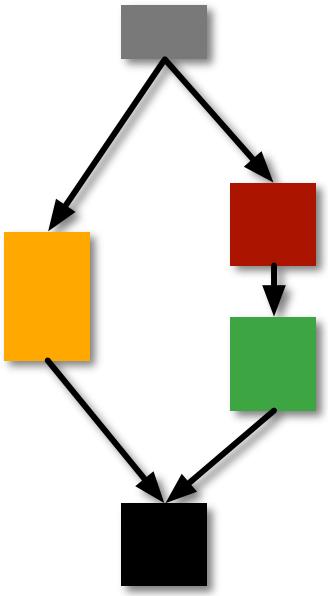
- Heterogeneous computing and OpenCL
- Background for parallel processing
- GPU architectures
- Intel Xeon Phi coprocessor
- Introduction to the OpenCL framework
- OpenCL for a single compute device
- OpenCL for multiple compute devices
- OpenCL+MPI for clusters
- HPL in OpenMP+MPI
- Limitations of the OpenCL programing model

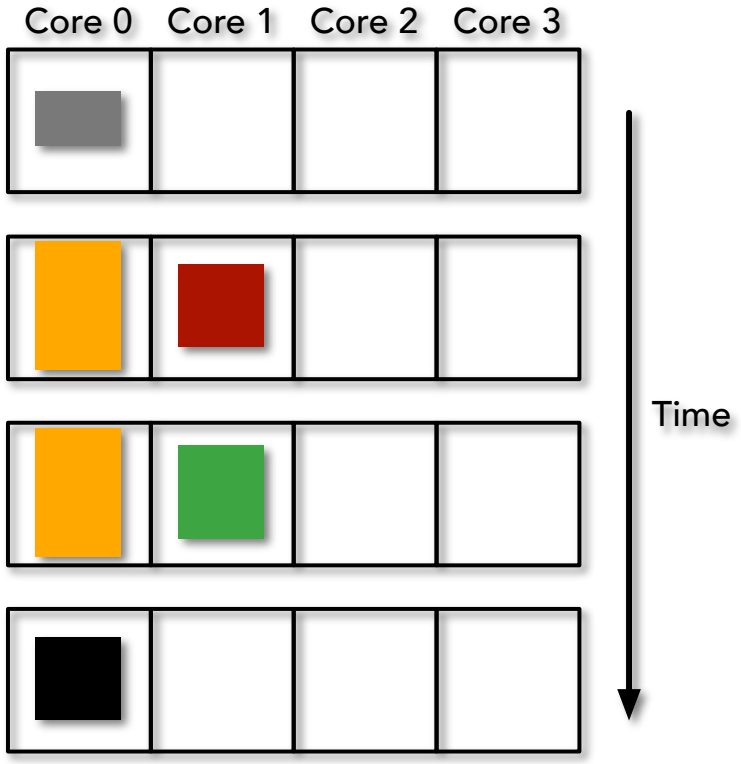
Task Parallelism

- Four chefs prepare 20 meals
- Four different tasks
  - Appetizer, salad, main dish, and dessert
- Each chef focuses on one of the four tasks simultaneously

Task Parallelism (contd.)

- Performing distinct tasks at the same time
- Dividing an application into different parallel tasks (functions)
  - Most applications have only a few parallel tasks





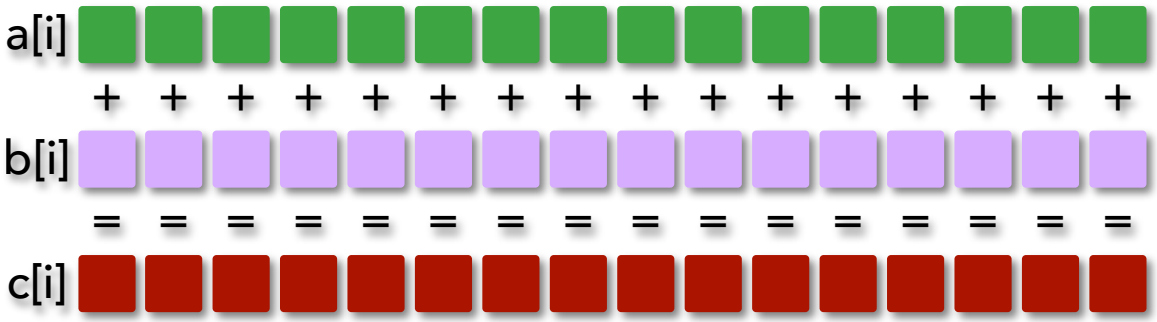
Data Parallelism

- Each of the four chefs producing 20/4 complete meals
- As the number of meals (data) increases, the number of chefs (processors) can be increased if there are sufficient resources, such as stoves, cutting boards, etc. (memory, interconnection network bandwidth, etc.)

Data Parallelism

- Also known as loop-level parallelism
- Performing the same operation to different data items at the same time
- More data, more parallelism

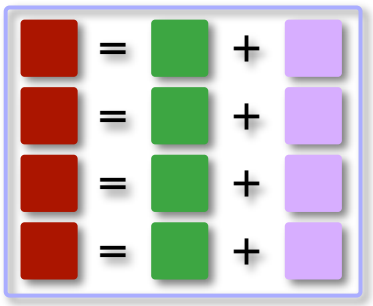
```
for(i = 0; i < 16; i++)
{
    c[i] = a[i] + b[i];
}
```



Data Parallelism (contd.)

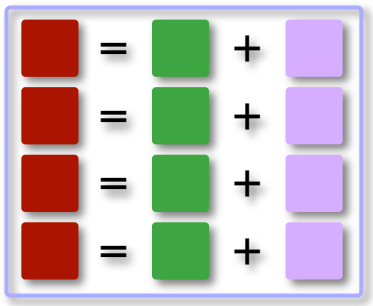
```
for(i = 0; i < 16; i++)
{
    c[i] = a[i] + b[i];
}
```

```
for(i = 0;i < 4; i++)
{
  c[i] = a[i] + b[i];
}
```



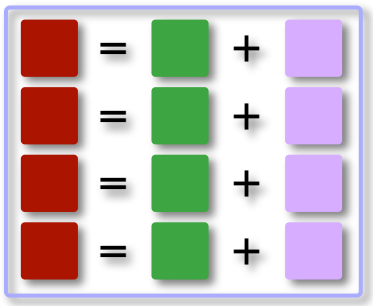
Core 0

```
for(i = 4;i < 8; i++)
{
  c[i] = a[i] + b[i];
}
```



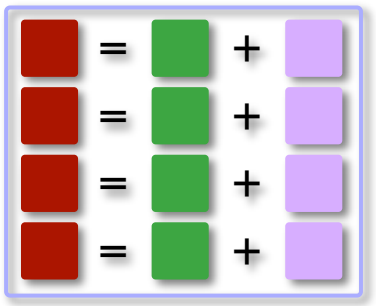
Core 1

```
for(i = 8; i < 12; i++)
{
  c[i] = a[i] + b[i];
}
```



Core 2

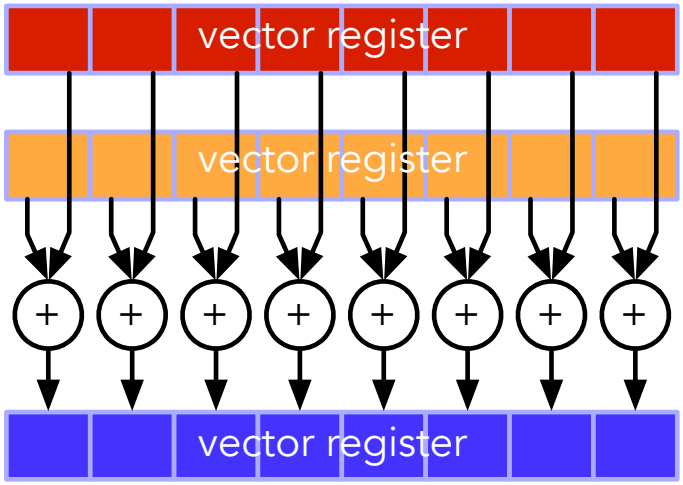
```
for(i = 12; i < 16; i++)
{
  c[i] = a[i] + b[i];
}
```



Core 3

SIMD

- Single Instruction, Multiple Data
- A single instruction (or copies of a single instruction) performs the same operation in parallel on multiple data items of the same type and size
  - A single program counter
- Compilers typically support auto-vectorization
- Multiple parallel execution units are called ***lanes***



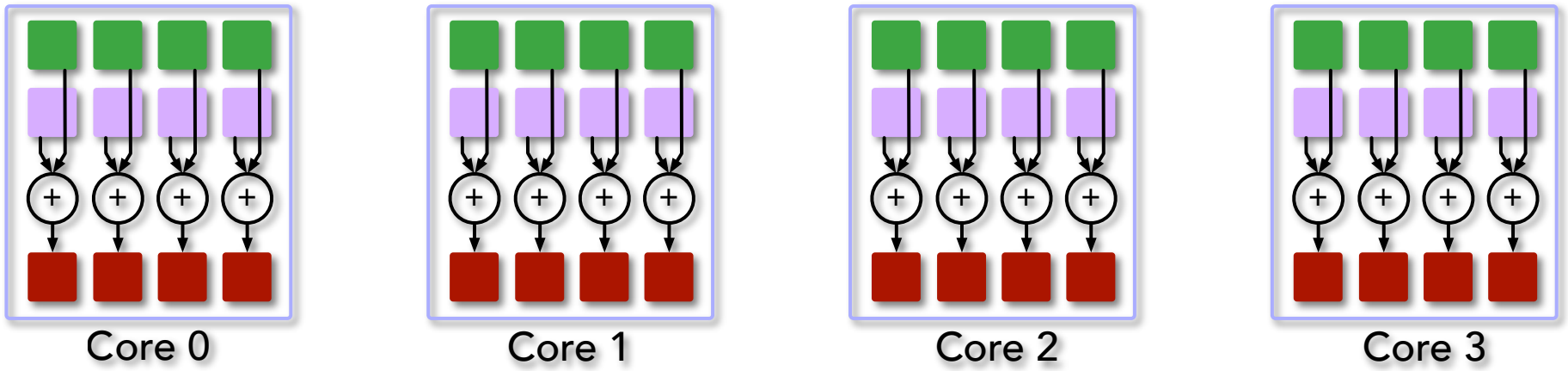
SIMD (contd.)

```
for(i = 0;i < 4; i++)
{
  c[i] = a[i] + b[i];
}
```

```
for(i = 4;i < 8; i++)
{
  c[i] = a[i] + b[i];
}
```

```
for(i = 8; i < 12; i++)
{
  c[i] = a[i] + b[i];
}
```

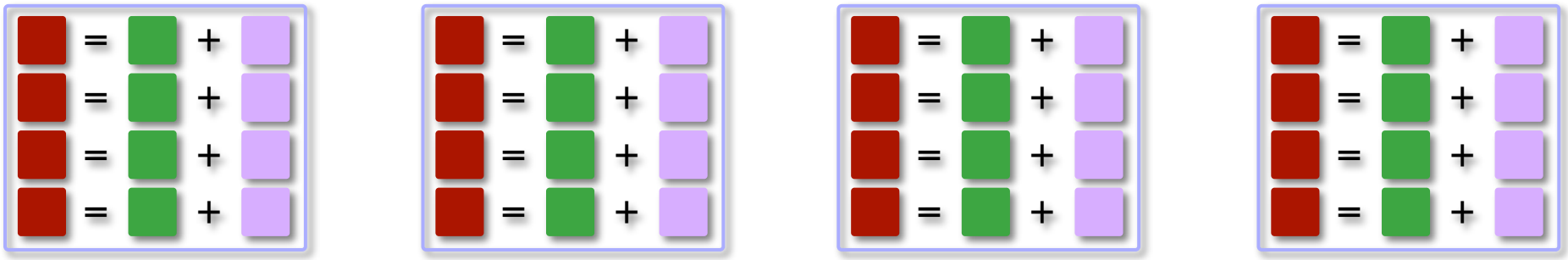
```
for(i = 12; i < 16; i++)
{
  c[i] = a[i] + b[i];
}
```



SPMD

- Single Program, Multiple Data
- Running the same code on different data
- ***Single conceptual thread of control***

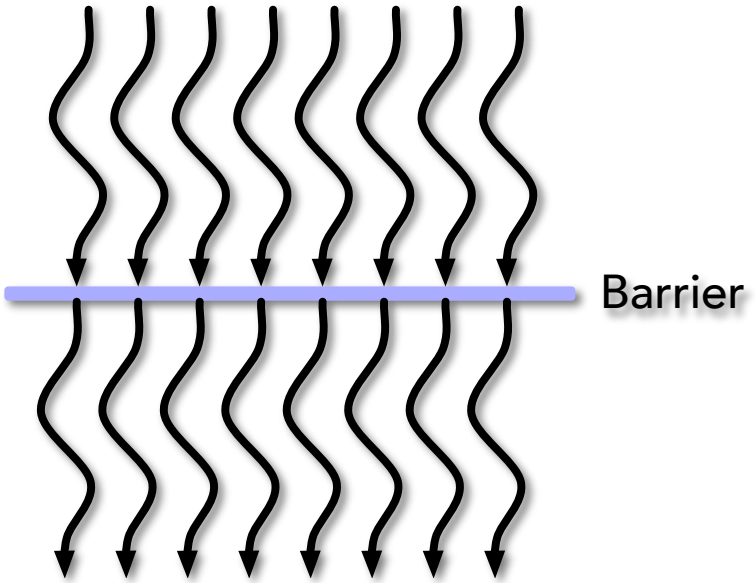
```
LL = myid() * 4;
UL = LL + 4
for(i = LL; i < UL; i++)
{
    c[i] = a[i] + b[i];
}
```



Barrier Synchronization

- Any thread must stop at the barrier and cannot proceed until all other threads reach the barrier

```
LL = myid() * 4;
UL = LL + 4
for(i = LL; i < UL; i++) {
    c[i] = a[i] + b[i];
    barrier();
    d[i] = c[i+1] + c[i-1];
}
```

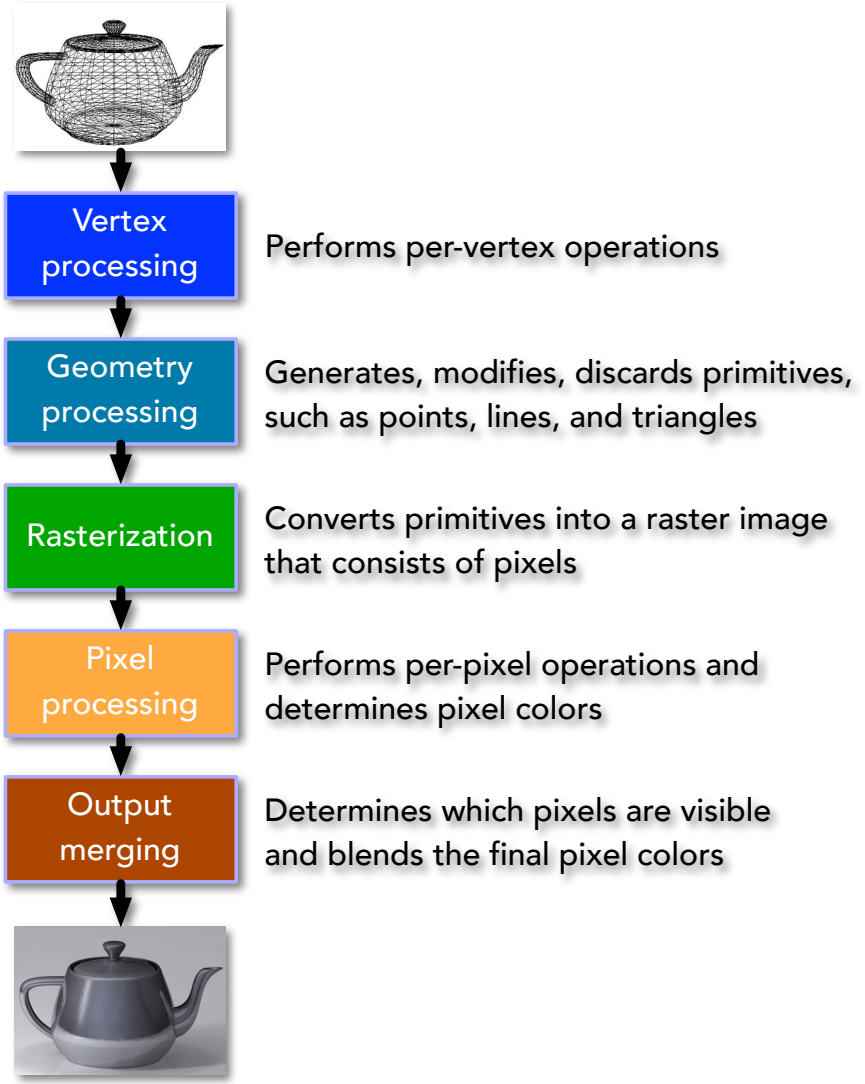


Part I: Introduction to OpenCL

- Heterogeneous computing and OpenCL
- Background for parallel processing
- GPU architectures (based on "Kayvon Fatahalian, From Shader Code to a Teraflop: How a Shader Core Works, ACM SIGGRAPH 2010 Class: Beyond Programmable Shading I")
- Intel Xeon Phi coprocessor
- Introduction to the OpenCL framework
- OpenCL for a single compute device
- OpenCL for multiple compute devices
- OpenCL+MPI for clusters
- HPL in OpenMP+MPI
- Limitations of the OpenCL programing model

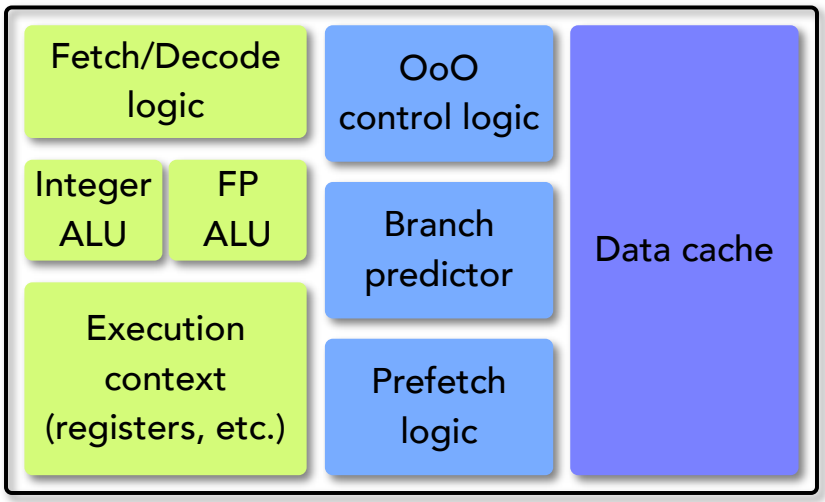
Rendering (Graphics) Pipeline

- Rendering
  - The process of generating an image from a 3D model
- Supported by commodity hardware
- Some stages are programmable
  - Vertex
  - Geometry
  - Pixel

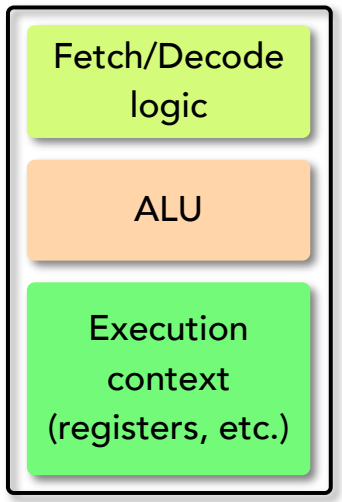


Shader Cores

- Shader
  - A program (function) that runs on the GPU to implement one of the programmable stages of the rendering pipeline
- ***Remove components that help a single instruction stream run fast***
  - Very simple, programmable core



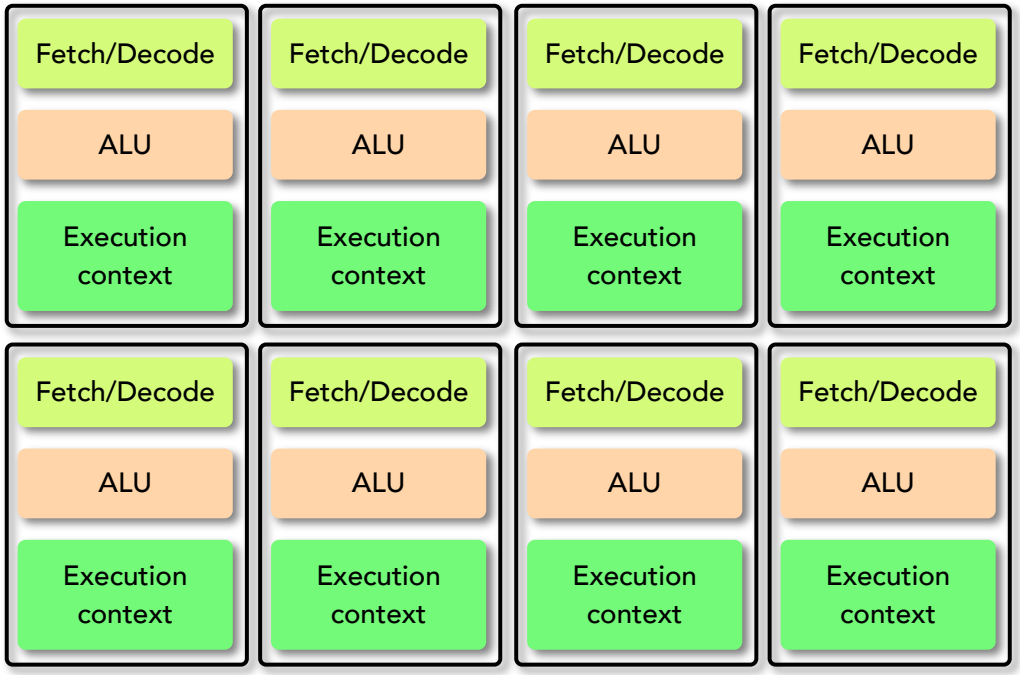
General-purpose CPU core



Shader core

Exploiting Characteristics of GPU Applications

- Data independence between triangles and pixels
  - Millions of triangles and pixels
  - Enables massively parallel processing
- ***Increase the number of cores***

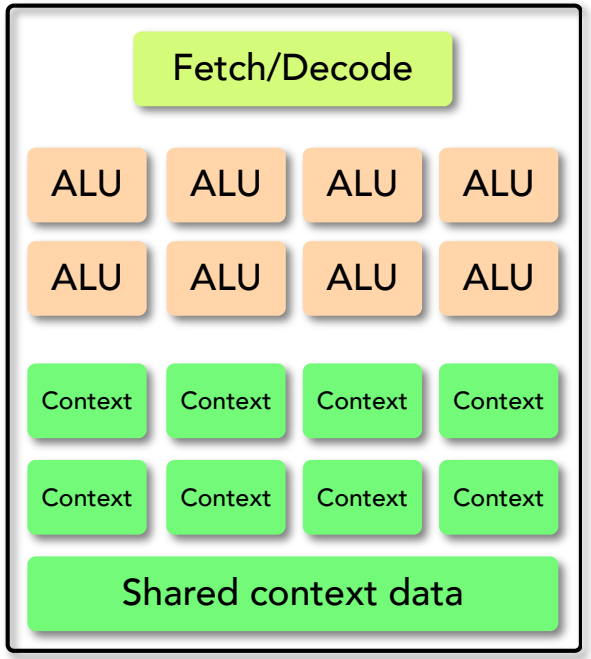


Exploiting Characteristics of GPU Applications (contd.)

- Instruction streams for pixels are not actually very different

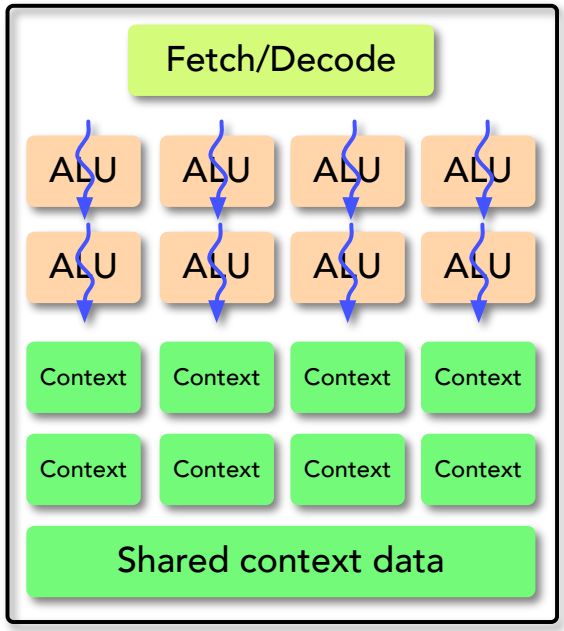
- Amortize cost/complexity of managing an instruction stream across many ALUs
  - Streaming multiprocessor

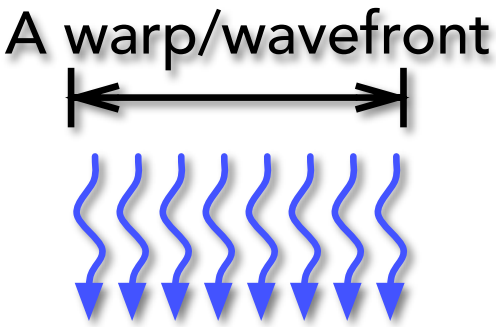
- SIMD processing
  - Implicit hardware vectorization using a scalar instruction
  - A single program counter across multiple ALUs



Warp vs. Wavefront

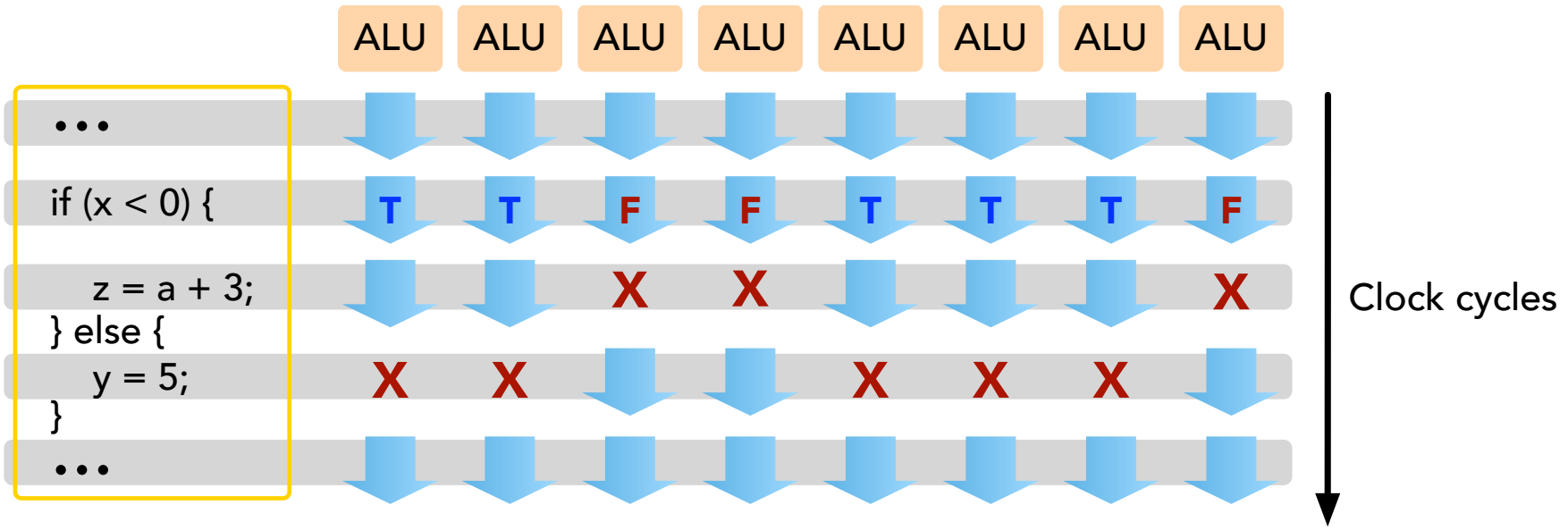
- The group of the same instruction execution sequences running on the ALUs in parallel
  - NVIDIA - Warp
    - SIMT: Single Instruction, Multiple Threads
    - All threads in a warp execute the same instruction at a time
  - AMD - Wavefront





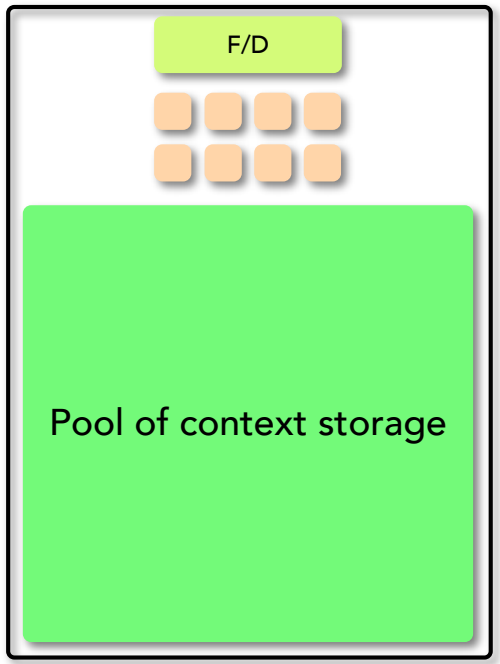
Executing Branches

- Conditional execution
- Not all ALUs do useful work



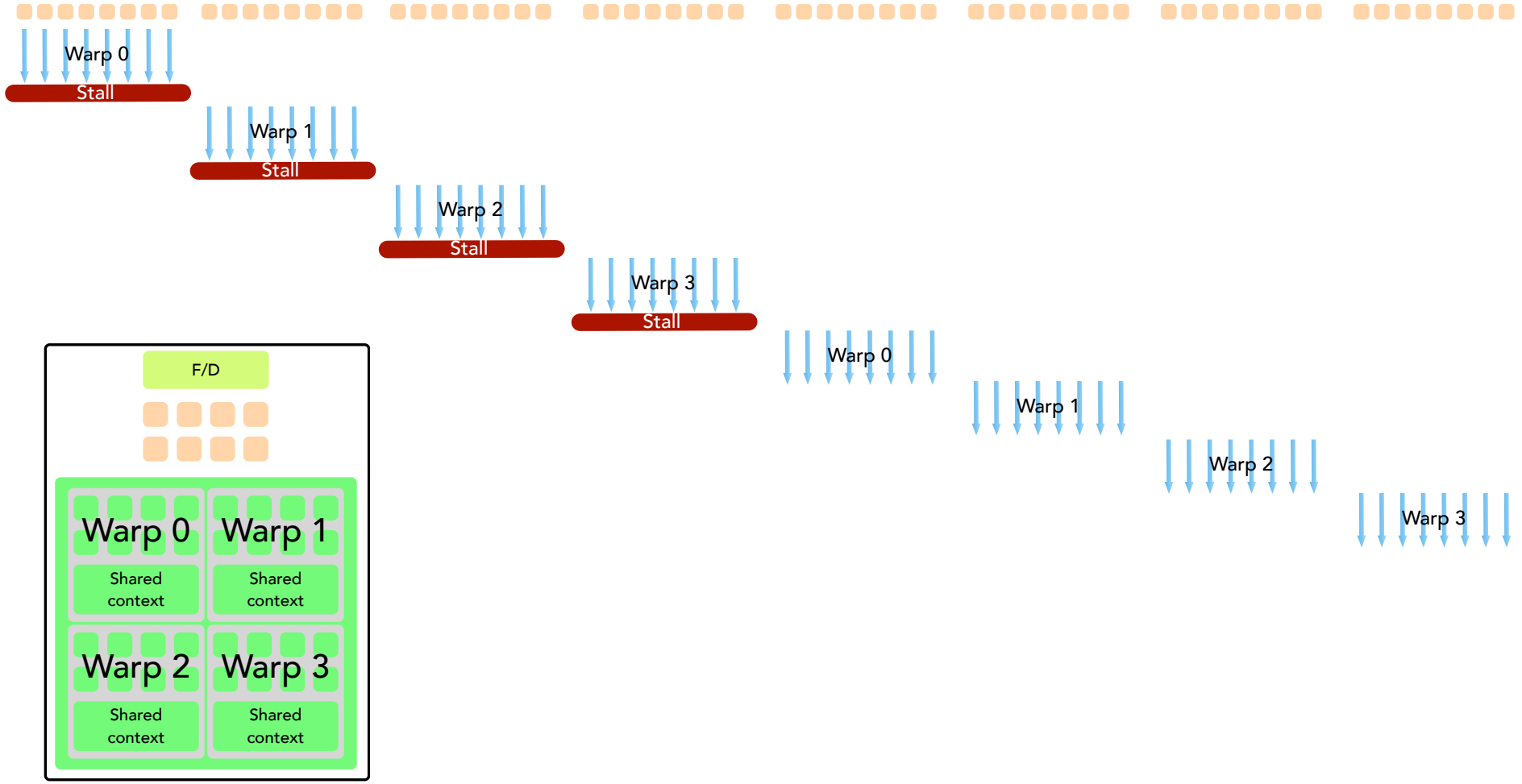
Hardware Context Switch

- To avoid stalls caused by high latency operations
- ***Interleave processing of many warps on a single streaming multiprocessor to avoid stalls***
- Unit of context switch: warp



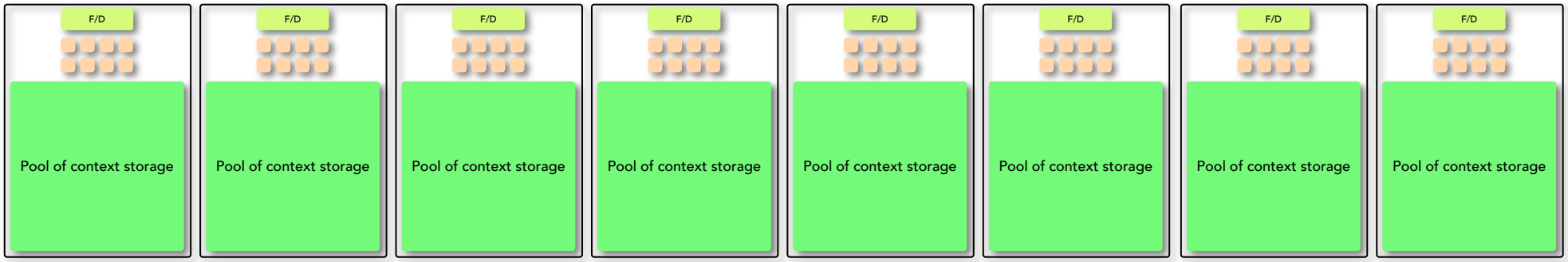
Hardware Context Switch (contd.)

- Increases throughput

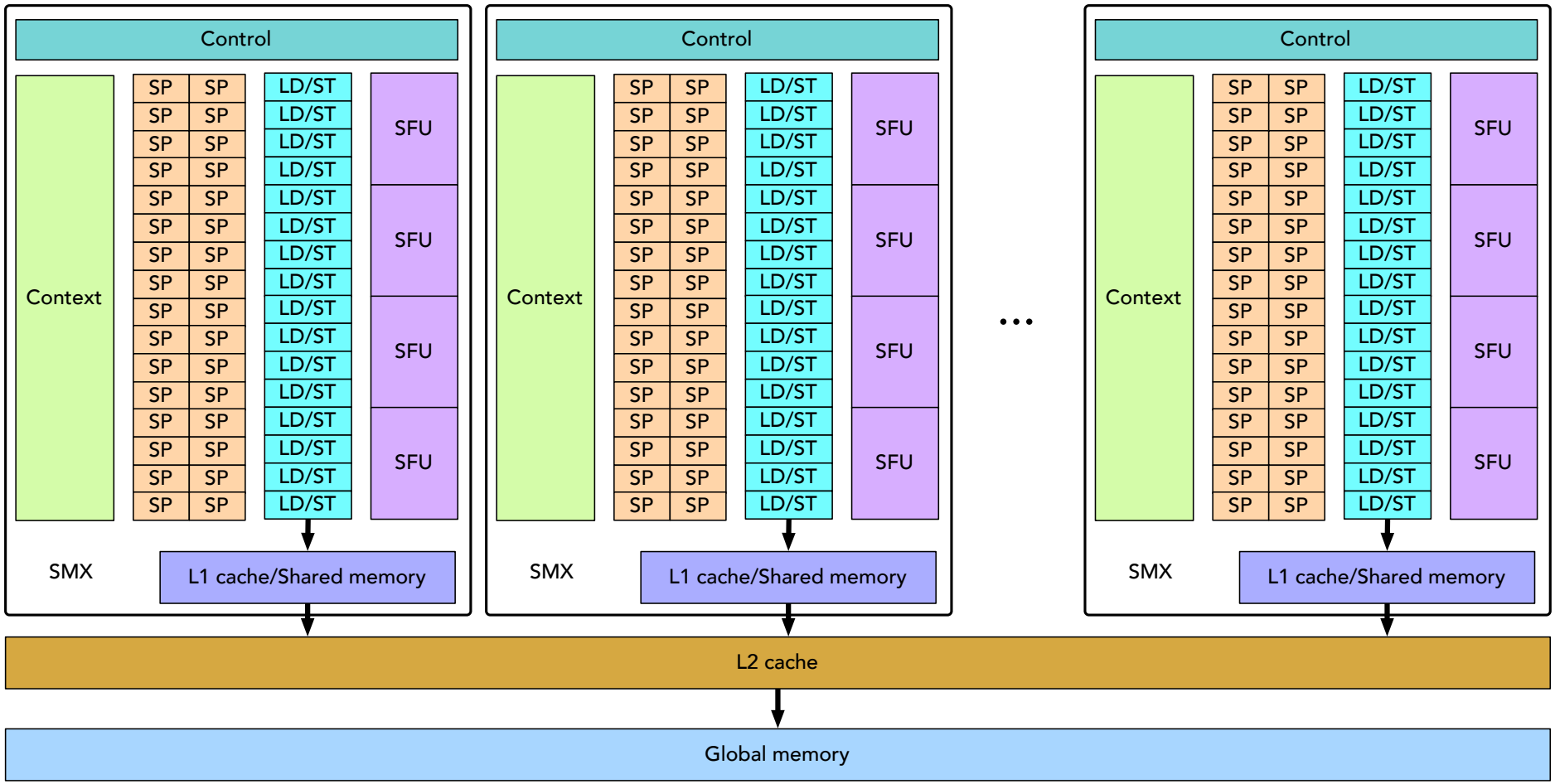


Characteristics of GPU Applications (contd.)

- ***Exploit parallelism across different shaders***
  - For different instruction streams
  - Multiple streaming multiprocessors



Finally ...


Control
Context
SP
LD/ST
SFU
SMX
L1 cache/Shared memory
L2 cache
Global memory

GPU Summary

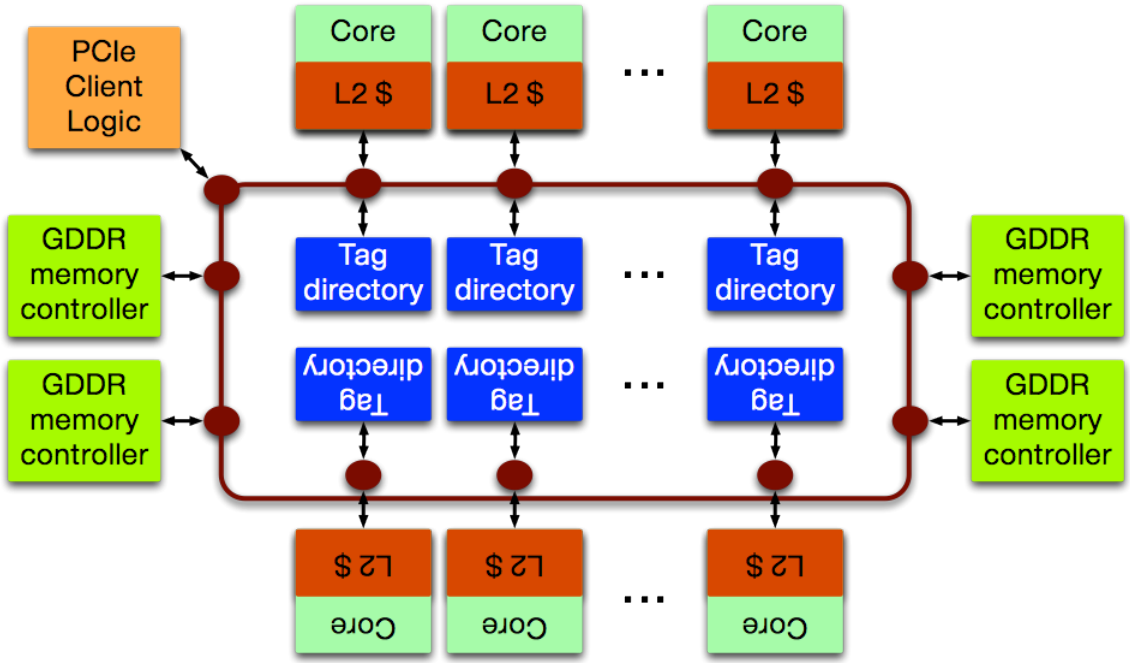
- Exploits massive data parallelism
  - Many simple compute units
  - SIMD processing (sharing an instruction stream across compute units)
- Hardware context switch
  - To tolerate high latencies

Part I: Introduction to OpenCL

- Heterogeneous computing and OpenCL
- Background for parallel processing
- GPU architectures
- Intel Xeon Phi coprocessor
- Introduction to the OpenCL framework
- OpenCL for a single compute device
- OpenCL for multiple compute devices
- OpenCL+MPI for clusters
- HPL in OpenMP+MPI
- Limitations of the OpenCL programing model

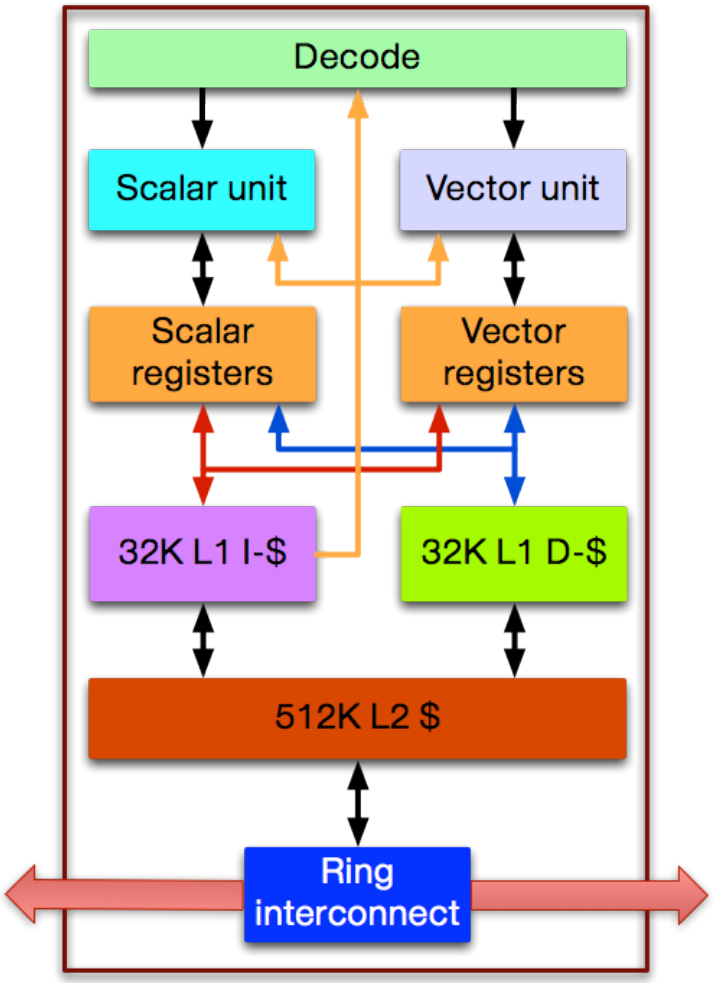
Intel Xeon Phi Coprocessors

- 60 Intel MIC Architecture cores and 240 threads per coprocessor
- Bidirectional ring (200 GB/sec)
- PCI Express form-factor
- 8GB GDDR Memory
- 8 memory controllers, 16 32-bit channels



Intel Xeon Phi Coprocessors (contd.)

- In-order x86 core
  - 4 hardware threads per core
  - 1 cycle scalar execution latency
  - 64-bit addressing
- Pentium processor family-base scalar units
- Fully-coherent L1 and L2 caches
  - Directory-based coherence protocol
- ***Vector unit***
  - 512-bit SIMD Instructions
  - 32 vector registers
  - Hold 16 singles or 8 doubles per register
  - Pipelined one-per-clock throughput
  - 4 clock latency, hidden by round-robin scheduling of threads



Xeon Phi Software Stack

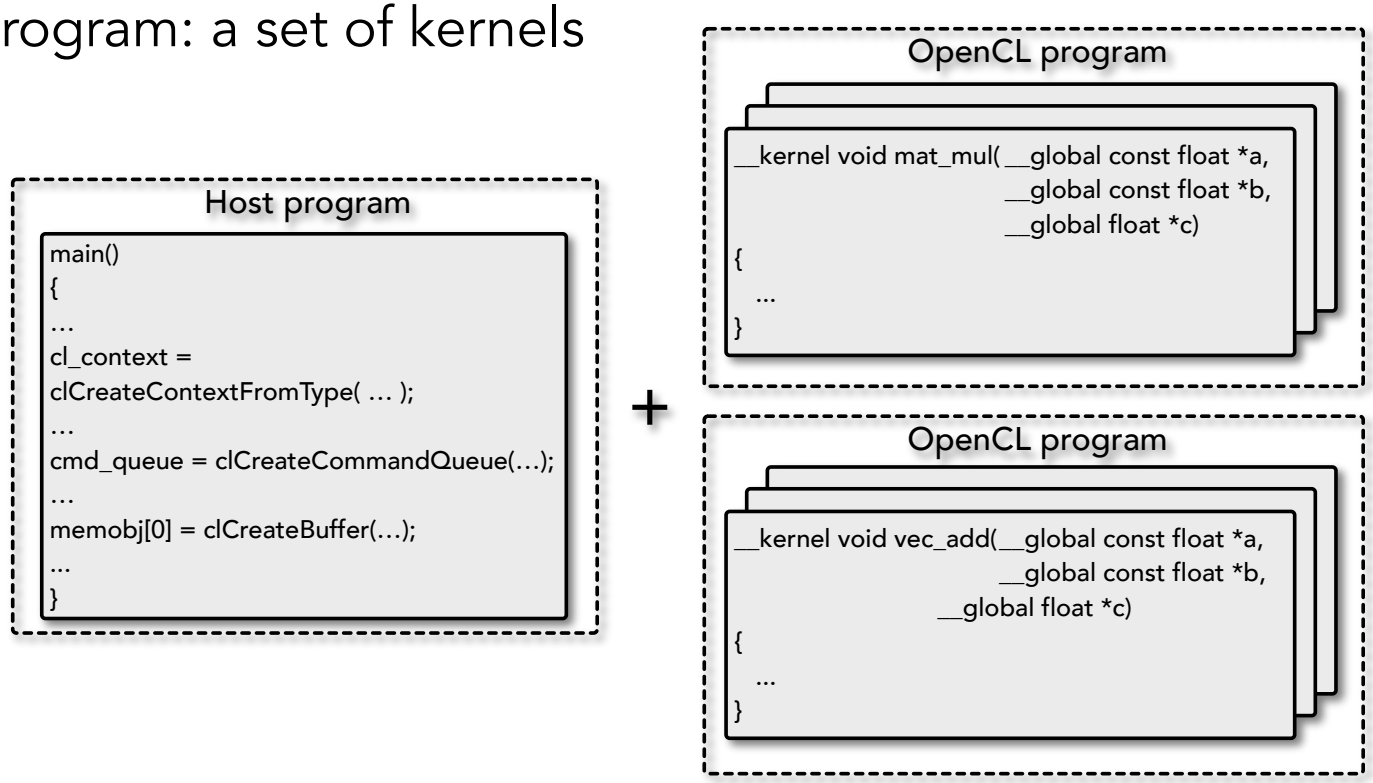
- Device driver
  - Device initialization, communication between the host and target cards
- Libraries
  - Buffer management
  - Host-to-card communication
  - Loading and unloading executables onto card
  - Invoking functions
- Card OS
  - The Linux-based OS running on the Intel Xeon Phi Coprocessor

Part I: Introduction to OpenCL

- Heterogeneous computing and OpenCL
- Background for parallel processing
- GPU architectures
- Intel Xeon Phi coprocessor
- Introduction to the OpenCL framework
- OpenCL for a single compute device
- OpenCL for multiple compute devices
- OpenCL+MPI for clusters
- HPL in OpenMP+MPI
- Limitations of the OpenCL programing model

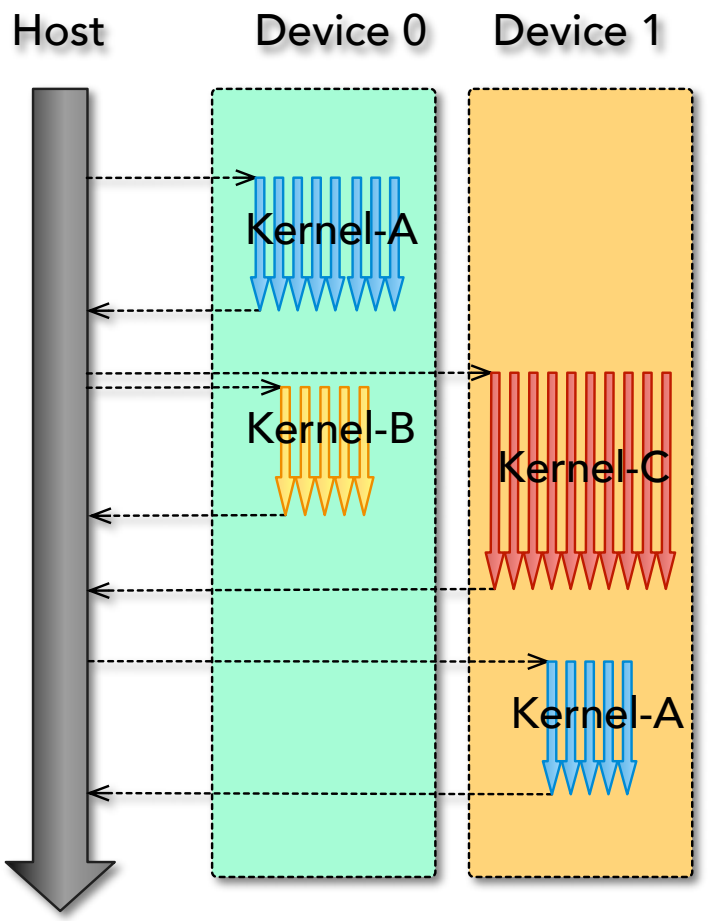
OpenCL Applications

- The combination of programs running on a host processor and OpenCL compute devices
  - Compute devices: CPUs, GPUs, Xeon Phi coprocessors, etc.
- A host program + OpenCL programs
  - OpenCL program: a set of kernels



OpenCL Application (contd.)

- Host program
  - Executes on the host and manages kernel execution
- Kernels
  - Basic unit of executable code (a function) on compute devices
  - When executed, many instances are created
    - SPMD
    - Exploits data parallelism
    - Can control the number of instances
- The host program and kernels all run in parallel

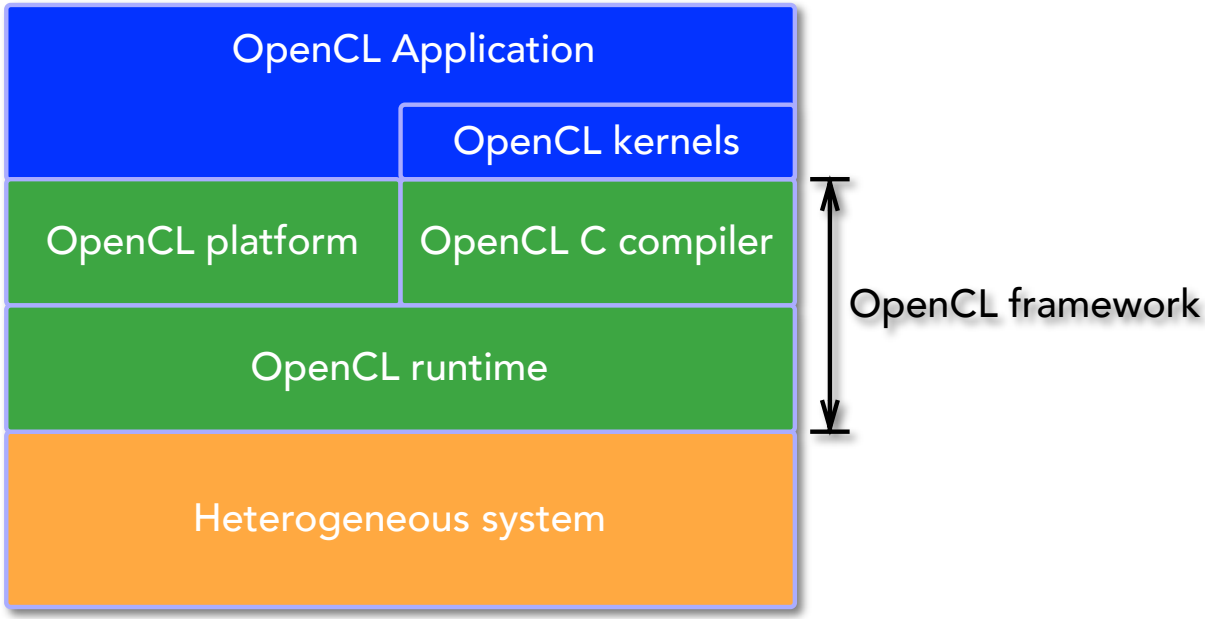


OpenCL C Language

- For kernels
- Based on ISO C99
- Restrictions
  - No standard C99 headers, function pointers, recursion, variable length arrays, and bit fields
- Extensions
  - Vector types
  - Image types
  - Synchronization
  - Address space qualifiers
  - Built-in functions

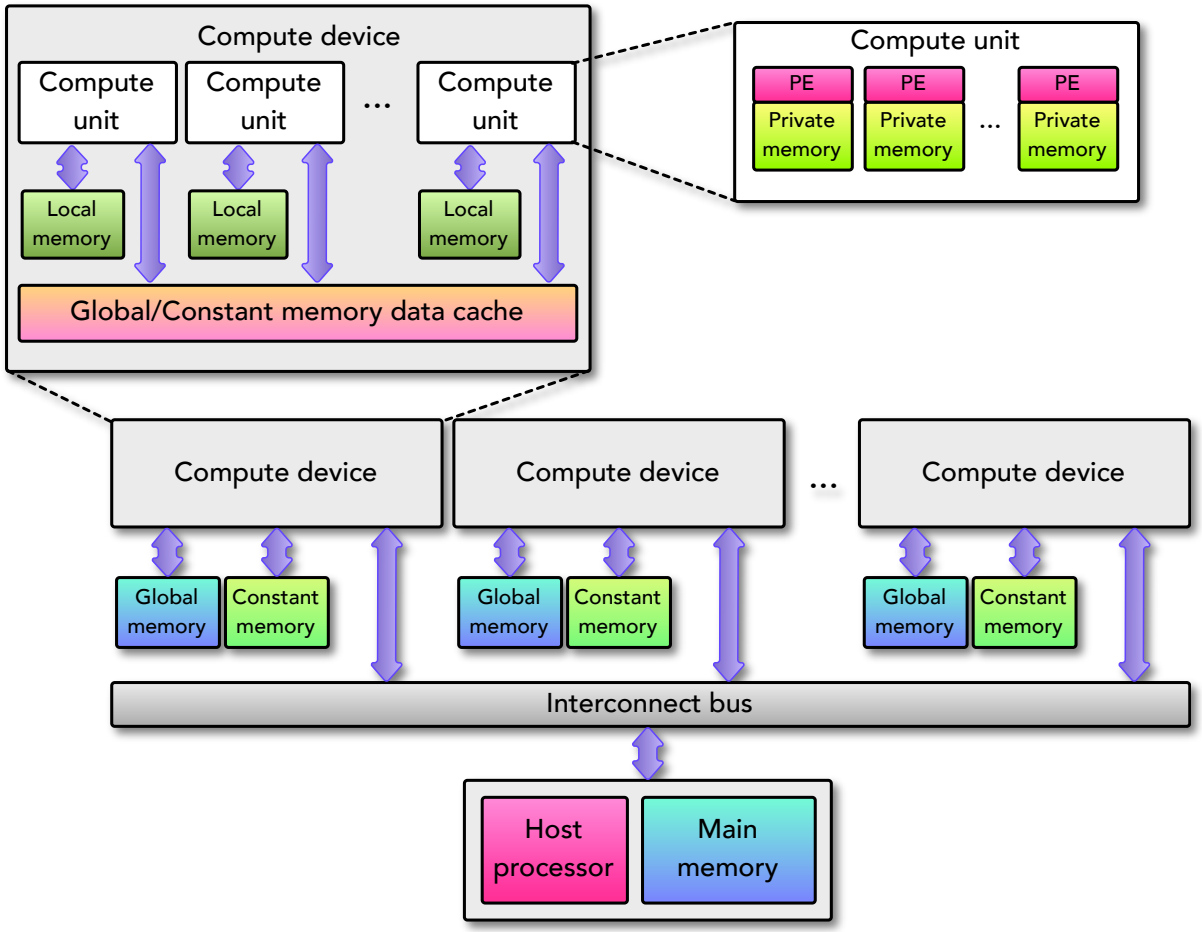
OpenCL Framework

- A software system that contains the set of components to support OpenCL application development and execution
  - To use a host and one or more compute devices as a single system
- OpenCL platform layer + OpenCL runtime + OpenCL C compiler



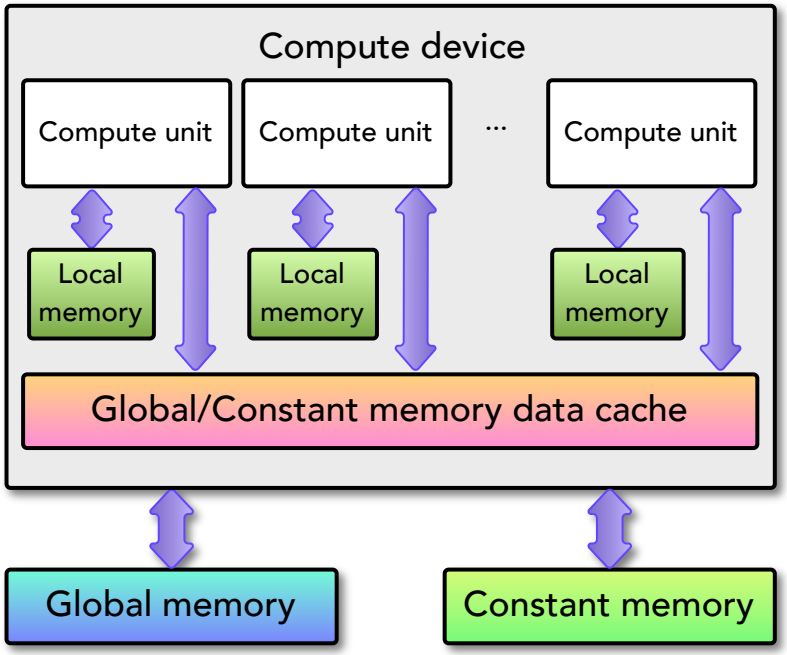
OpenCL Platform Model

- One host + one or more compute devices (e.g., GPUs)
- The host can access global memory and constant memory (read/write)
- Compute devices may not access main memory



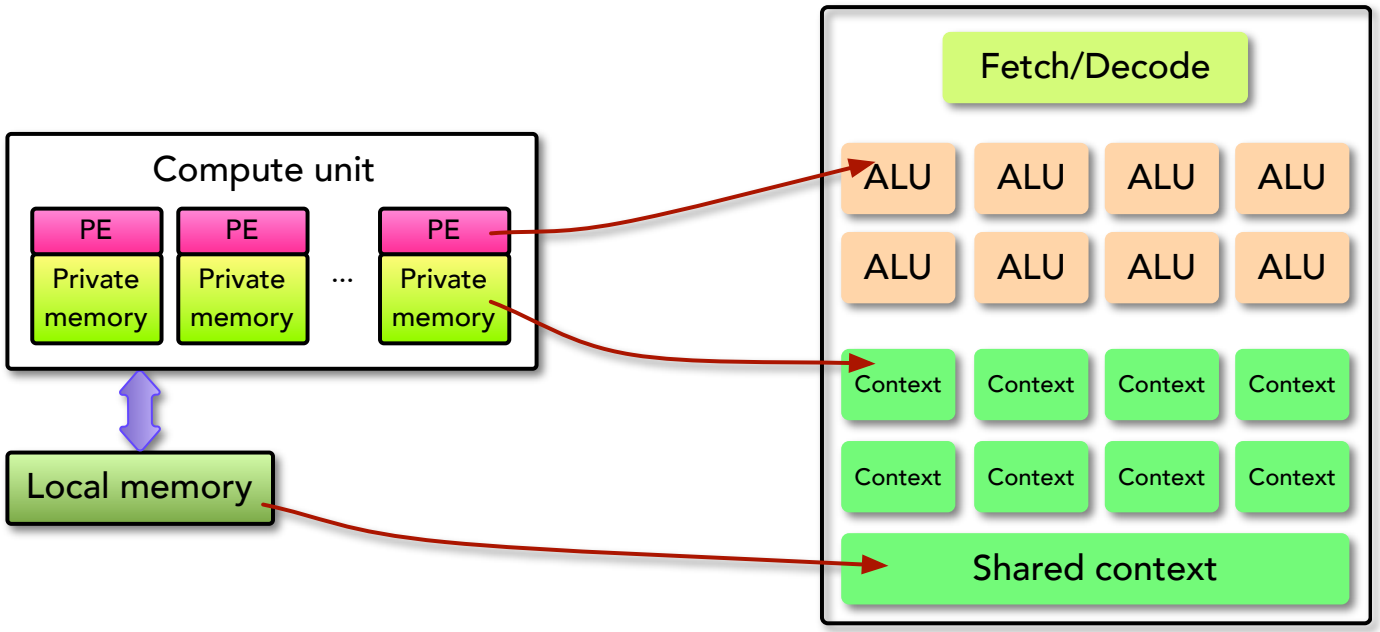
Compute Device

- A collection of one or more Compute Units (CUs)
- Constant memory is read-only for compute devices



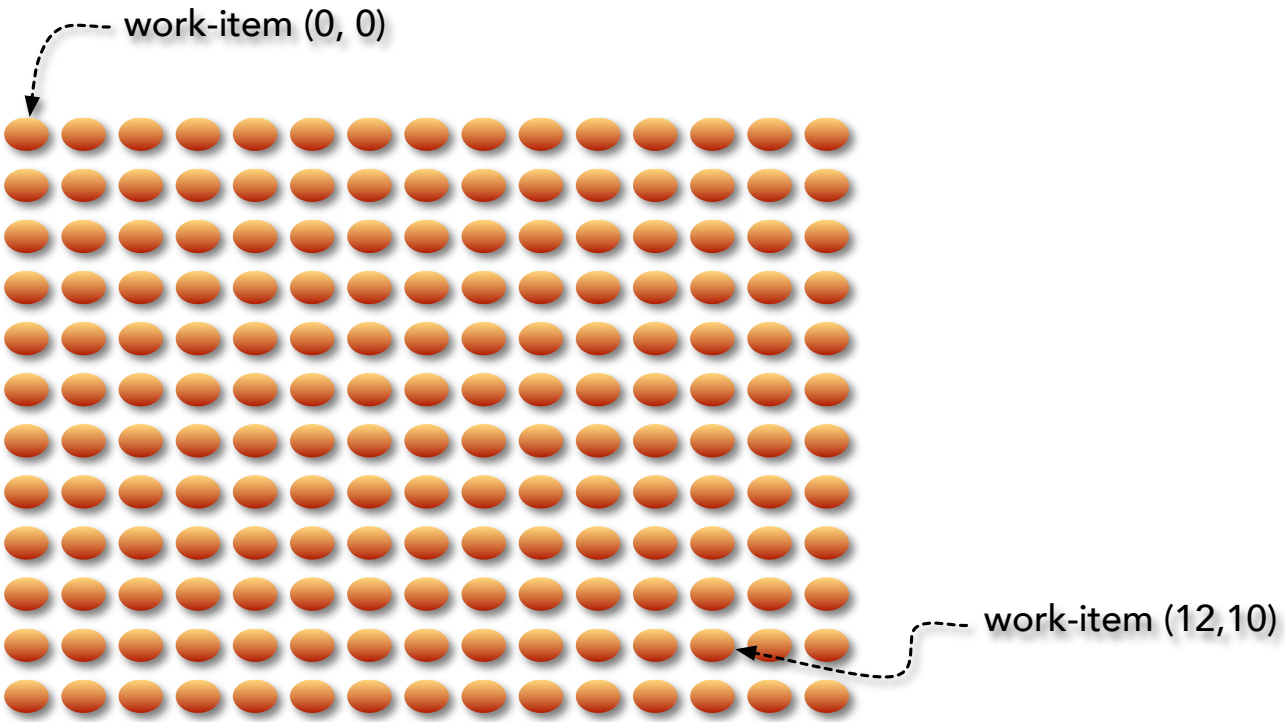
Compute Unit

- A collection of one or more Processing Elements (PEs)
- Each PE has its own private memory
- Local memory
  - Shared by PEs
  - Not visible to other CUs



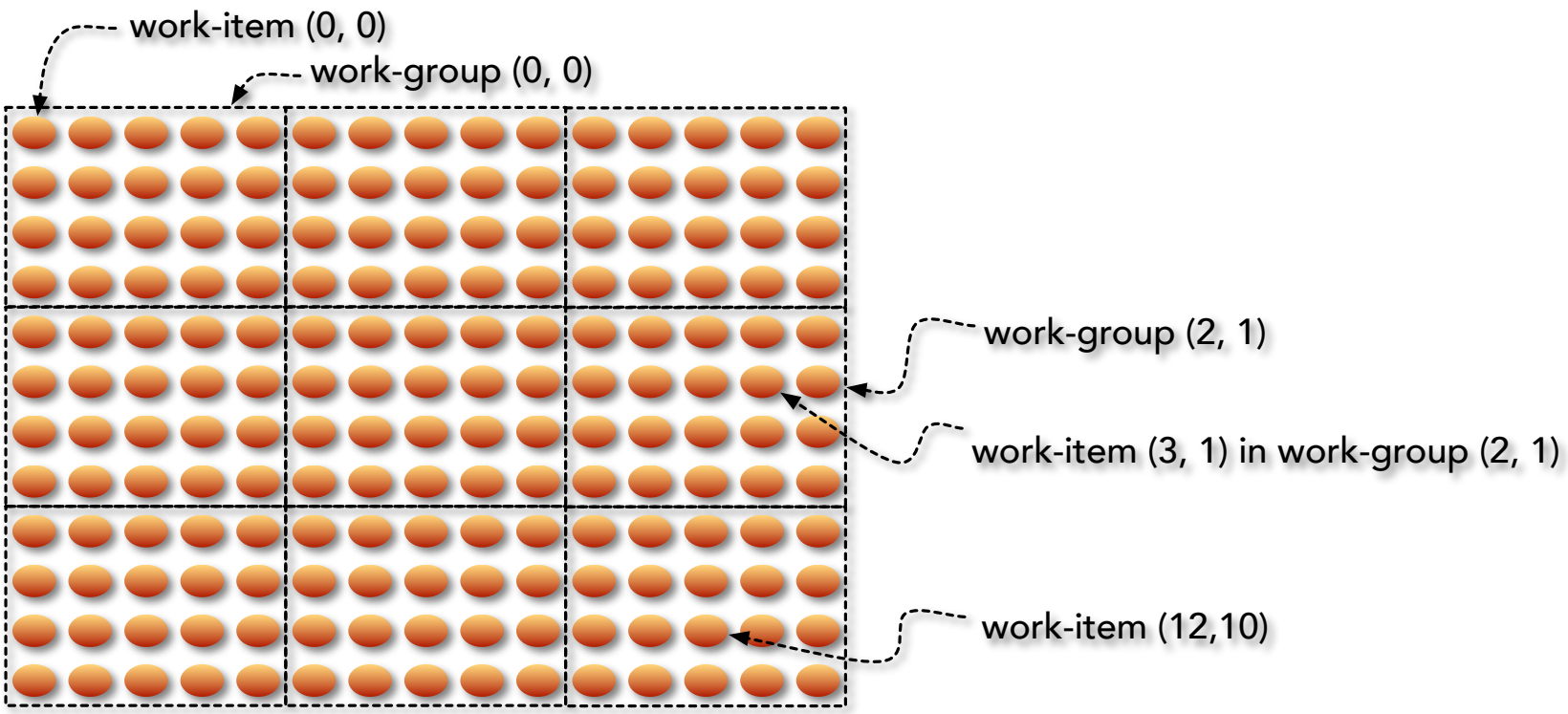
Kernel Index Space

- Defines the total number of ***work-items (kernel instances)*** that execute in parallel
- N-dimensional index space (N = 1, 2, or 3)
- A work-item executes for each point in the space
  - Has a unique global ID



Work-groups

- Work-items are organized into ***work-groups***
  - A work-item in a work-group has a unique local ID
- A work-group executes on a compute unit
- Choose the dimension and size that are best for your kernel



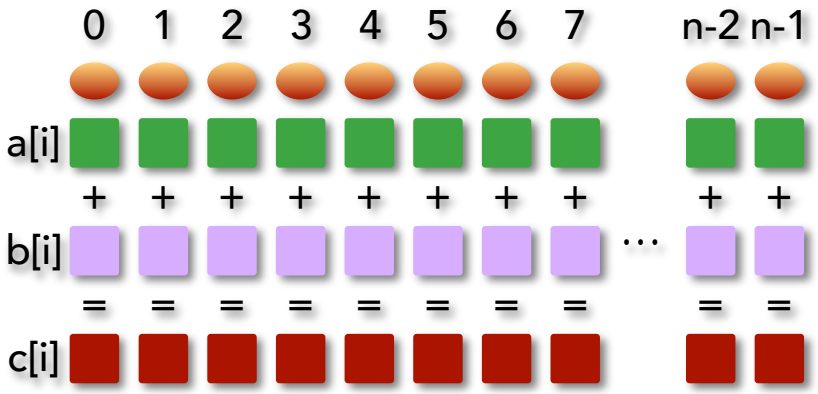
Kernel Example

```
void vec_add(int n,
             const float *a,
             const float *b,
             float *c)
{
    int i;
    for (i = 0; i < n; i++)
        c[i] = a[i] + b[i];
}
```

```
__kernel void vec_add( __global const float *a,
                       __global const float *b,
                       __global float *c)
{
    int id = get_global_id(0);

    c[id] = a[id] + b[id];
}
```

Data Parallel Programming Model

- A set of instructions from the kernel are applied concurrently to each point in the kernel index space
  - SPMD
- The index space defines how the data maps onto the work-items



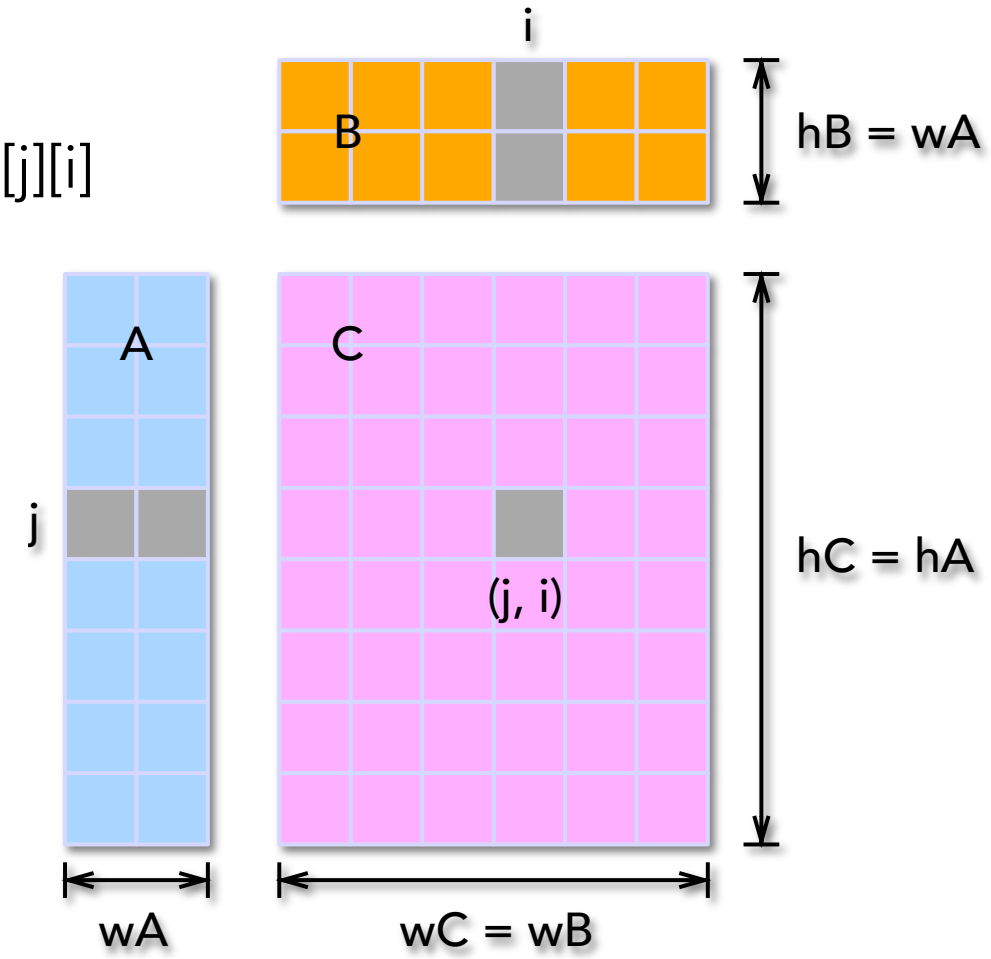

```
__kernel void vec_add( __global const float *a,
                       __global const float *b,
                       __global float *c)
{
    int id = get_global_id(0);

    c[id] = a[id] + b[id];
}
```

Part I: Introduction to OpenCL

- Heterogeneous computing and OpenCL
- Background for parallel processing
- GPU architectures
- Intel Xeon Phi coprocessor
- Introduction to the OpenCL framework
- OpenCL for a single compute device
- OpenCL for multiple compute devices
- OpenCL+MPI for clusters
- HPL in OpenMP+MPI
- Limitations of the OpenCL programing model

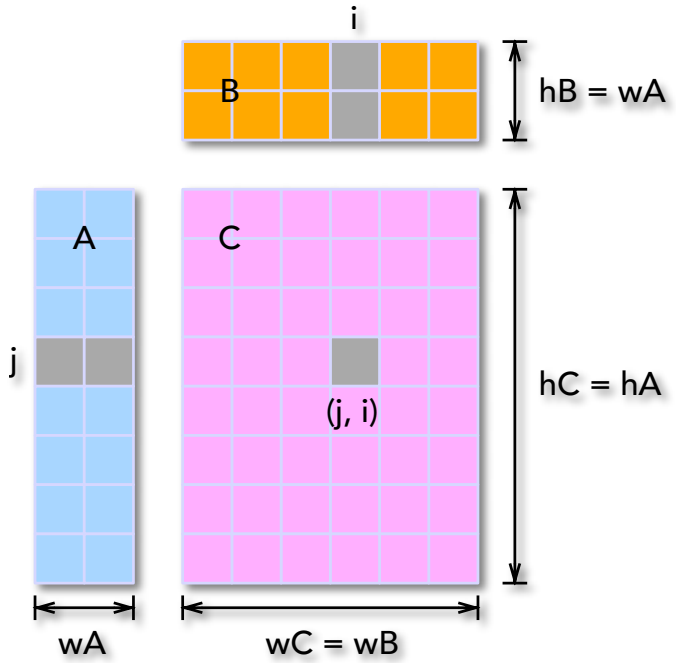
Matrix Multiply

- Each work-item computes an element of C
  - Load a row j of A
  - Load a column i of B
  - Compute the dot product C[j][i]



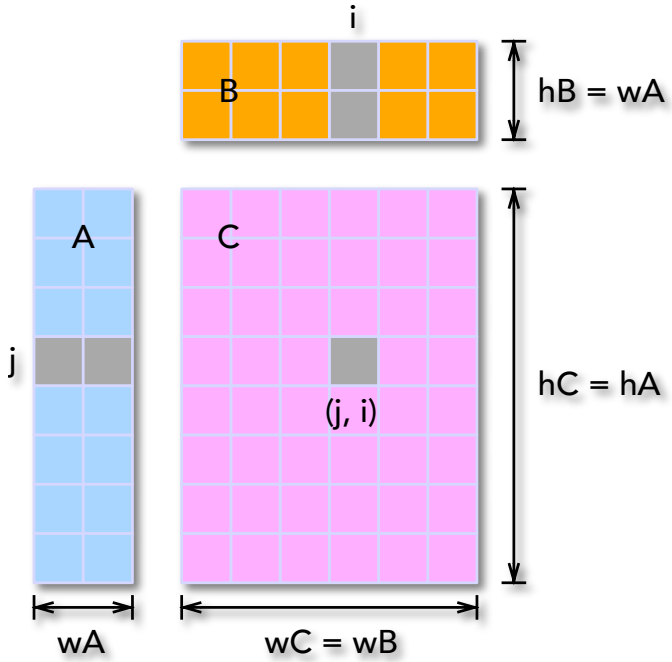
Kernel for Matrix Multiply

```
__kernel void matrix_mul(__global float* C,
                         __global float* A,
                         __global float* B,
                         int wA,
                         int wB)
{
    int i = get_global_id(0);
    int j = get_global_id(1);
    int k;

    float acc = 0.0;
    for (k = 0; k < wA; k++)
        acc += A[j * wA + k] * B[k * wB + i];
    C[j * wB + i] = acc;
}
```



Host Program for Matrix Multiply

```
//////////////////
// Host program //
//////////////////
#include <CL/cl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// wA == hB, hC == hA, wC == wB
#define wA_SIZE  1024
#define hA_SIZE  1024
#define wB_SIZE  1024

const char* kernel_src = // Kernel source code
    "__kernel void matrix_mul(__global float* C,"
    "                         __global float* A,"
    ...
    "}";
```



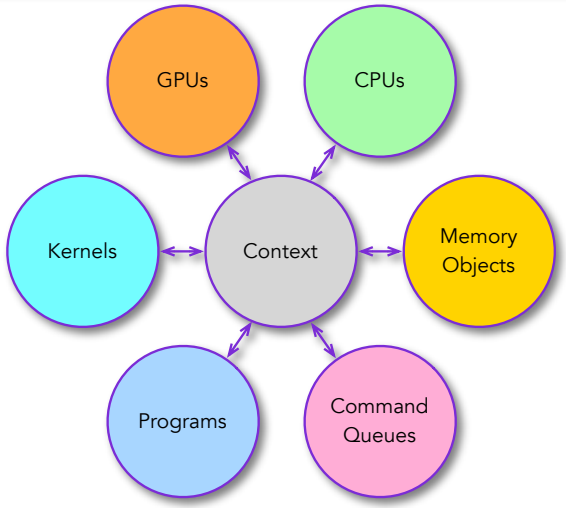
Space Allocation to Matrices

```
int main(int argc, char** argv)
{
    int  wA = wA_SIZE, hA = hA_SIZE;
    int  wB = wB_SIZE, hB = wA;
    int  wC = wB, hC = wA;

    size_t sizeA = wA * hA * sizeof(float);
    size_t sizeB = wB * hB * sizeof(float);
    size_t sizeC = wC * hC * sizeof(float);

// Allocate memory spaces to matrices hostA, hostB, and hostC
    float *hostA = (float*) malloc(sizeA);
    float *hostB = (float*) malloc(sizeB);
    float *hostC = (float*) malloc(sizeC);

// Initialize hostA and hostB
  ...
```

Creating OpenCL Context

```
// Obtain a list of available OpenCL platforms
    cl_platform_id platform;
    clGetPlatformIDs(1, &platform, NULL);

// Obtain the list of available devices on the OpenCL platform
    cl_device_id device;
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);

// Create an OpenCL context on a GPU device
    cl_context context;
    context = clCreateContext(0, 1, &device, NULL, NULL, NULL);
```

- OpenCL context
  - The environment in which the kernels execute
  - The domain in which synchronization and memory management is defined



Creating Command-queues

```
// Create a command queue and attach it to the compute device. The
// third argument specifies if it is an in-order or out-of-order
//(CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE) queue
    cl_command_queue  command_queue;
    command_queue = clCreateCommandQueue(context, device, 0, NULL);
```

- Commands
  - OpenCL operations that are submitted to command-queues for execution
    - Kernel execution commands
    - Memory commands
    - Synchronization commands
- Command-queues
  - Contain commands that will be executed on a specific compute device
    - Created by the host program and attached to a specific compute device
    - Commands are issued in-order or out-of-order

Allocating Memory Objects

```
// Allocate buffer memory objects
    cl_mem bufferA;
    cl_mem bufferB;
    cl_mem bufferC;

    bufferA = clCreateBuffer(context, CL_MEM_READ_ONLY,
                             sizeA, NULL, NULL);
    bufferB = clCreateBuffer(context, CL_MEM_READ_ONLY,
                             sizeB, NULL, NULL);
    bufferC = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                             sizeC, NULL, NULL);
```

- Allocate OpenCL memory objects to store the data accessed by the kernel
  - No device binding to the memory objects yet
- Two types of memory objects
  - Buffer objects and image objects

Creating the Kernel

```
// Create an OpenCL program object for the context
// and load the kernel source into the program object
    size_t kernel_src_len = strlen(kernel_src);

    cl_program program;
    program = clCreateProgramWithSource(context, 1,
                                        (const char**) &kernel_src,
                                         &kernel_src_len, NULL);

// Build (compile and link) the program executable
// from the source or binary for the device
    clBuildProgram(program, 1, &device, NULL, NULL, NULL);

// Create a kernel object from the program
    cl_kernel kernel;
    kernel = clCreateKernel(program, "matrix_mul", NULL);
```

Setting up Kernel Arguments (contd.)

```
// Set the arguments of the kernel

    clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*) &bufferC);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), (void*) &bufferA);
    clSetKernelArg(kernel, 2, sizeof(cl_mem), (void*) &bufferB);
    clSetKernelArg(kernel, 3, sizeof(cl_int), (void*) &wA);
    clSetKernelArg(kernel, 4, sizeof(cl_int), (void*) &wB);
```

```
__kernel void matrix_mul(__global float* C,
                         __global float* A,
                         __global float* B,
                         int wA,
                         int wB)
```

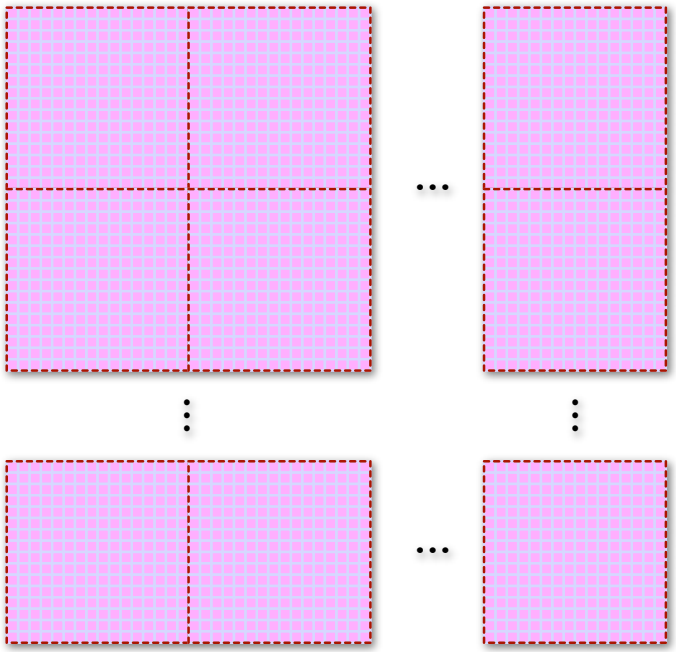
- Arguments to the kernel are referred by indices that go from 0 for the leftmost argument to n-1, where n is the total number of arguments declared by the kernel

Preparing Input Data to the Kernel

```
// Copy the input matrices to the corresponding buffers
// The third argument CL_FALSE specifies a non-blocking (CL_FALSE)
// or blocking (CL_TRUE) write
    clEnqueueWriteBuffer(command_queue, bufferA, CL_FALSE, 0,
                         sizeA, hostA, 0, NULL, NULL);

    clEnqueueWriteBuffer(command_queue, bufferB, CL_FALSE, 0,
                         sizeB, hostB, 0, NULL, NULL);
```

Setting up the Kernel Index Space

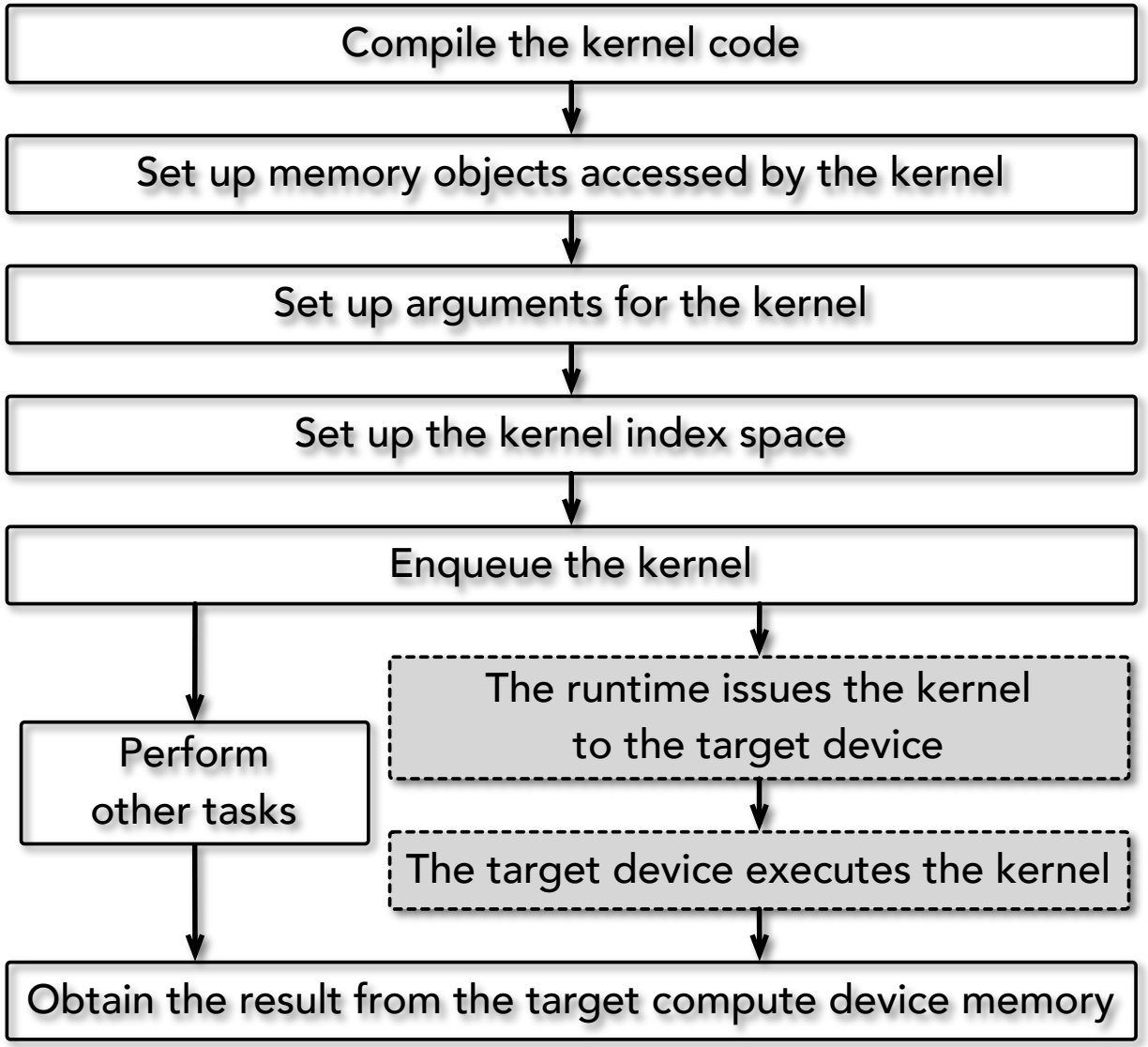
```
// The kernel index space is two dimensional. Specify the number of
// work-items in each dimension of the kernel index space
    size_t global[2] = { wC, hC };

// The number of total work-items in a work-group. Specify the
// number of work-items in each dimension of the work-group
    size_t local[2] = { 16, 16 };
```



Launching the Kernel

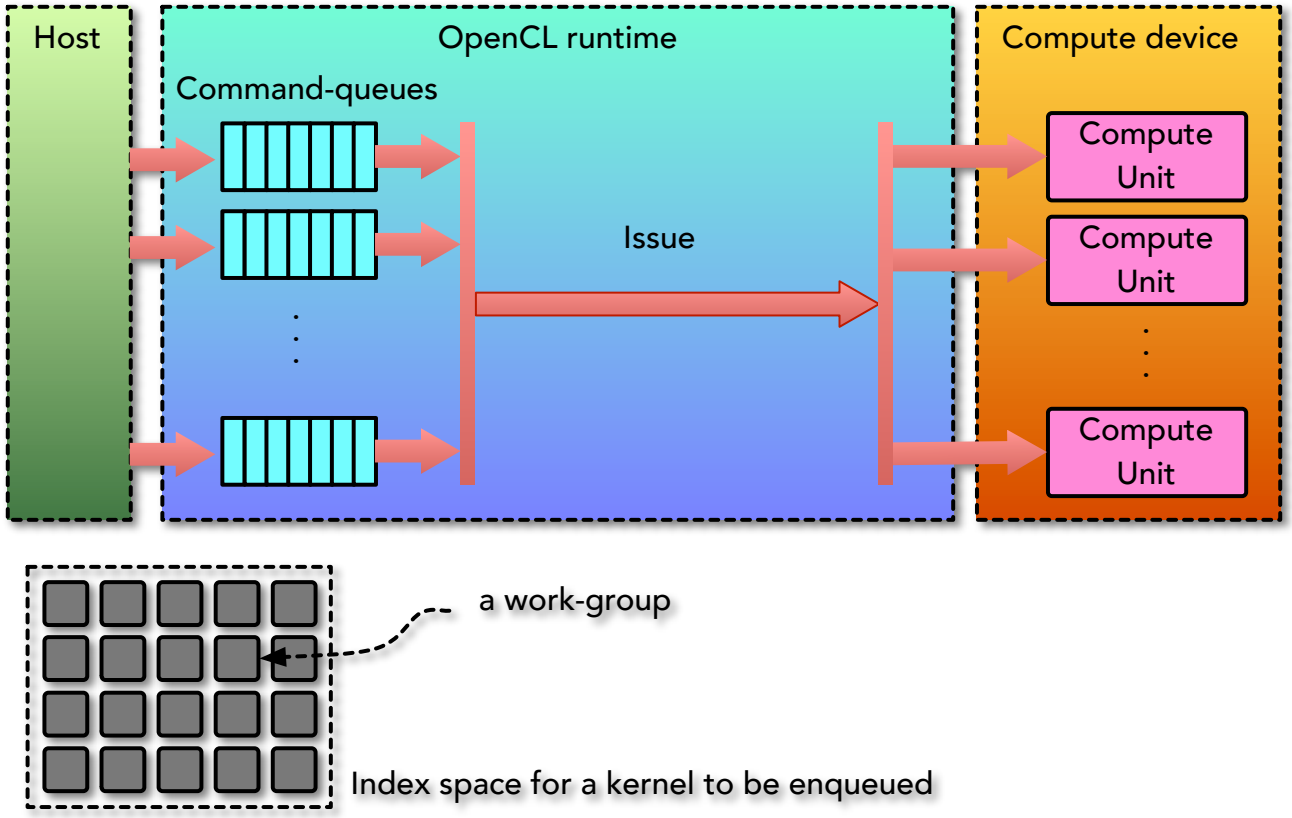
```
// Launch the kernel, note that command_queue is an in-order queue
   clEnqueueNDRangeKernel(command_queue, kernel, 2, NULL,
                          global, local, 0, NULL, NULL);

// Copy the result from bufferC to hostC
// A blocking read because of CL_TRUE
   clEnqueueReadBuffer(command_queue, bufferC, CL_TRUE, 0,
                       sizeC, hostC, 0, NULL, NULL);

// Print hostC
   ...

   return 0;
}
```

What the Host does for Kernel Execution



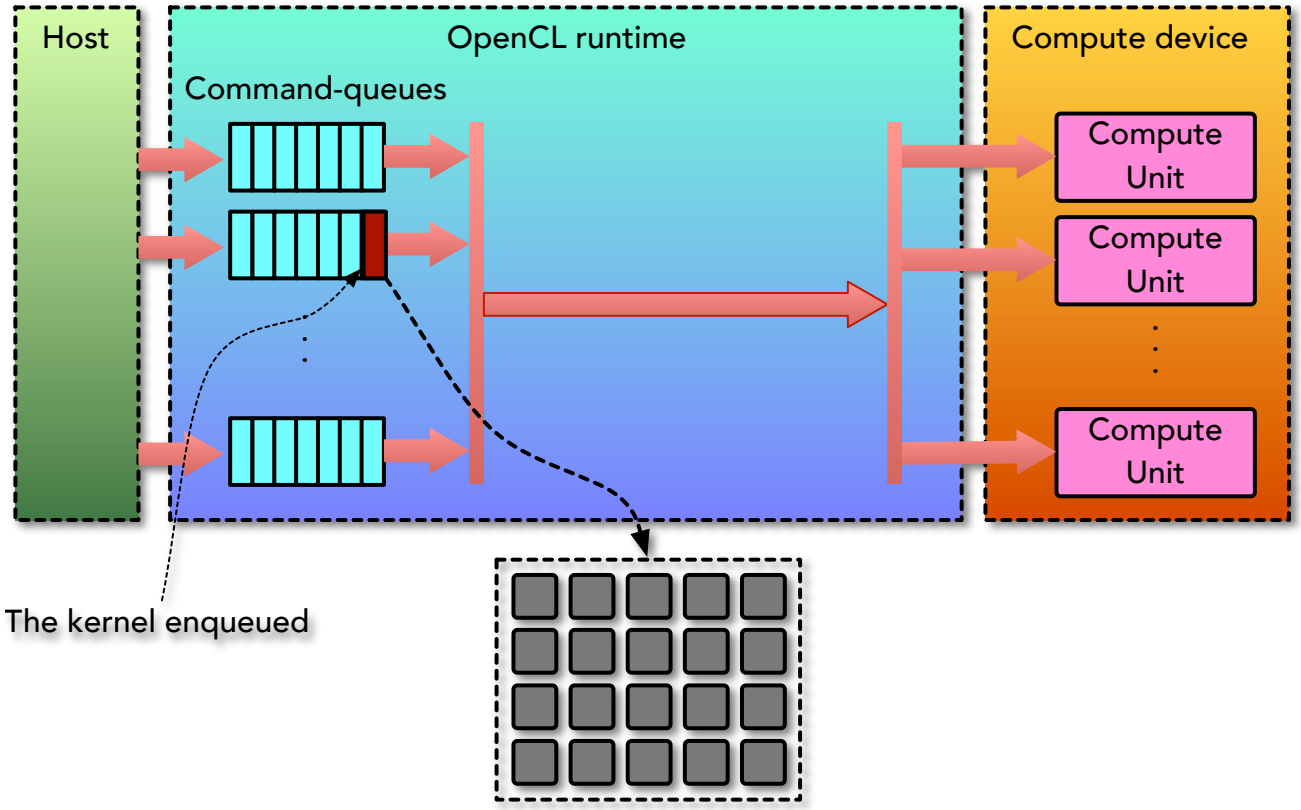
OpenCL Runtime

- The host program determines the index space of a kernel to be executed



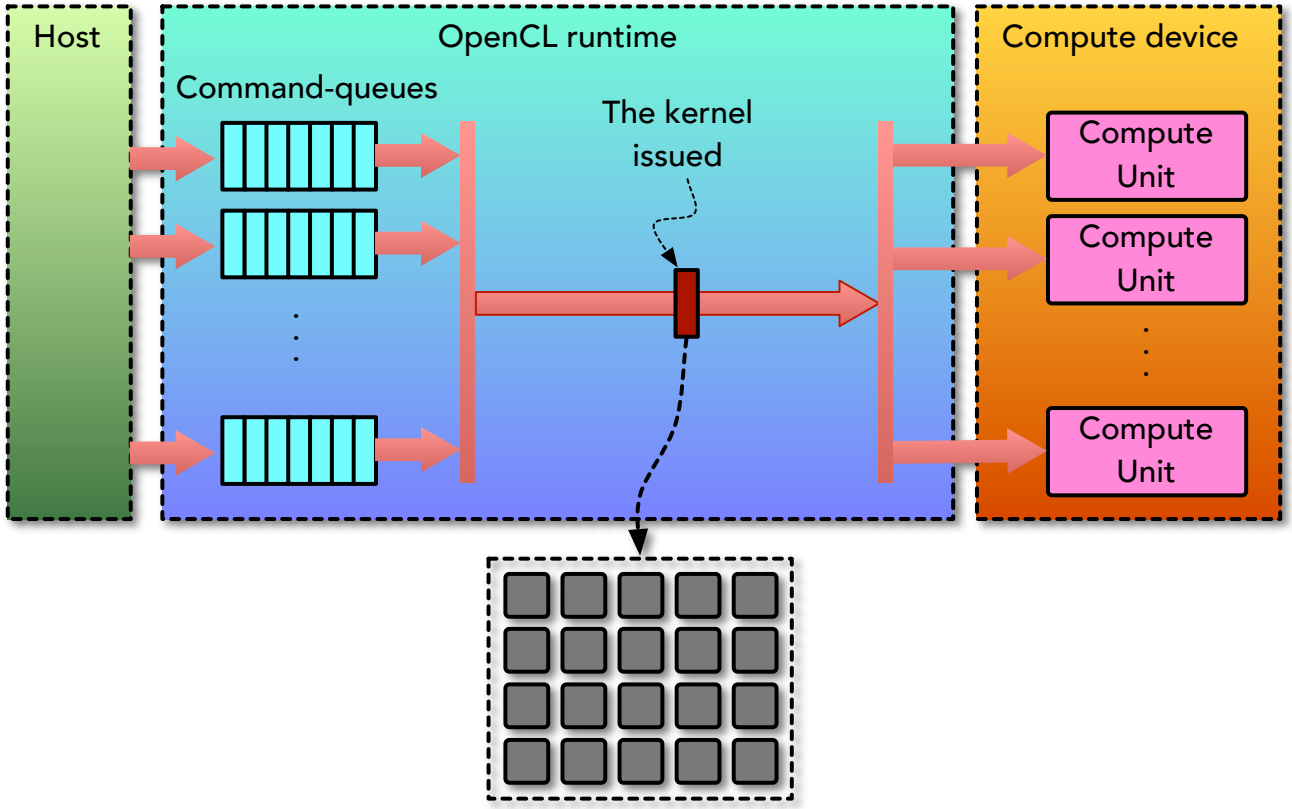
OpenCL Runtime (contd.)

- The host program inserts the kernel command to a command-queue



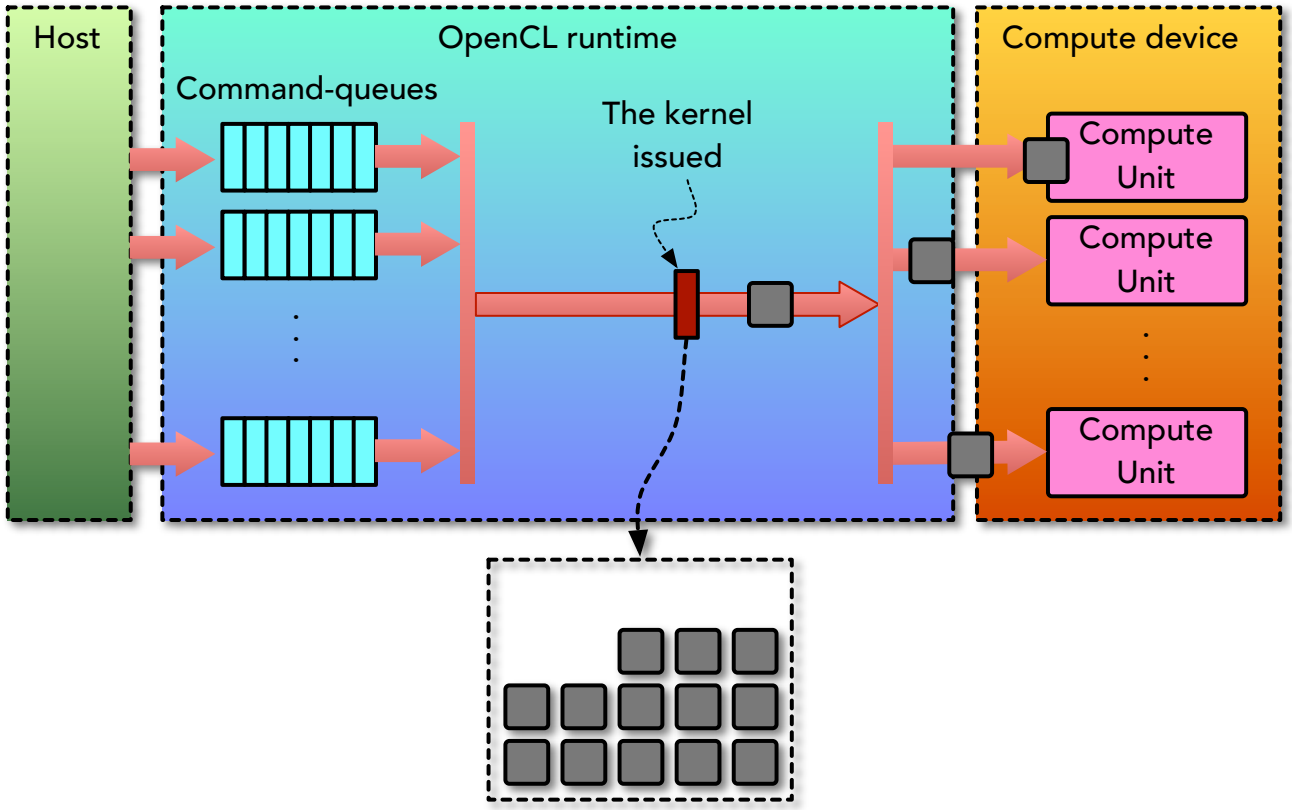
OpenCL Runtime (contd.)

- The runtime issues the command to the target device
  - In-order or out-of-order (depending on the queue type)
  - After resolving dependences between all enqueued commands



OpenCL Runtime (contd.)

- The runtime distributes the kernel workload to CUs in the target device
  - The granularity of the distribution: a work-group

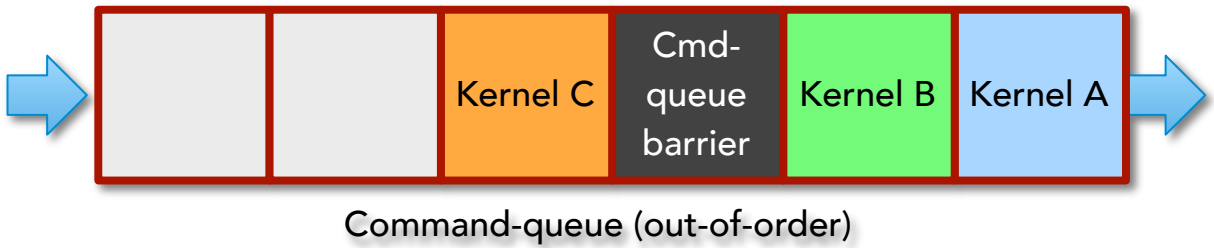


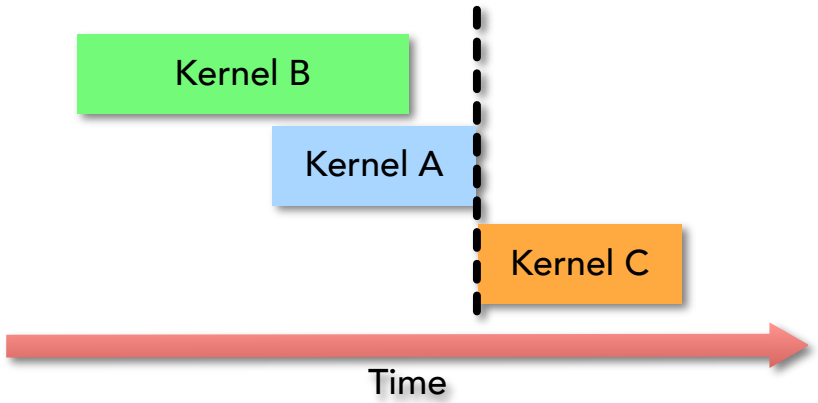
Work-group Barriers

- Synchronization only between work-items within a single work-group
  - The barrier must be encountered by all work-items of the work-group executing the kernel or by none at all
- No synchronization mechanism between different work-groups

Synchronization Between Commands

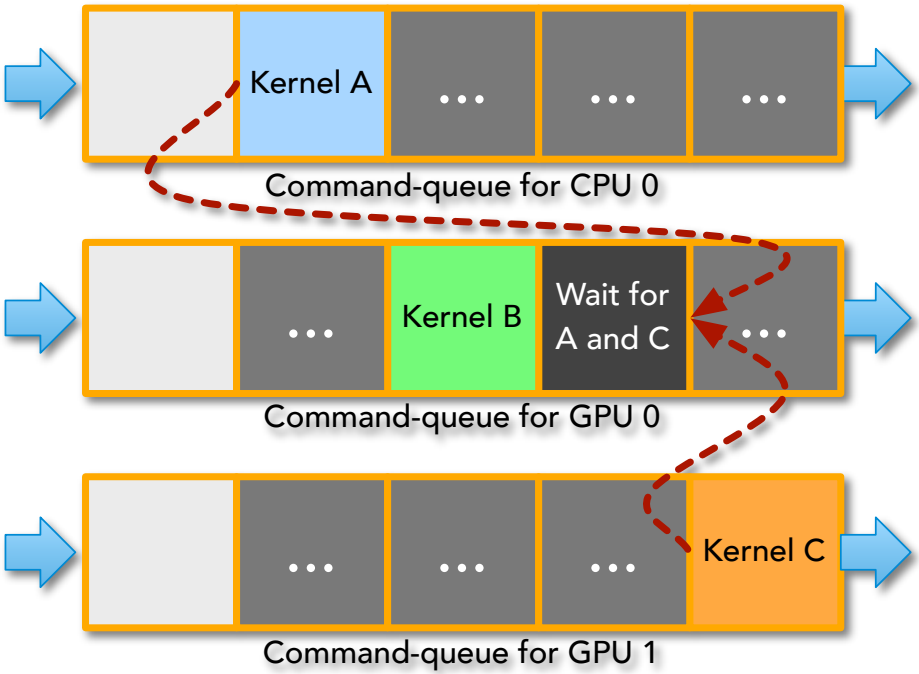
- Command-queue barrier
  - Between commands in ***a single command-queue***

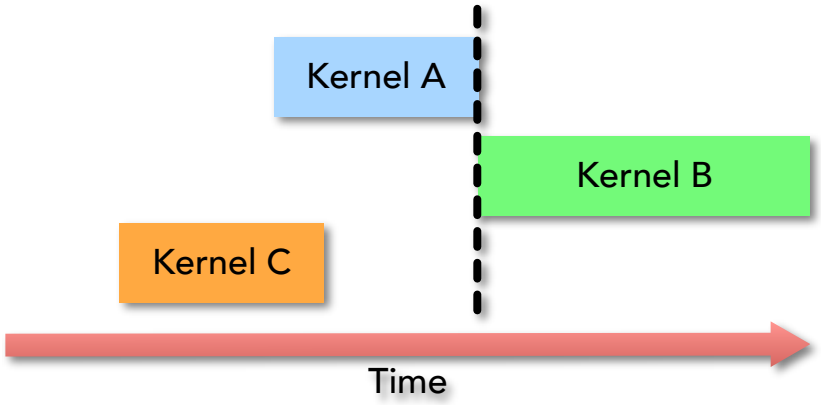




Synchronization Between Commands (contd.)

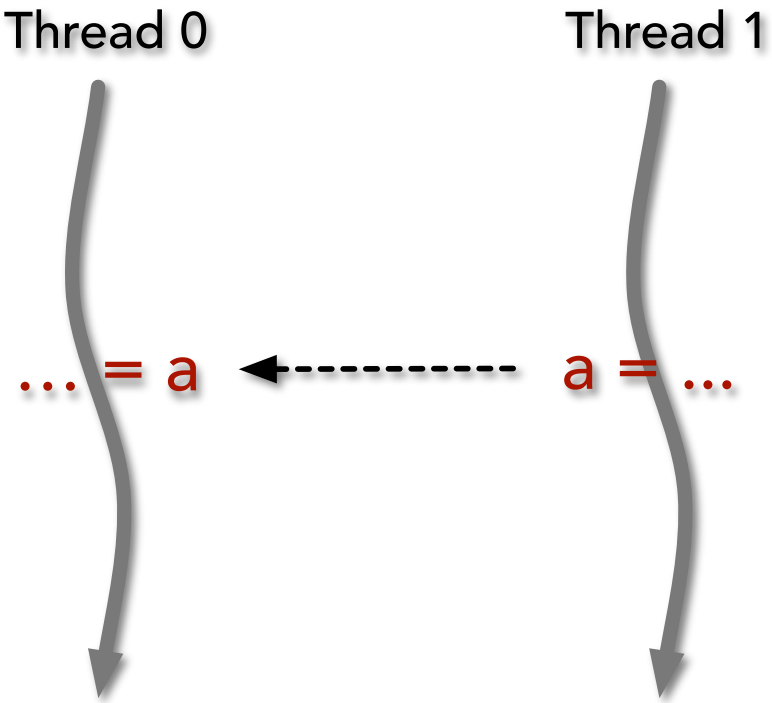
- All OpenCL API functions that enqueue commands return an event object
- Event synchronization
  - Between commands in different (possibly the same) command-queues





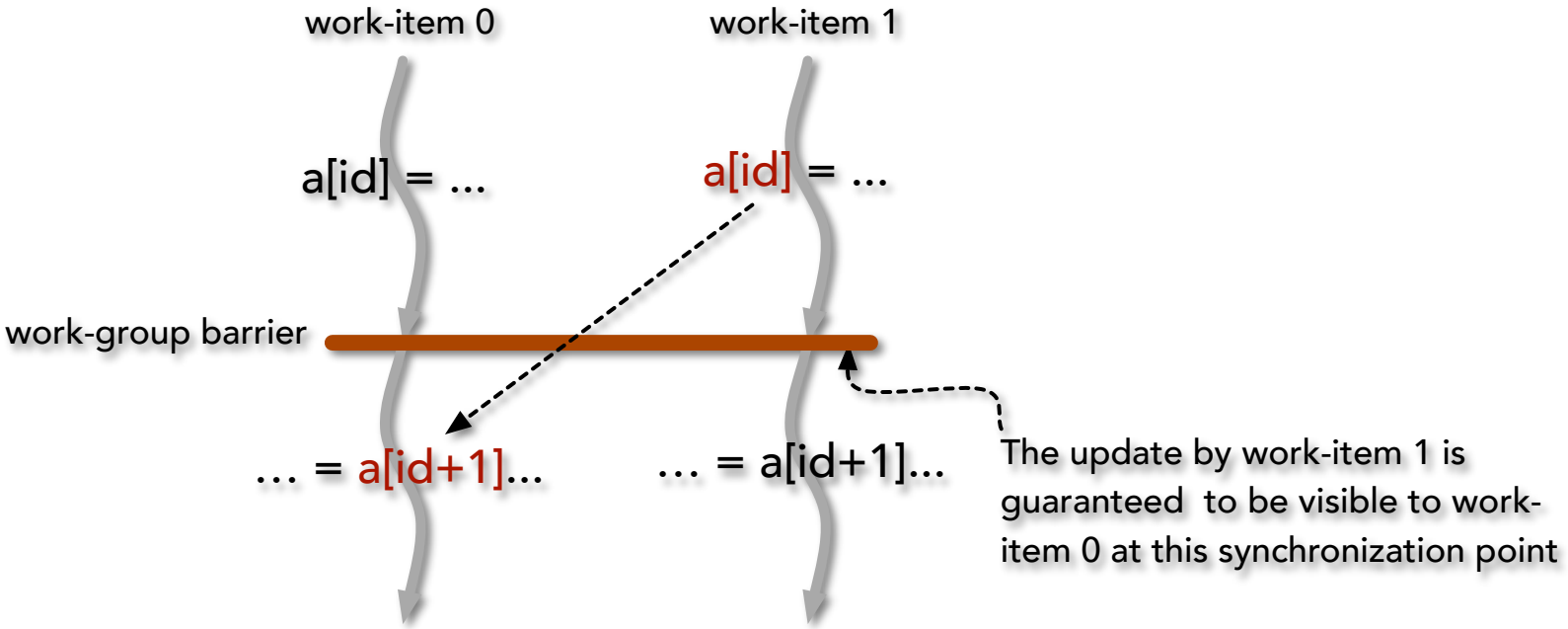
Memory Consistency Model

- When is the updates of memory by a thread visible to another thread?
- Memory access ordering



Memory Consistency Model (contd.)

- A relaxed memory consistency model
  - Memory is not consistent at all times
- Local/global memory is consistent across work-items in a single work-group at a work-group barrier
  - Not between different work-groups



Memory Consistency Model (contd.)

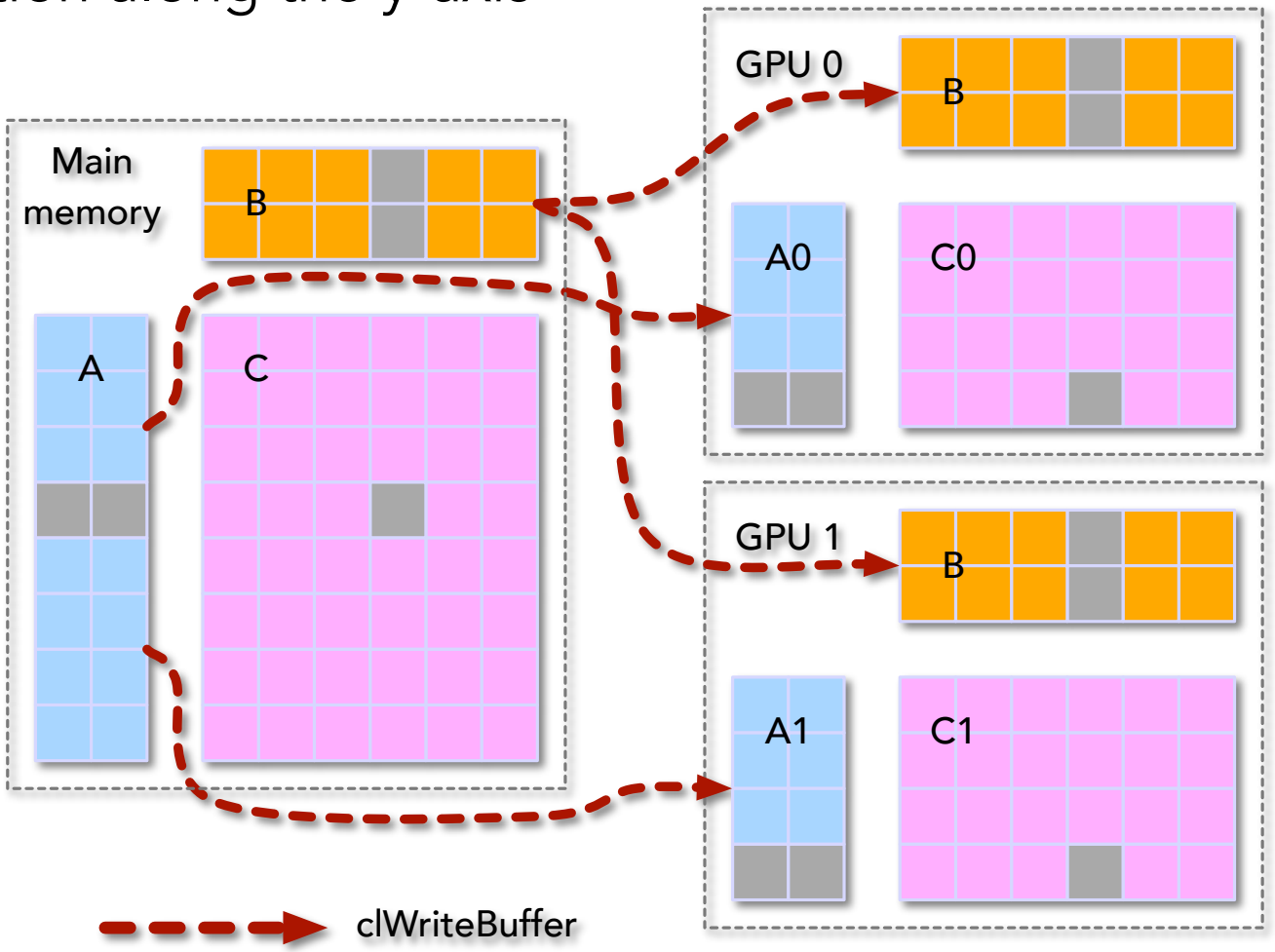
- Consistency for memory objects is enforced at a ***synchronization point***
  - clFinish
  - Work-group barriers
  - Command-queue barriers
  - Event synchronization

Part I: Introduction to OpenCL

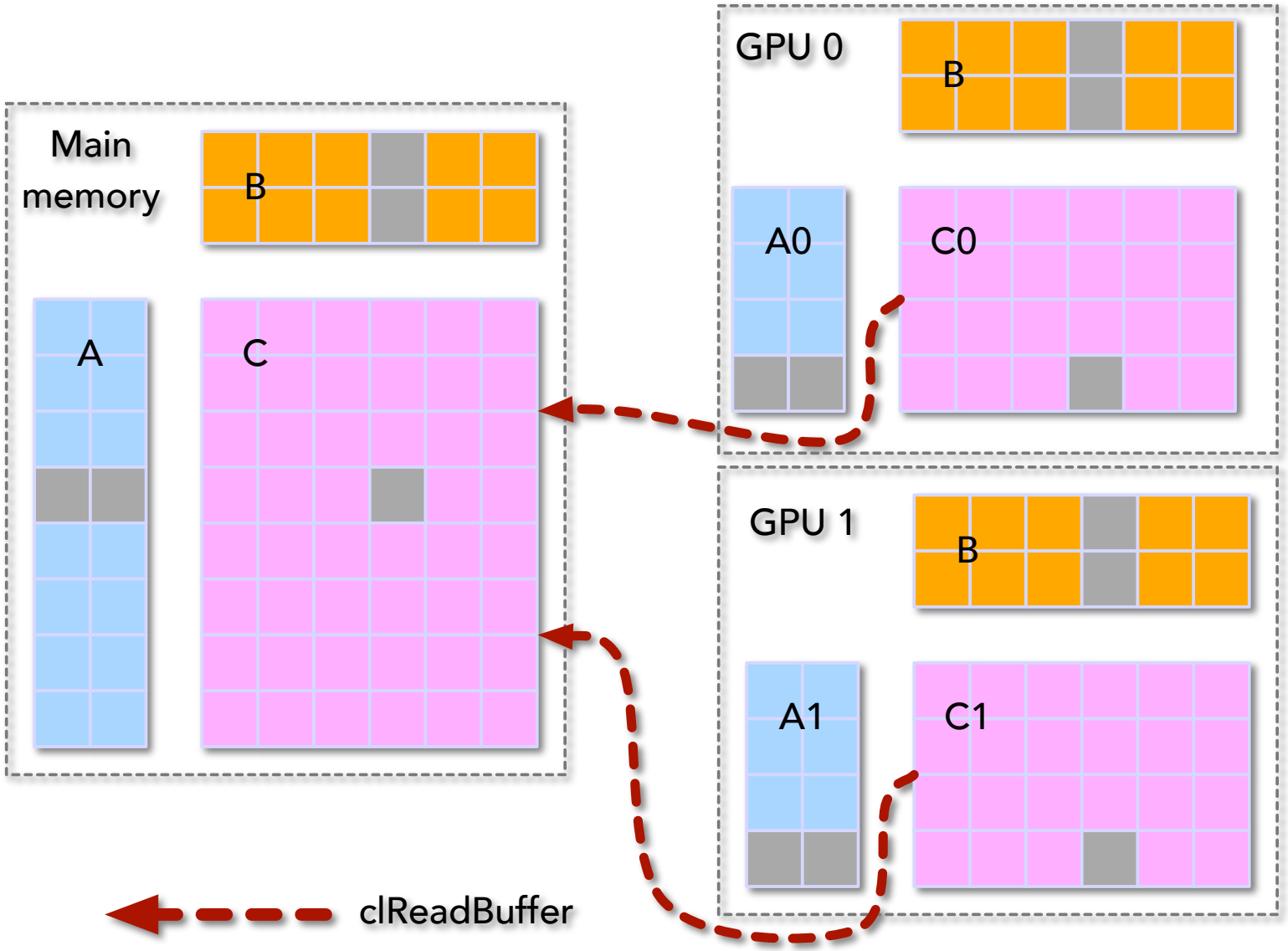
- Heterogeneous computing and OpenCL
- Background for parallel processing
- GPU architectures
- Intel Xeon Phi coprocessor
- Introduction to the OpenCL framework
- OpenCL for a single compute device
- OpenCL for multiple compute devices
- OpenCL+MPI for clusters
- HPL in OpenMP+MPI
- Limitations of the OpenCL programing model

Matrix Multiply for Multiple GPUs

- Partition the kernel index space into N regions for N GPUs
  - Partition along the y-axis



Matrix Multiply for Multiple GPUs (contd.)



Host Program for Multiple GPUs

- Assume that the system has four GPUs

```
...
#include <string.h>

// wA == hB, hC == hA, wC == wB
#define wA_SIZE  1024
#define hA_SIZE  1024
#define wB_SIZE  1024

#define MAX_DEV 4

const char* kernel_src = // Kernel source code
    "__kernel void matrix_mul(__global float* C,"
    "                         __global float* A,"
    ...
    "}";
```

Host Program for Multiple GPUs (contd.)

```
int main(int argc, char** argv)
{
    int  wA = wA_SIZE, hA = hA_SIZE;
    int  wB = wB_SIZE, hB = wA;
    int  wC = wB, hC = wA;

    size_t sizeA = wA * hA * sizeof(float);
    size_t sizeB = wB * hB * sizeof(float);
    size_t sizeC = wC * hC * sizeof(float);

// Allocate memory spaces to matrices hostA, hostB, and hostC
    float *hostA = (float*) malloc(sizeA);
    float *hostB = (float*) malloc(sizeB);
    float *hostC = (float*) malloc(sizeC);

// Initialize hostA and hostB
  ...
```

Host Program for Multiple GPUs (contd.)

```
// Obtain a list of available OpenCL platforms
    cl_platform_id platform;
    clGetPlatformIDs(1, &platform, NULL);


// Obtain the list of available devices on the OpenCL platform
    int ndev;
    cl_device_id device [MAX_DEV];
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 0, NULL,
                   (unsigned int *) &ndev);
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU,
                   ndev, device, NULL);

// Create an OpenCL context with the GPU devices
    cl_context context;
    context = clCreateContext(0, ndev, device, NULL, NULL, NULL);
```

Host Program for Multiple GPUs (contd.)

```
// Create command queues and attach it to the compute device
    cl_command_queue command_queue[MAX_DEV];
    for (i = 0; i < ndev; i++)
        command_queue = clCreateCommandQueue(context, device[i],
                                             0, NULL);

// Allocate buffer memory objects
    cl_mem bufferA[MAX_DEV];
    cl_mem bufferB[MAX_DEV];
    cl_mem bufferC[MAX_DEV];
    for (i = 0; i < ndev; i++) {
        bufferA[i] = clCreateBuffer(context, CL_MEM_READ_ONLY,
                                    sizeA/ndev, NULL, NULL);
        bufferB[i] = clCreateBuffer(context, CL_MEM_READ_ONLY,
                                    sizeB, NULL, NULL);
        bufferC[i] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                                    sizeC/ndev, NULL, NULL);
  }
```

Host Program for Multiple GPUs (contd.)

```
// Create an OpenCL program object for the context
// and load the kernel source into the program object
    size_t kernel_src_len = strlen(kernel_src);

    cl_program program;
    program = clCreateProgramWithSource(context, 1,
                                        (const char**) &kernel_src,
                                        &kernel_src_len, NULL);

// Build (compile and link) the program executable
// from the source or binary for the device
    clBuildProgram(program, ndev, device, NULL, NULL, NULL);

// Create kernel objects from the program
    cl_kernel kernel[MAX_DEV];
    for (i = 0; i < ndev; i++) {
        kernel[i] = clCreateKernel(program, "matrix_mul", NULL);
    }
```

Host Program for Multiple GPUs (contd.)

```
// Set the arguments of the kernel objects
    for (i = 0; i < ndev; i++) {
        clSetKernelArg(kernel[i], 0, sizeof(cl_mem),
                       (void*) &bufferC[i]);
        clSetKernelArg(kernel[i], 1, sizeof(cl_mem),
                       (void*) &bufferA[i]);
        clSetKernelArg(kernel[i], 2, sizeof(cl_mem),
                       (void*) &bufferB[i]);
        clSetKernelArg(kernel[i], 3, sizeof(cl_int),
                       (void*) &wA);
        clSetKernelArg(kernel[i], 4, sizeof(cl_int),
                       (void*) &wB);
    }
```

Host Program for Multiple GPUs (contd.)

```
// Copy the input matrices to the corresponding buffers
    for (i = 0; i < ndev; i++) {
        clEnqueueWriteBuffer(command_queue[i], bufferA[i],
                        CL_FALSE, 0, sizeA/ndev,
                        void*) ((size_t) hostA + (sizeA/ndev)*i),
                        0, NULL, NULL);

        clEnqueueWriteBuffer(command_queue[i], bufferB[i],
                        CL_FALSE, 0, sizeB, hostB, 0, NULL, NULL);
    }
```

Host Program for Multiple GPUs (contd.)

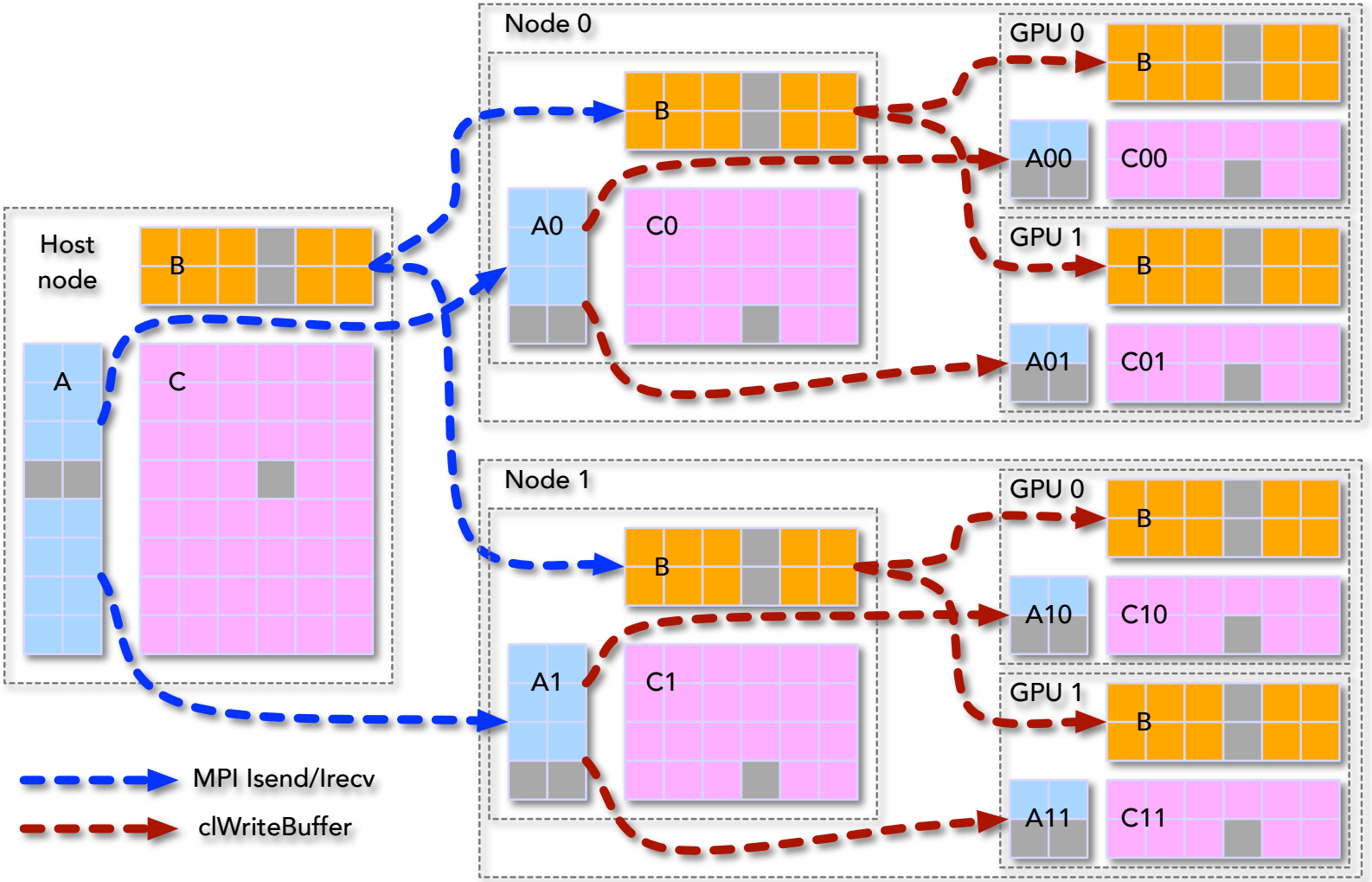
```
// Set up the kernel index space
    size_t global[2] = { wC, hC/ndev };
    size_t local[2] = { 16, 16 };

// Launch the kernel
    for (i = 0; i < ndev; i++)
        clEnqueueNDRangeKernel(command_queue[i], kernel[i], 2, NULL,
                               global, local, 0, NULL, NULL);

// Copy the result from bufferC to hostC
    for (i = 0; i < ndev; i++)
        clEnqueueReadBuffer(command_queue[i], bufferC[i],
                            CL_TRUE, 0, sizeC/ndev,
                            (void*) ((size_t) hostC+(sizeC/ndev)*i),
                            0, NULL, NULL);
// Print hostC
  ...
  return 0;
}
```

Part I: Introduction to OpenCL

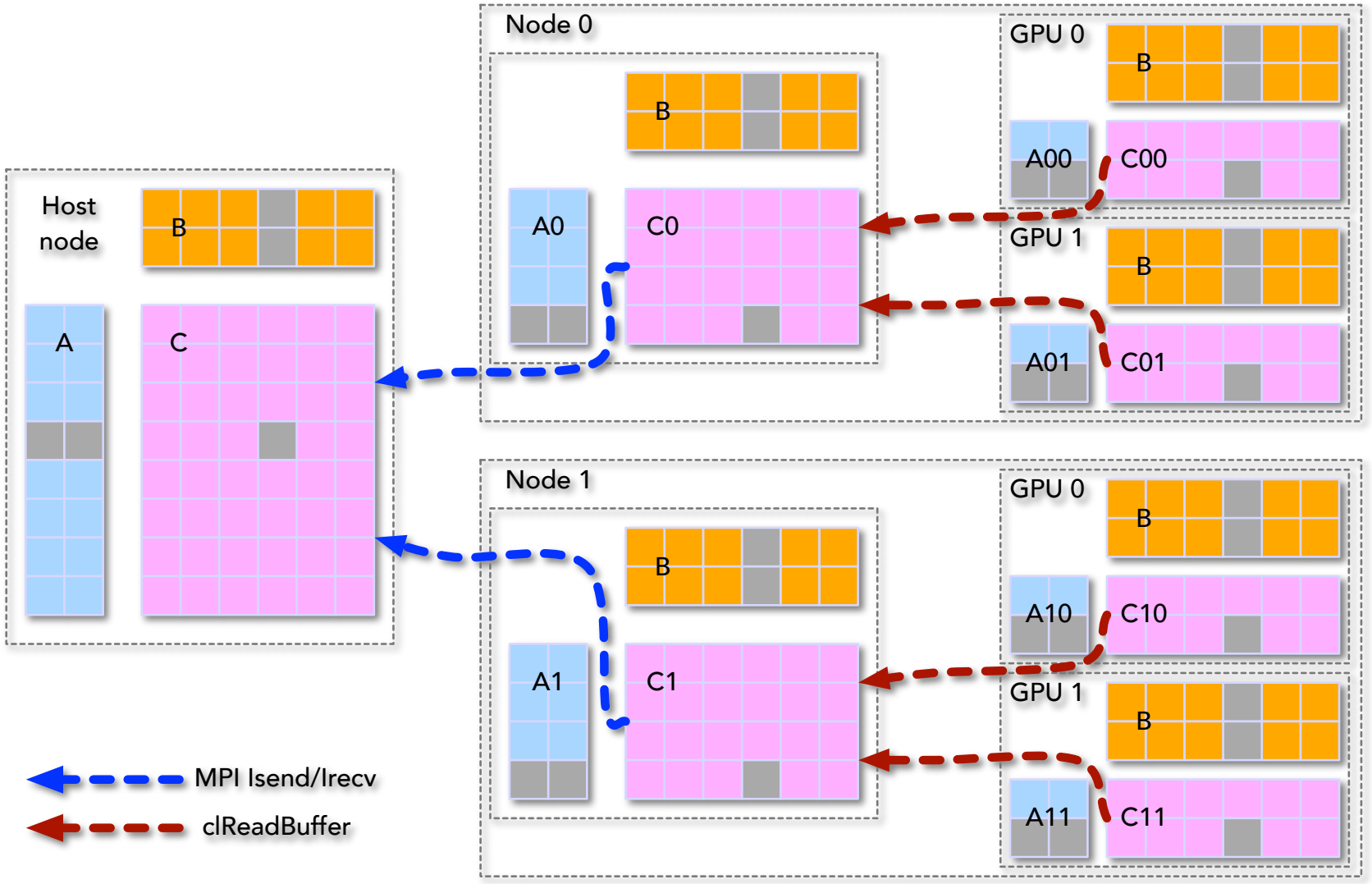
- Heterogeneous computing and OpenCL
- Background for parallel processing
- GPU architectures
- Intel Xeon Phi coprocessor
- Introduction to the OpenCL framework
- OpenCL for a single compute device
- OpenCL for multiple compute devices
- OpenCL+MPI for clusters
- HPL in OpenMP+MPI
- Limitations of the OpenCL programing model

Matrix Multiply for the Cluster

- Hierarchical Partitioning



Matrix Multiply for the Cluster (contd.)



Buffer Writes in OpenCL+MPI

```
if (rank == 0) {
    for (i = 1; i < nnode; i++) {
        MPI_Isend((void*) ((size_t) hostA+(sizeA/nnode) * i),
                  (int) sizeA / nnode, MPI_CHAR, i, 0,
                  MPI_COMM_WORLD, &request[0]);
        MPI_Isend(hostB, (int) sizeB, MPI_CHAR, i, 0,
                  MPI_COMM_WORLD, &request[1]);
    }
} else {
    MPI_Irecv(hostA, (int) sizeA / nnode, MPI_CHAR, 0, 0,
                  MPI_COMM_WORLD, &request[0]);
    MPI_Irecv(hostB, (int) sizeB, MPI_CHAR, 0, 0,
                  MPI_COMM_WORLD, &request[1]);
}

MPI_Waitall(2, request, status);
```

Buffer Writes in OpenCL+MPI (contd.)

```
for (i = 0; i < ndev; i++) {
    clEnqueueWriteBuffer(command_queue[i], bufferA[i], CL_FALSE, 0,
                         sizeA/nnode/ndev,
                         (void*) ((size_t) hostA+(sizeA/nnode/ndev)*i),
                         0, NULL, NULL);

    clEnqueueWriteBuffer(command_queue[i], bufferB[i], CL_FALSE, 0,
                         sizeB, B, 0, NULL, NULL);
}
```

Buffer Reads in OpenCL+MPI

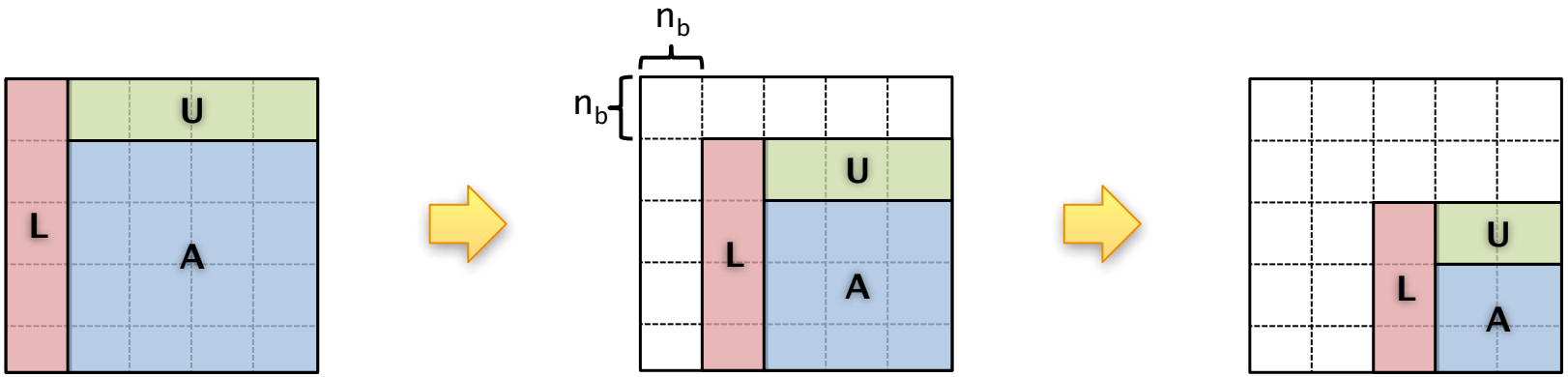
```
for (i = 0; i < ndev; i++) {
    clEnqueueReadBuffer(command_queue[i], bufferC[i], CL_TRUE, 0,
                sizeC/nnode/ndev,
                (void*) ((size_t) hostC + (sizeC/nnode/ndev) * i),
                0, NULL, NULL);
}

if (rank == 0) {
    for (i = 1; i < nnode; i++) {
    MPI_Irecv((void*) ((size_t) hostC+(sizeC/nnode) * i),
                        (int) sizeC/nnode, MPI_CHAR,
                        i, 0, MPI_COMM_WORLD, &request[0]);
    }
} else {
    MPI_Isend(host C, (int) sizeC/nnode, MPI_CHAR, 0, 0,
                MPI_COMM_WORLD, &request[0]);
}

MPI_Wait(request, status);
```

Part I: Introduction to OpenCL

- Heterogeneous computing and OpenCL
- Background for parallel processing
- GPU architectures
- Intel Xeon Phi coprocessor
- Introduction to the OpenCL framework
- OpenCL for a single compute device
- OpenCL for multiple compute devices
- OpenCL+MPI for clusters
- HPL in OpenMP+MPI
- Limitations of the OpenCL programing model

High Performance LINPACK (HPL)

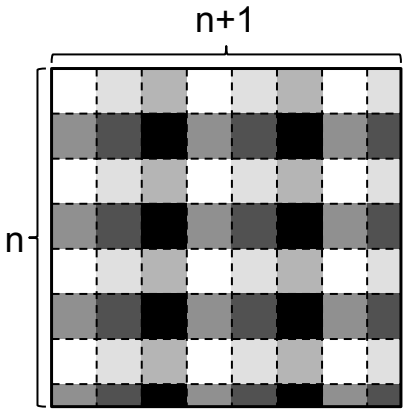
- Solving a dense linear system of order n
  - n×(n+1) coefficient matrix
- Using the blocked LU decomposition algorithm
  - The matrix is logically divided into blocks with a size of $n_b \times n_b$
  - Each iteration performs the following three steps:
    - **Factorizatio**n: the LU factorization is performed on L
    - **Swap**: Rows in U and A are swapped
    - **Update**: U and A are updated using L

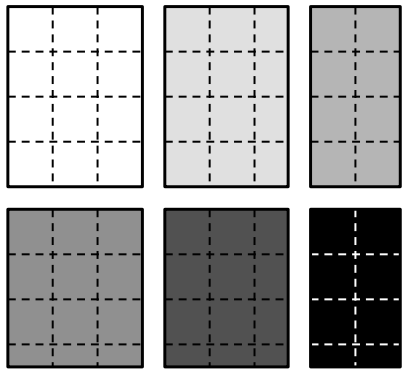


HPL on Heterogeneous Clusters

- Nodes are arranged as a two-dimensional grid
- The blocks of the matrix are cyclically distributed across the nodes

- Most of the floating-point operations in HPL are from the **update** phase
  - DGEMM and DTRSM routines in BLAS
  - E.g., with n = 200,000 and n_b = 2,000
    - Factorization: 0.5%, DTRSM: 0.5%, DGEMM: 99%



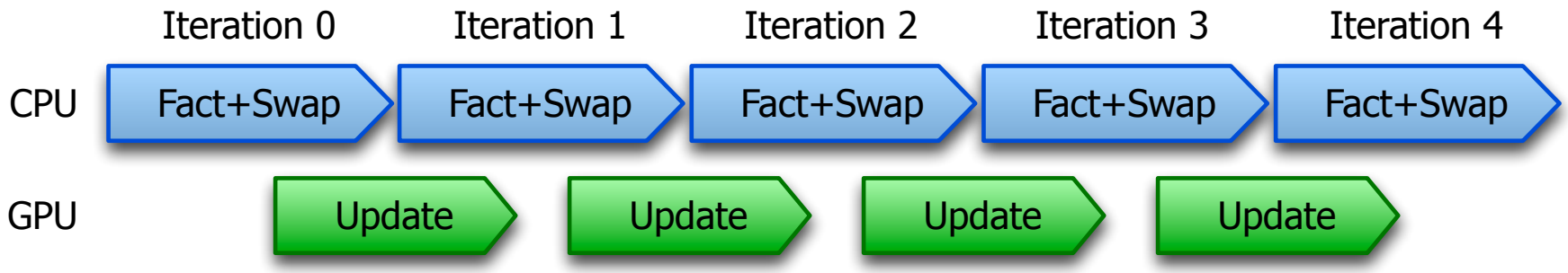


OpenCL + MPI version of HPL

- Multi-GPU versions of DGEMM and DTRSM in OpenCL
  - Kernels access image objects (for speed)
- Load balancing between GPUs
  - Dynamically assigning blocks in the matrix to the GPUs

OpenCL + MPI version of HPL

- Overlap the computation of GPUs with the computation/ communication of CPUs
- CPUs become a performance bottleneck
  - The performance gap between CPUs and GPUs is big
- Optimize the CPU side
  - Overlapping computation of CPUs and communication between nodes
  - Exploit idle CPUs in some nodes



Performance of the HPL in OpenCL+MPI

- Chundoong
  - 56-node heterogeneous CPU/GPU cluster at Seoul National University
    - 16 CPU cores and 4 GPUs per node
- Performance: 106.8 TFLOPS
  - #277 in Top500 (November 2012)
- Power efficiency: 1870 MFLOPS/Watts
  - #32 in Green500 (November 2012)

Part I: Introduction to OpenCL

- Heterogeneous computing and OpenCL
- Background for parallel processing
- GPU architectures
- Intel Xeon Phi coprocessor
- Introduction to the OpenCL framework
- OpenCL for a single compute device
- OpenCL for multiple compute devices
- OpenCL+MPI for clusters
- HPL in OpenMP+MPI
- Limitations of the OpenCL programing model

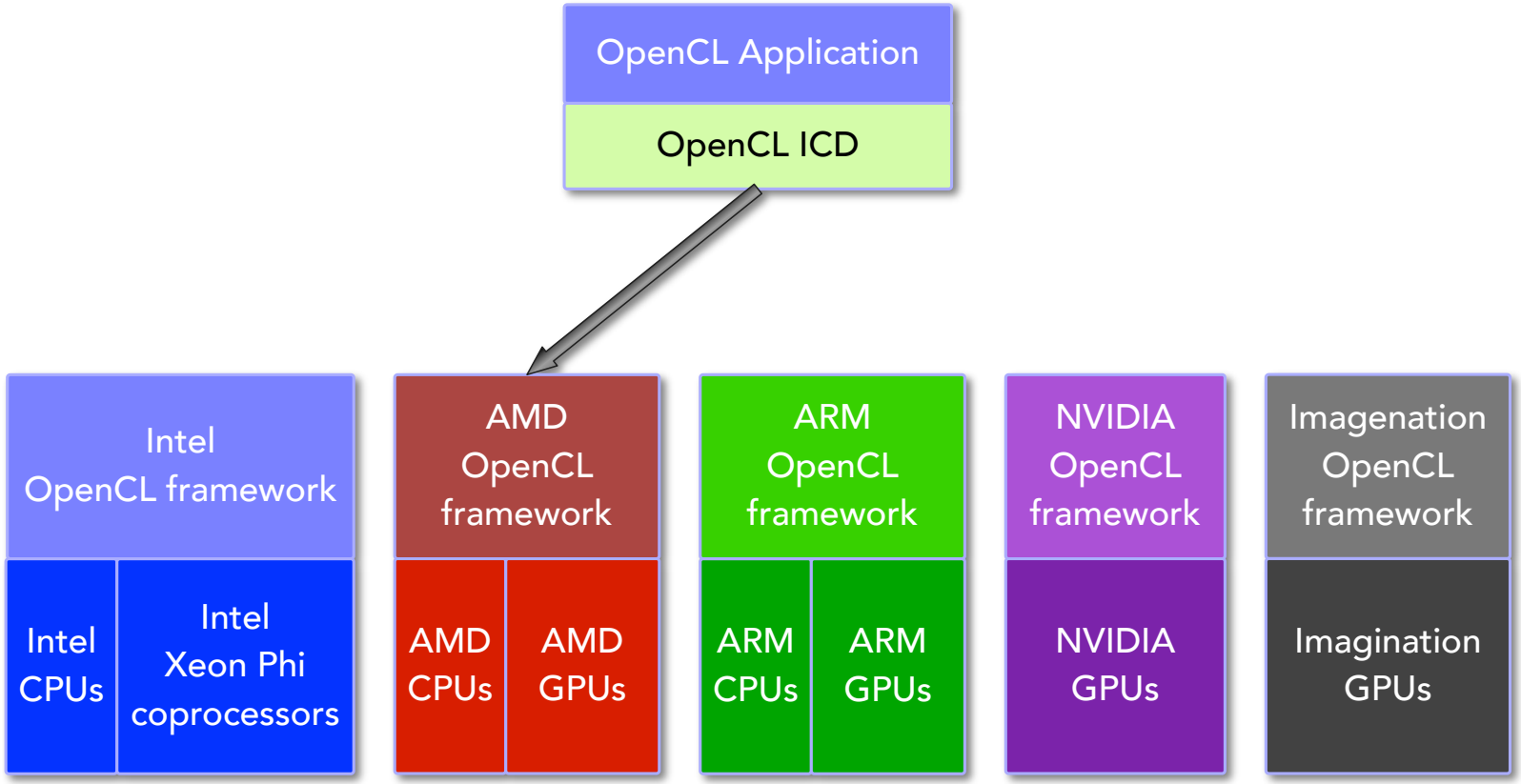
Limitations

- Current OpenCL implementations are targeting parallelism for multiple compute devices under a single operating system instance

- An application for a heterogeneous cluster
  - MPI + OpenCL or MPI + CUDA
  - Complicated, less portable, and hard to maintain

OpenCL ICD

- 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

Part II

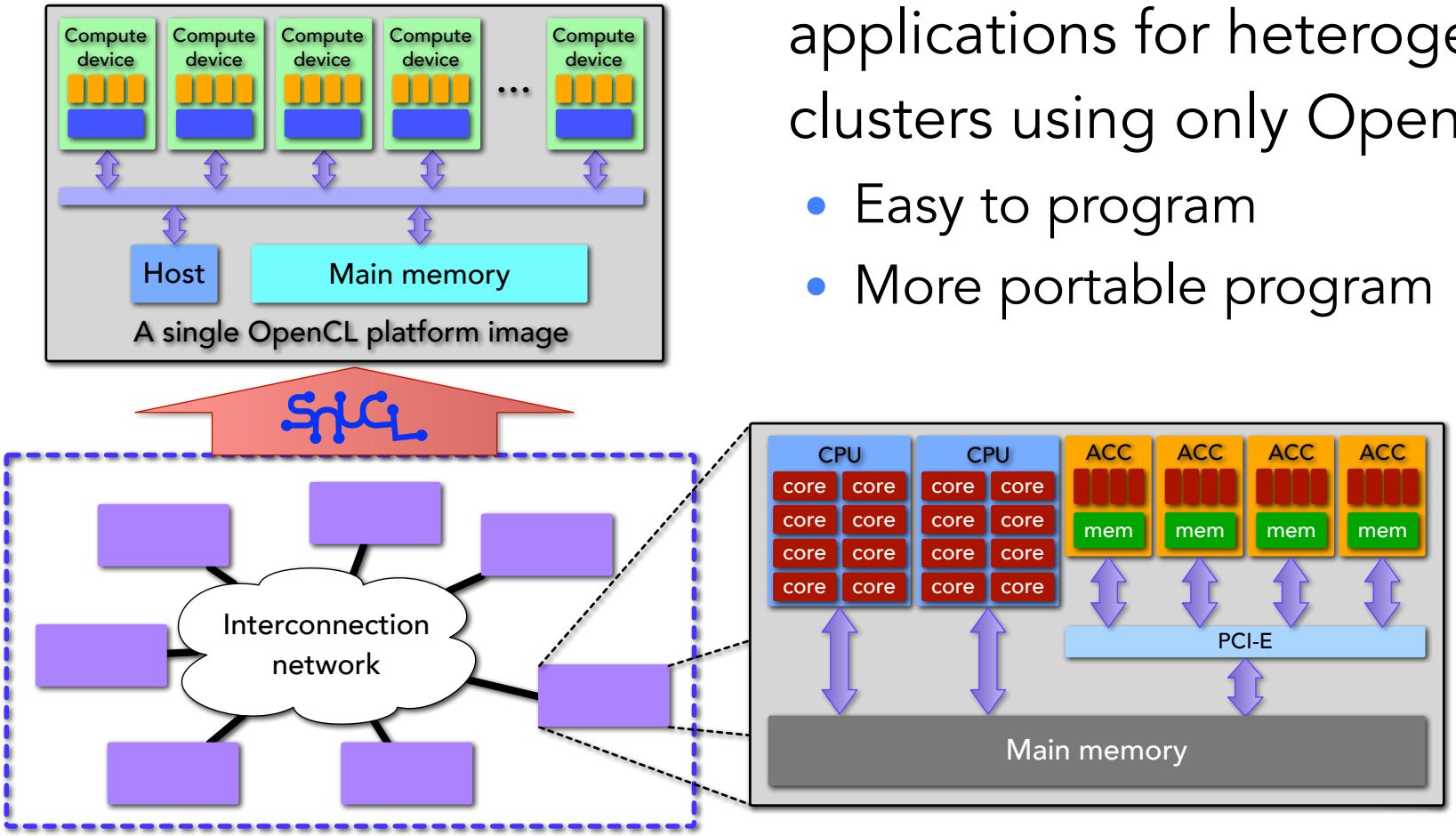
Introduction to SnuCL

Part II: Introduction to SnuCL

- Illusion of a single OpenCL platform Image
- Collective communication extensions to OpenCL
- SnuCL runtime
- How to write SnuCL applications
- SnuCL benchmark applications
- SnuCL Performance evaluation
- Future directions

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

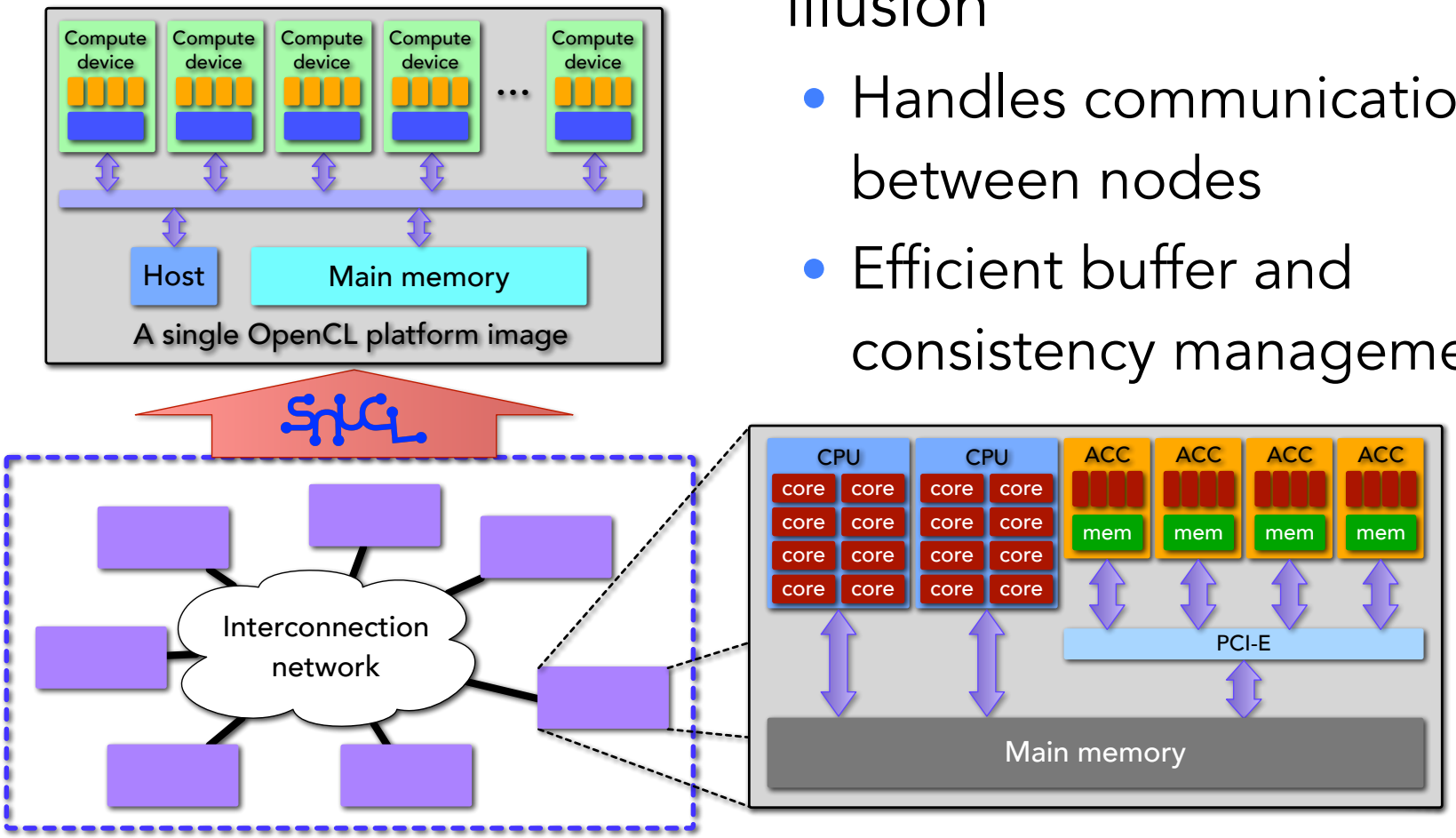


SnuCL

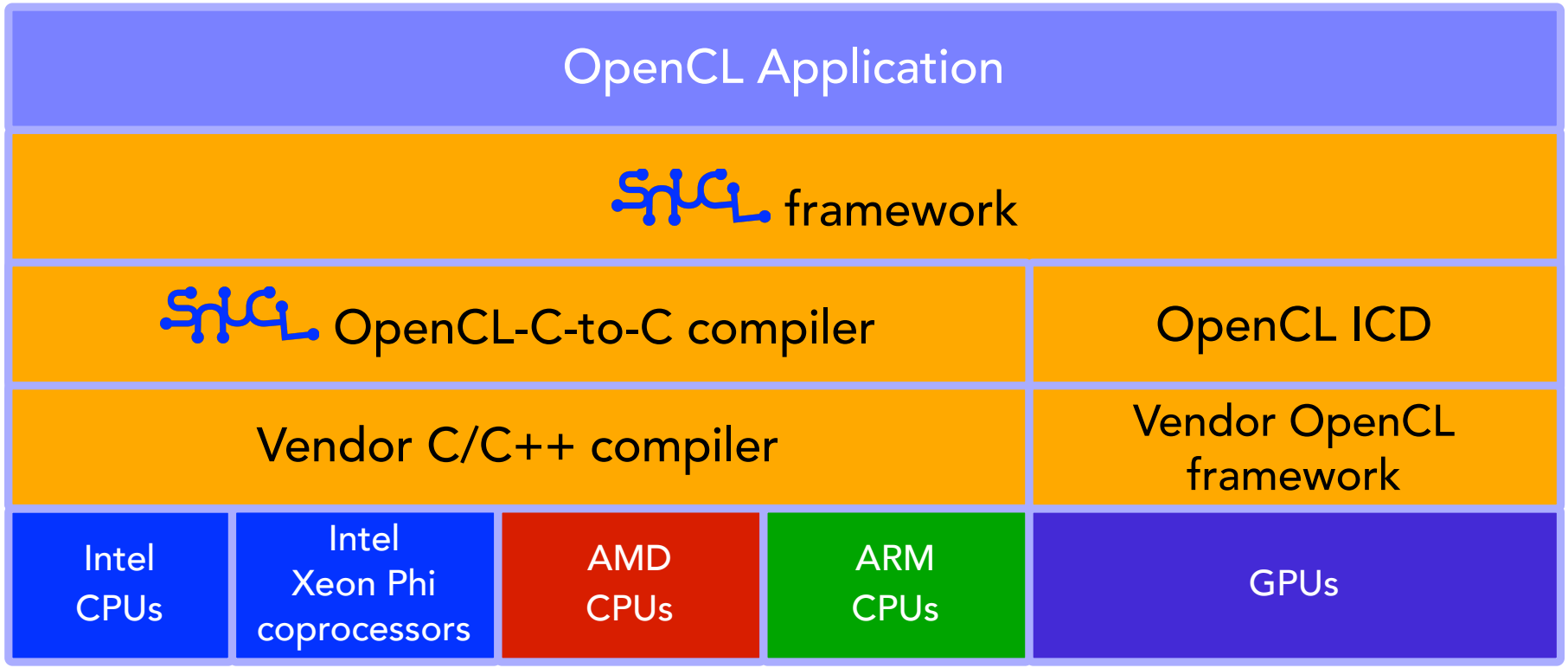
- An OpenCL framework [ICS'12]
  - Platform layer + runtime + kernel compiler
- Freely available, open-source software developed at Seoul National University
  - http://aces.snu.ac.kr
  - Supports OpenCL 1.2
  - Passed most of OpenCL conformance tests
- Supports x86 CPUs, ARM CPUs, AMD GPUs, NVIDIA GPUs, 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



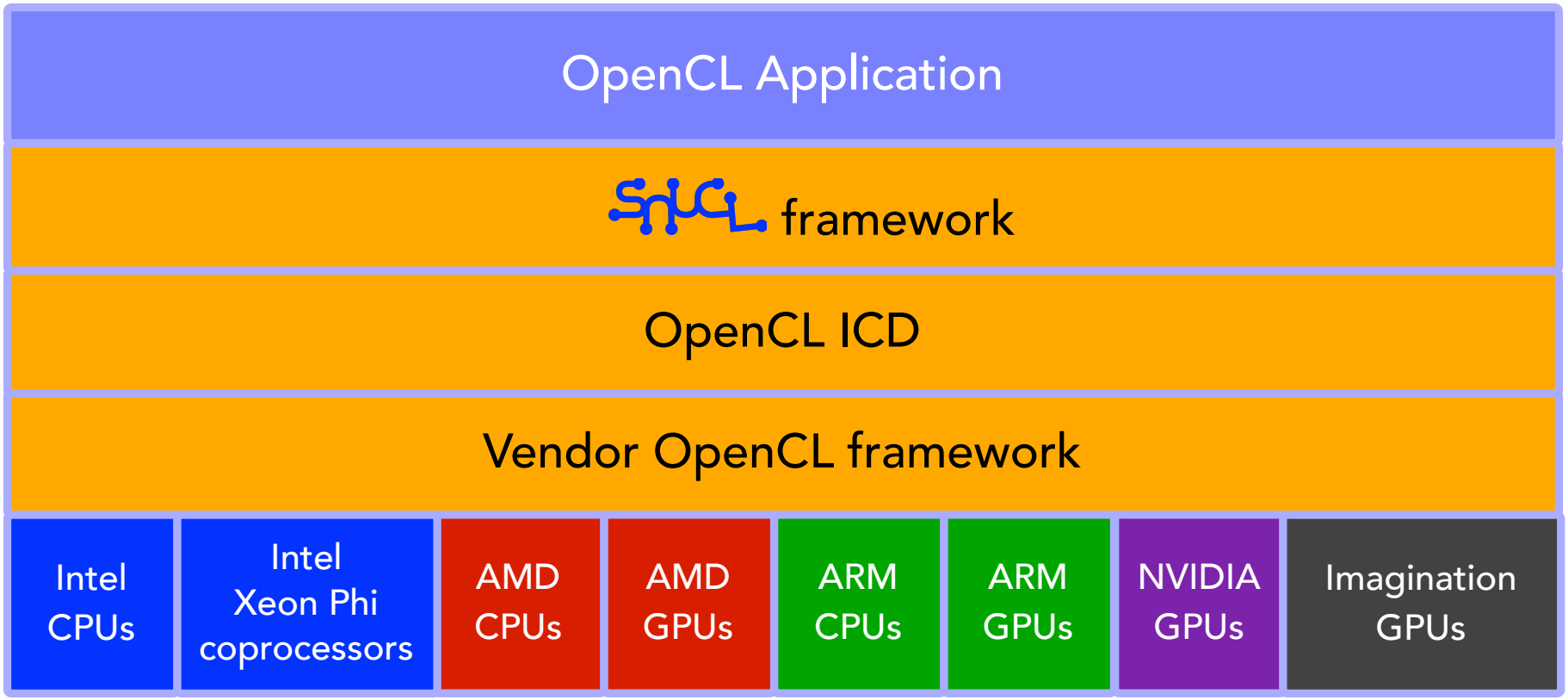
Two Ways of Using SnuCL



Emulating PEs CPU and Xeon Phi Devices

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

Two Ways of Using SnuCL (contd.)



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

ICD Dispatch Tables

- Every ICD compatible OpenCL implementation has an ICD dispatch table
  - Contains all OpenCL API function pointers
- SnuCL keeps the ICD dispatch tables in all available OpenCL frameworks in the system
- Use the tables to direct calls to a particular vendor implementation

```
struct KHRicdVendorDispatchRec
{
  KHRpfn_clGetPlatformIDs   clGetPlatformIDs;
  KHRpfn_clGetPlatformInfo  clGetPlatformInfo;
  KHRpfn_clGetDeviceIDs     clGetDeviceIDs;
  KHRpfn_clGetDeviceInfo    clGetDeviceInfo;
  KHRpfn_clCreateContext    clCreateContext;
  ...
};
```

Effect of Using SnuCL

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

Previous approach (OpenCL + MPI)	SnuCL (OpenCL only)
… cl_mem bufferA = clCreateBuffer(…); cl_mem bufferB = clCreateBuffer(…); … **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, …);** **}** …	… cl_mem bufferA = clCreateBuffer(…); cl_mem bufferB = clCreateBuffer(…); … **clEnqueueCopyBuffer(cq, bufferA, bufferB, …);** …

Part II: Introduction to SnuCL

- Illusion of a single OpenCL platform Image
- Collective communication extensions to OpenCL
- SnuCL runtime
- How to write SnuCL applications
- SnuCL benchmark applications
- SnuCL Performance evaluation
- Future directions

SnuCL Extensions to OpenCL

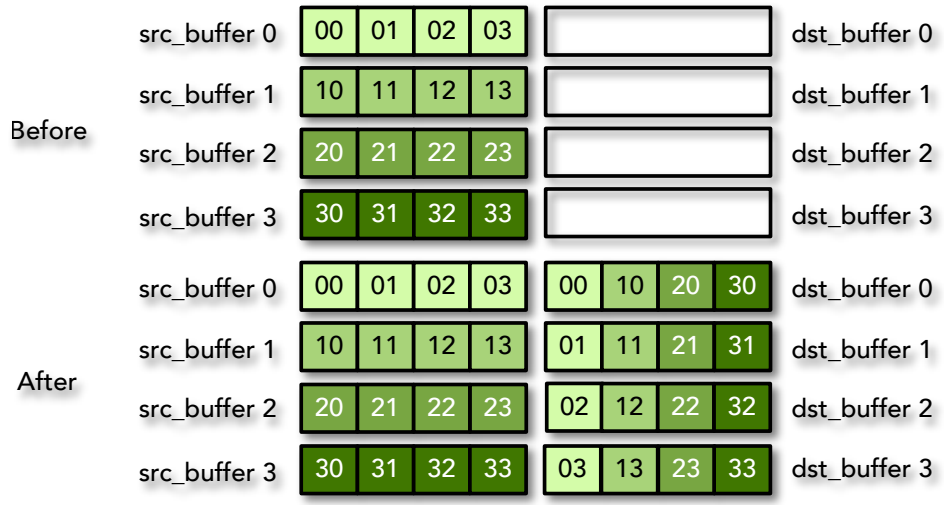
- SnuCL has extensions to OpenCL for copying buffers
  - Similar to MPI collective communication operations

SnuCL	MPI Equivalent
clEnqueueAlltoAllBuffer	MPI_Alltoall
clEnqueueBroadcastBuffer	MPI_Bcast
clEnqueueScatterBuffer	MPI_Scatter
clEnqueueGatherBuffer	MPI_Gather
clEnqueueAllGatherBuffer	MPI_Allgather
clEnqueueReduceBuffer	MPI_Reduce
clEnqueueAllReduceBuffer	MPI_Allreduce
clEnqueueReduceScatterBuffer	MPI_Reduce_scatter
clEnqueueScanBuffer	MPI_Scan

SnuCL Extensions to OpenCL (contd.)

- The cmd_queue_list specifies target compute devices
  - The destination buffers reside in the target compute devices

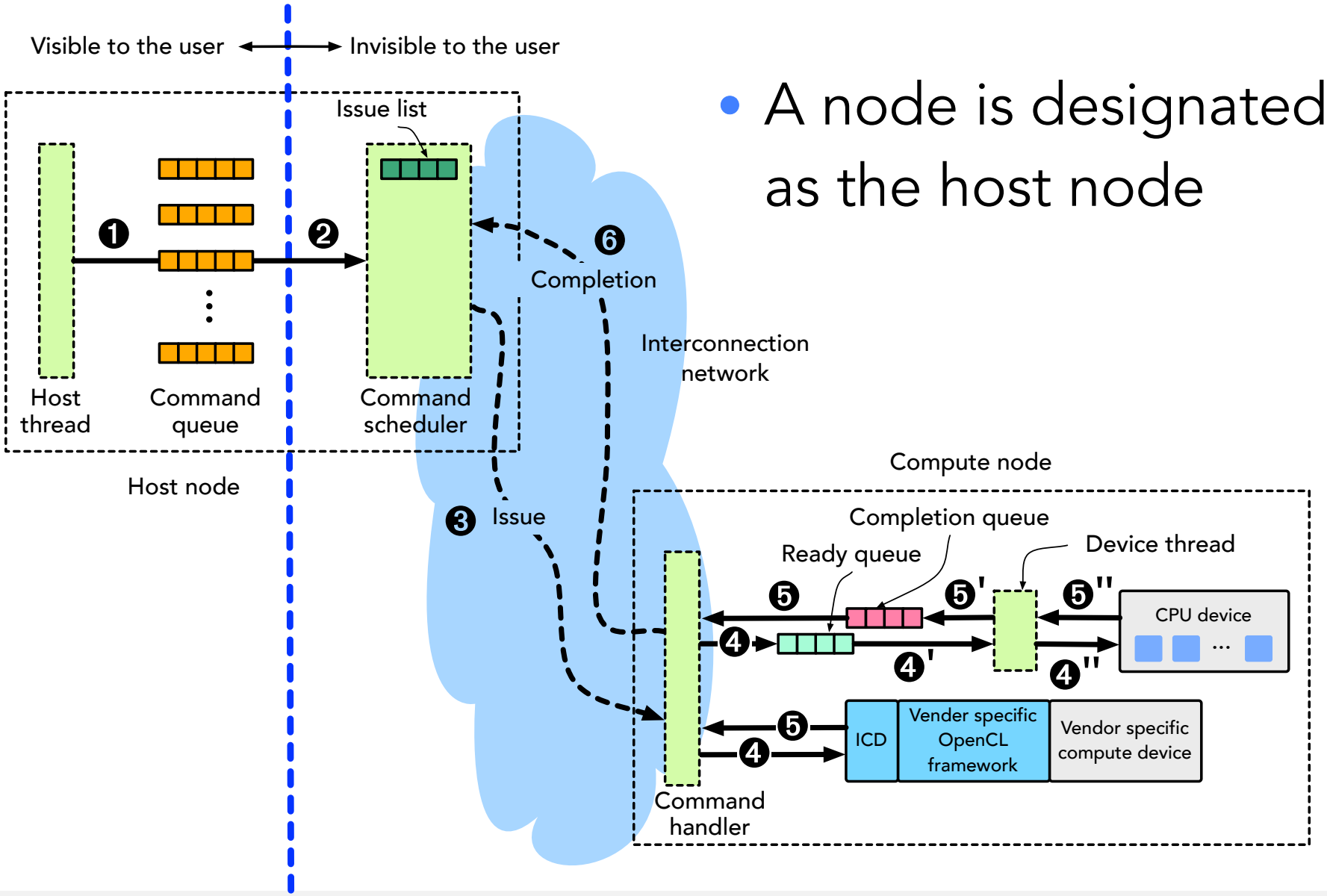
```
cl_int clEnqueueAlltoAllBuffer(
  cl_command_queue *cmd_queue_list, cl_uint num_buffers,
  cl_mem *src_buffer_list, cl_mem *dst_buffer_list,
  size_t *src_offset_list, size_t *dst_offset_list,
  size_t bytes_to_copy, cl_uint num_events_in_wait_list,
  const cl_event *event_wait_list, cl_event *event)
```



Part II: Introduction to SnuCL

- Illusion of a single OpenCL platform Image
- Collective communication extensions to OpenCL
- SnuCL runtime
- How to write SnuCL applications
- SnuCL benchmark applications
- SnuCL Performance evaluation
- Future directions

SnuCL Runtime



- A node is designated as the host node

Dynamic Scheduling for CPU/Xeon Phi Devices

- From Li et al. [ICPP 1993]
- S = ⌈N/(2P)⌉
  - 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

Part II: Introduction to SnuCL

- Illusion of a single OpenCL platform Image
- Collective communication extensions to OpenCL
- SnuCL runtime
- How to write SnuCL applications
- SnuCL benchmark applications
- SnuCL Performance evaluation
- Future directions

Matrix Multiply Implementations

- For a single GPU
- For multiple GPUs in a single OpenCL platform
- For multiple GPUs in a heterogeneous cluster
  - OpenCL + MPI
  - SnuCL
  - SnuCL with collective communication extensions

SnuCL for the Cluster

- Assume that the cluster has 32 GPUs
- Use the multi-GPU version of the matrix multiply
  - Modification in the OpenCL host program
    - **`#define MAX_DEV 4` ⇒ `#define MAX_DEV 32`**

Using Communication Extensions

```
for (i = 0; i < ndev; i++) {
  clEnqueueWriteBuffer(command_queue[i], bufferA[i], CL_FALSE, 0, sizeA/ndev,
                       (void*) ((size_t) hostA+(sizeA/ndev)*i),
                       0, NULL, NULL);
}
for (i = 0; i < ndev; i++) {
  clEnqueueWriteBuffer(command_queue[i], bufferB[i], CL_FALSE, 0, sizeB,
                       hostB, 0, NULL, NULL);
}
```

```
clEnqueueWriteBuffer(command_queue[0], bufferB[0], CL_TRUE, 0, sizeB,
                     hostB, 0, NULL, NULL);
clEnqueueBroadcastBuffer(cmq + 1, bufferB[0], ndev-1, bufferB+1,
                         0, NULL, sizeB, 0, NULL, NULL);
```

Matrix Multiply for CPU Devices

- The kernel code is portable across different types of compute devices
- The OpenCL application for GPUs will run on CPU devices too
  - Replace **`CL_DEVICE_TYPE_GPU`** with **`CL_DEVICE_TYPE_CPU`**

Part II: Introduction to SnuCL

- Illusion of a single OpenCL platform Image
- Collective communication extensions to OpenCL
- SnuCL runtime
- How to write SnuCL applications
- SnuCL benchmark applications
- SnuCL Performance evaluation
- Future directions

SNU NPB Suite

- Most of the applications in NAS Parallel Benchmarks (NPB 3.3) are implemented in C, OpenMP C, and OpenCL [IISWC '11]
  - NPB-SER-C: a serial C version of the NPB code
  - NPB-OMP-C: an OpenMP C version of the NPB code
  - NPB-OCL: an OpenCL version of the NPB code for a single device
  - NPB-OCL-MD: an OpenCL version of the NPB code for multiple OpenCL compute devices

- Source code is publicly available
  - http://aces.snu.ac.kr

Part II: Introduction to SnuCL

- Illusion of a single OpenCL platform Image
- Collective communication extensions to OpenCL
- SnuCL runtime
- How to write SnuCL applications
- SnuCL benchmark applications
- SnuCL Performance evaluation
- Future directions

Applications

Application	Source	Description	Input	Global memory size (MB)	Extensions used
BinomialOption	AMD	Binomial option pricing	65504 or 2097152 samples, 512 steps, 100 iterations	2.0 or 64.0	
BlackScholes	PARSEC	Black-Scholes PDE	33538048 options, 100 iterations	895.6	
BT	NAS	Block tridiagonal solver	Class C or Class D	1982.1 or 30686.7	
CG	NAS	Conjugate gradient	Class C or Class D	1102.6 or 20399.1	
CP	Parboil	Coulombic potential	16384x16384, 1000 atoms	4.1	
EP	NAS	Embarrassingly parallel	Class D	0.8	
FT	NAS	3-D FFT PDE	Class B or Class C	2816.0 or 11264.0	AlltoAll
MatrixMul	NVIDIA	Matrix multiplication	10752x10752 or 16384x16384	1323.0 or 3072.0	Broadcast
MG	NAS	Multigrid	Class C or Class D	3575.3 or 28343.7	
Nbody	NVIDIA	N-Body simulation	1048576 bodies	64.0	
SP	NAS	Pentadiagonal solver	Class C or Class D	1477.9 or 19974.4	

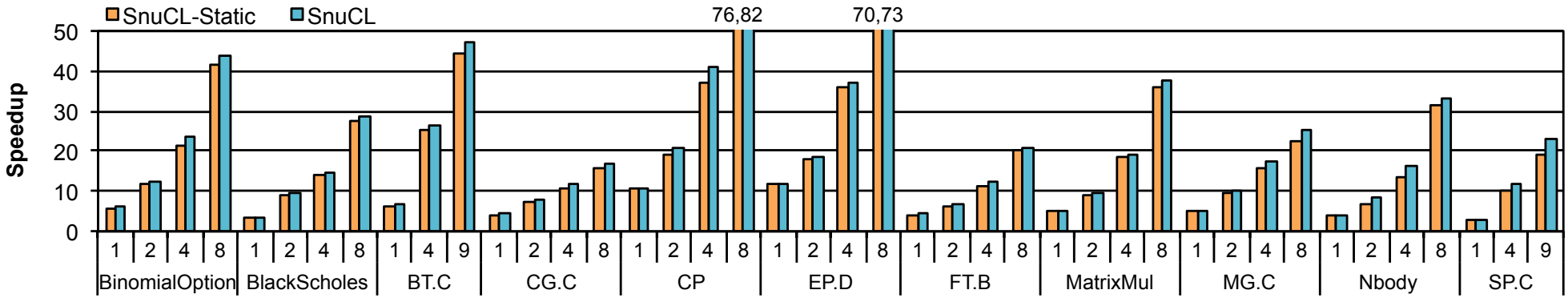
Cluster A

- 1 host node and 9 compute nodes
- A CPU device consists of 22 logical cores (11 physical cores)

	Host node	Compute node	
Processors	2 x Intel Xeon X5680	2 x Intel Xeon X5660	4 x NVIDIA GTX 480
Clock frequency	3.33GHz	2.80GHz	1.40GHz
Cores per processor	6	6	480
Memory size	72GB	48GB	1.5GB
Quantity	1	9	
OS	Red Hat Enterprise Linux Server 5.5		
Interconnection	Mellanox InfiniBand QDR		

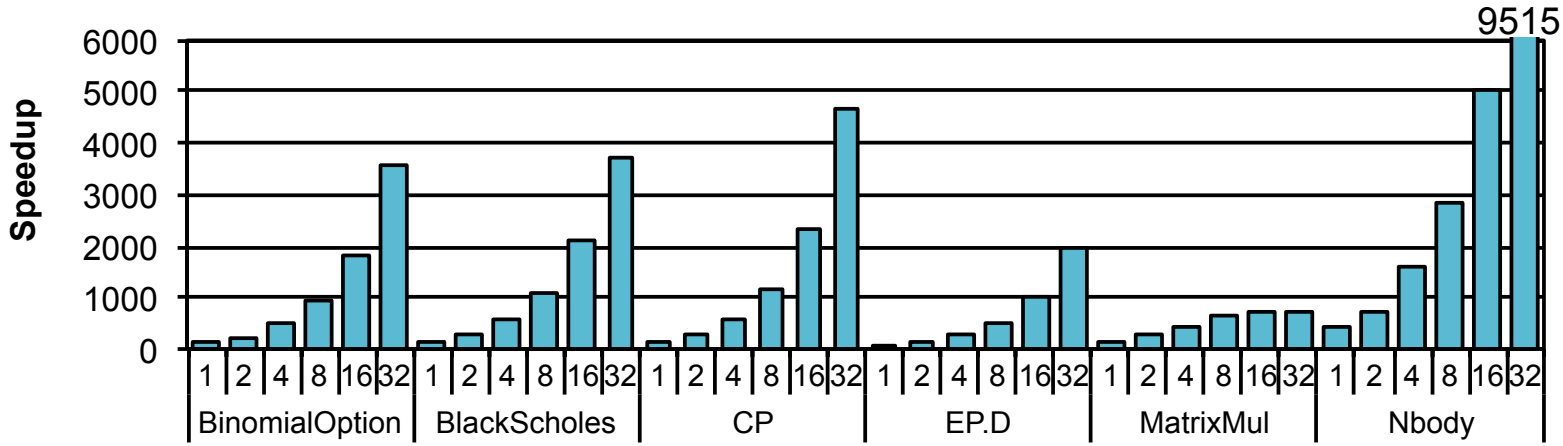
Speedup with CPU Devices (over 1 CPU core)

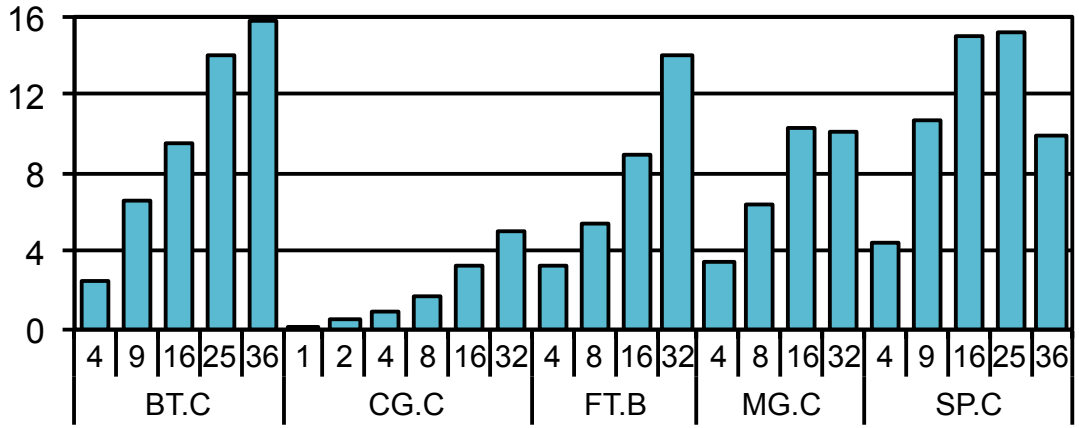
- CPU devices only
- SnuCL-Static : Using the static scheduling for the kernel workload distribution
- The numbers on x-axis represent the number of CPU compute devices



Speedup with GPU Devices (over 1 CPU core)

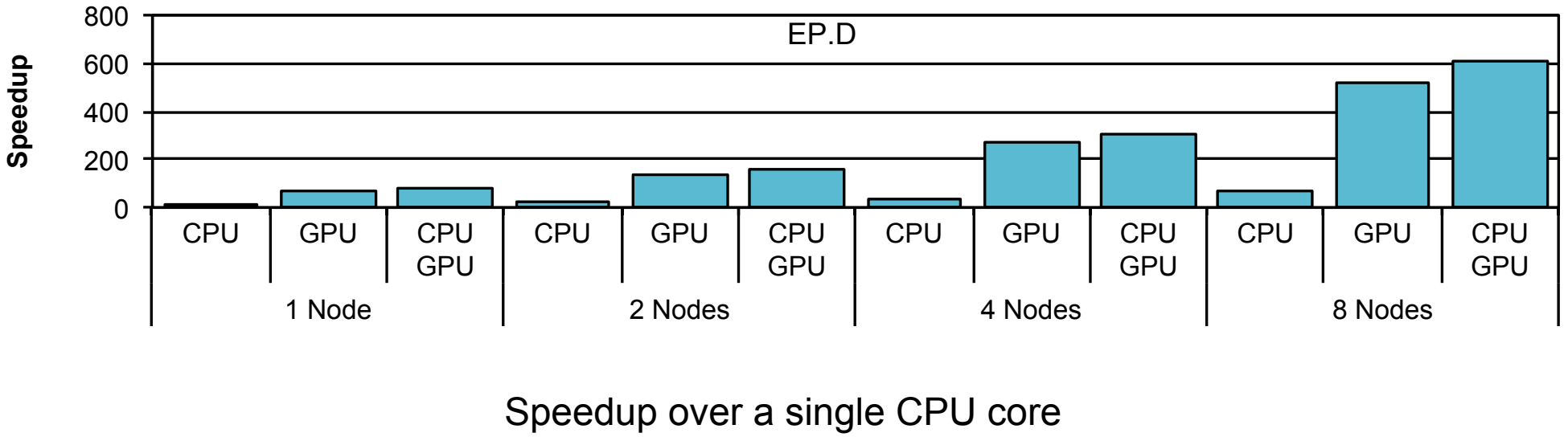
- GPU devices only
- The numbers on x-axis represent the number of GPU compute devices





Exploiting both CPU and GPU Devices

- 1 CPU device and 1 GPU device per node
- Load balancing between the CPU device and the GPU device
  - Based on their throughput



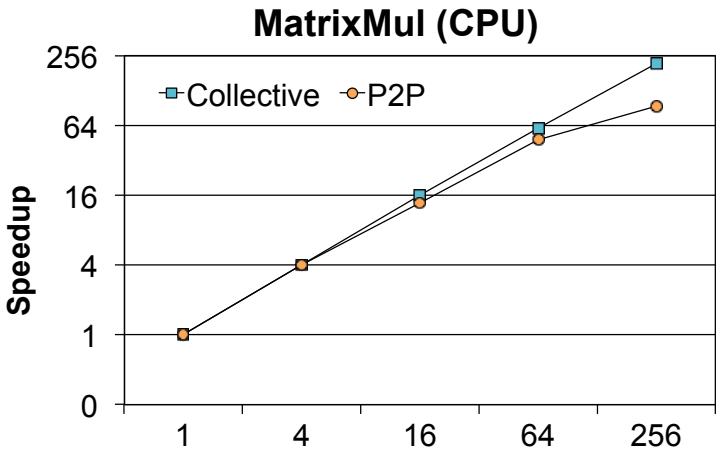
Cluster B

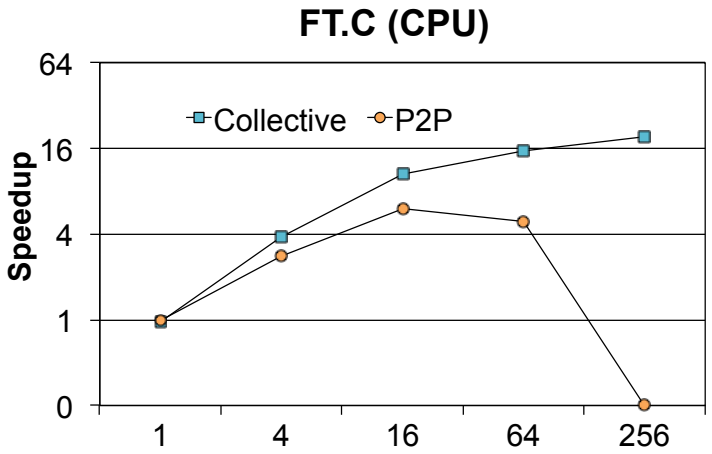
- 257-node CPU cluster to test scalability

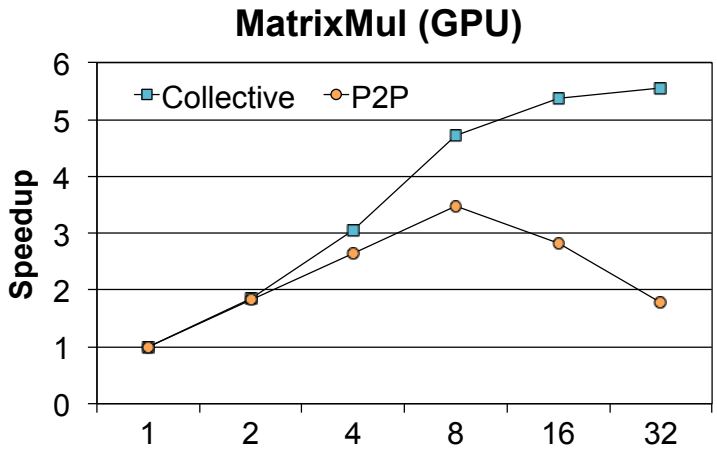
	Host node	Compute node
Processors	2 x Intel Xeon X5570	2 x Intel Xeon X5570
Clock frequency	2.93GHz	2.93GHz
Cores per processor	4	4
Memory size	24GB	24GB
Quantity	1	256
OS	Red Hat Enterprise Linux Server 5.3	
Interconnection	Mellanox InfiniBand QDR	

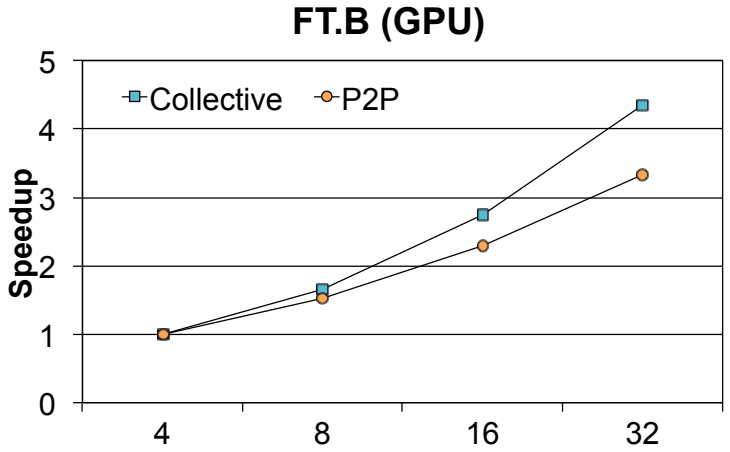
Collective Communication

- P2P : Using **`clEnqueueCopyBuffer()`**
- The numbers on x-axis represent the number of compute devices



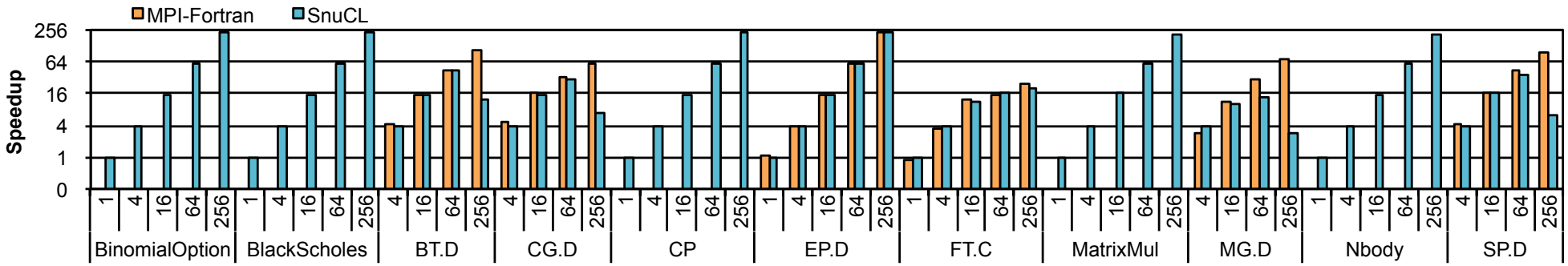






SnuCL vs. MPI-Fortran

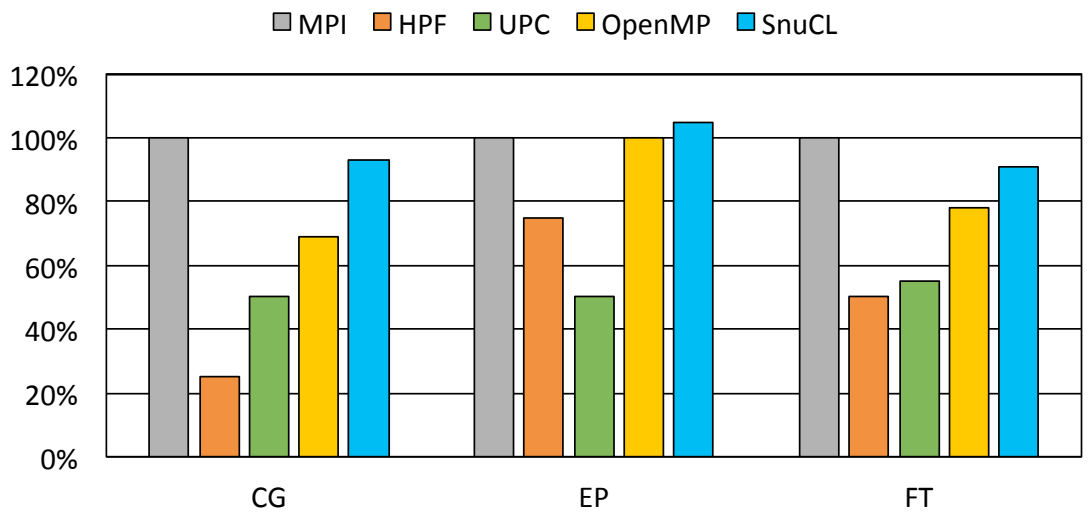
- Speedup over a single compute node (a CPU compute device with 4 CPU cores) in Cluster B
- Normalized to MPI-Fortran
  - MPI-Fortran: the unmodified original MPI-Fortran versions from NPB
- The numbers on x-axis represent the number of compute nodes (CPU compute devices)



SnuCL and other Shared Memory Programming Models

- **HPF**: Christian Clemencon et al, HPF and MPI Implementation of the NAS Parallel Benchmarks Supported by Integrated Program Engineering Tools, PDCS 1996
- **UPC**: Tarek El-Ghazawi et al, UPC Performance and Potential: A NPB Experimental Study, SC 2002
- **OpenMP**: Okwan Kwon et al, A Hybrid Approach of OpenMP for Clusters, PPoPP 2012

- Using number of nodes in common, applications in common
- Speedup over the MPI-Fortran version with 16 nodes



Part II: Introduction to SnuCL

- Illusion of a single OpenCL platform Image
- Collective communication extensions to OpenCL
- SnuCL runtime
- How to write SnuCL applications
- SnuCL benchmark applications
- SnuCL Performance evaluation
- Future directions

Future Directions

- SnuCL with ICD (by July 2013)
- Scalability up to 1000 nodes (by December 2013)
- Auto vectorization (by December 2013)
- Planned
  - Autotuning
  - Performance portability
  - Intelligent load balancing
  - Achieving a single compute device image for multiple heterogeneous devices in a heterogeneous cluster

References

- **[ICS '12]** Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. SnuCL: an OpenCL Framework for Heterogeneous CPU/GPU Clusters, *ICS '12: Proceedings of the 26th International Conference on Supercomputing*, San Servolo Island, Venice, Italy, June 2012.
- **[IISWC '11]** Sangmin Seo, Gangwon Jo, and Jaejin Lee. Performance Characterization of the NAS Parallel Benchmarks in OpenCL, *IISWC '11: Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, Austin, Texas, USA, November 2011.
- **[PACT '11]** Jun Lee, Jungwon Kim, Junghyun Kim, Sangmin Seo, and Jaejin Lee. An OpenCL Framework for Homogeneous Manycores with no Hardware Cache Coherence, *PACT '11: Proceedings of the 20th ACM/IEEE/IFIP International Conference on Parallel Architectures and Compilation Techniques*, Galveston Island, Texas, USA, October 2011.

References (contd.)

- **[LCPC '11]** Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. OpenCL as a Programming Model for GPU Clusters, *LCPC '11: Proceedings of the 24th International Workshop on Languages and Compilers for Parallel Computing*, Fort Collins, Colorado, USA, September 2011.
- **[PPoPP '11]** Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. Achieving a Single Compute Device Image in OpenCL for Multiple GPUs, *PPoPP '11: Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 277 — 288, San Antonio, Texas, USA, February 2011, DOI: 10.1145/1941553.1941591.
- **[PACT '10]** Jaejin Lee, Jungwon Kim, Sangmin Seo, Seungkyun Kim, Jungho Park, Honggyu Kim, Thanh Tuan Dao, Yongjin Cho, Sung Jong Seo, Seung Hak Lee, Seung Mo Cho, Hyo Jung Song, Sang-Bum Suh, and Jong-Deok Choi. An OpenCL Framework for Heterogeneous Multicores with Local Memory, *PACT '10: Proceedings of the 19th ACM/IEEE/IFIP International Conference on Parallel Architectures and Compilation Techniques*, pp. 193 — 204, Vienna, Austria, September 2010, DOI: 10.1145/1854273.1854301.

Contributors to SnuCL

Jungwon Kim

Sangmin Seo

Jun Lee

Gangwon Jo

Junghyun Kim

Jungho Park

Jeongho Nah

Jaejin Lee

A new version of SnuCL will be available in July 2013

http://aces.snu.ac.kr