공학박사학위논문

An OpenCL Framework for Heterogeneous Clusters

이종 클러스터를 위한 OpenCL 프레임워크

2013년 8월

서울대학교 대학원
전기컴퓨터공학부
김 정 원

Abstract

OpenCL is a unified programming model for different types of computational units in a single heterogeneous computing system. OpenCL provides a common hardware abstraction layer across different computational units. Programmers can write OpenCL applications once and run them on any OpenCL-compliant hardware. However, one of the limitations of current OpenCL is that it is restricted to a programming model on a single operating system image. It does not work for a cluster of multiple nodes unless the programmer explicitly uses communication libraries, such as MPI. A heterogeneous cluster contains multiple general-purpose multicore CPUs and multiple accelerators to solve bigger problems within an acceptable time frame. As such clusters widen their user base, application developers for the clusters are being forced to turn to an unattractive mix of programming models, such as MPI-OpenCL. This makes the application more complex, hard to maintain, and less portable.

In this thesis, we propose SnuCL, an OpenCL framework for heterogeneous clusters. We show that the original OpenCL semantics naturally fits to the heterogeneous cluster programming environment, and

the framework achieves high performance and ease of programming. SnuCL provides a system image running a single operating system instance for heterogeneous clusters to the user. It allows the application to utilize compute devices in a compute node as if they were in the host node. No communication API, such as the MPI library, is required in the application source. With SnuCL, an OpenCL application becomes portable not only between heterogeneous devices in a single node, but also between compute devices in the cluster environment. We implement SnuCL and evaluate its performance using eleven OpenCL benchmark applications.

Keywords : OpenCL, Clusters, Heterogeneous computing, Programming models, Runtime system, Parallelization

Student ID : 2006-23153

Contents

List of Figures

List of Tables

Chapter I

Introduction

A recent trend in high-performance computing (HPC) is to use heterogeneous parallel systems including accelerators such as GPUs, Cell BE processors, and Intel Xeon Phi coprocessors. This trend is driven by the need for achieving high performance at low energy consumption. Programming for heterogeneous architectures uses multiple diverse platforms and systems simultaneously and it adds to the complexity of programming. The complexity of heterogeneous parallel programming hinders effective use of available resources by programmers, and it leads to lower productivity.

OpenCL[21] is an open and cross-platform programming model for heterogeneous parallel computing systems. OpenCL provides a common low-level hardware abstraction across different devices and architectures. Programmers can write OpenCL applications once and run them on any OpenCL-compliant hardware.

The platform model of OpenCL is designed for a single operating

system image. All OpenCL implementations from various hardware vendors are implemented for the model. In order to write an OpenCL application for clusters with multiple nodes, the application is being forced to turn to an unattractive mixture of programming models: OpenCL for accelerators and MPI for inter-node communication.

I.1 Heterogeneous Computing

A heterogeneous computing system typically refers to a single computer system that contains different types of computational units. It distributes data and program execution among different computational units that are each best suited to specific tasks. The computational unit could be a CPU, GPU, DSP, FPGA, or ASIC. Introducing such additional, specialized computational resources in a system enables the user to gain extra performance. In addition, exploiting the inherent capabilities of a wide range of computational resources enables the user to solve difficult and complex problems efficiently and easily. A typical example of the heterogeneous computing system is a GPGPU system.

The GPGPU system has been a great success so far. However, in the future, applications may not be written for GPGPUs only, but for more general heterogeneous computing systems to improve power efficiency and performance. Open Computing Language (OpenCL)[21] is a unified programming model for different types of computational

units in a heterogeneous computing system. OpenCL provides a common hardware abstraction layer across different computational units. Programmers can write OpenCL applications once and run them on any OpenCL-compliant hardware. This portability is one of OpenCL's chief advantages. With OpenCL, programmers no longer have to use vendor-specific languages or libraries to write a program for vendor-specific hardware. Some industry-leading hardware vendors such as AMD[2], IBM[15], Intel[18], NVIDIA[37], and Samsung[40] have provided OpenCL implementations for their hardware. This makes OpenCL a standard parallel programming model for general-purpose, heterogeneous computing systems.

I.2 Motivation

One of the limitations of current OpenCL is that it is restricted to a programming model on a single operating system image. The same thing is true for CUDA[25]. A heterogeneous CPU/GPU cluster contains multiple general-purpose multicore CPUs and multiple GPUs to solve bigger problems within an acceptable time frame. As such clusters widen their user base, application developers for the clusters are being forced to turn to an unattractive mix of programming models, such as MPI-OpenCL and MPI-CUDA. This makes the application more complex, hard to maintain, and less portable.

The mixed programming model requires the hierarchical distri-

bution of the workload and data (across nodes and across compute devices in a node). MPI functions are used to communicate between the nodes in the cluster. As a result, the resulting application may not be executed in a single node.

To use OpenCL as a programming model for clusters, there are some limitations arising from the architecture of OpenCL. First, OpenCL is designed for a master/slave execution model. A single host manages multiple compute devices. The main drawback of this model is its lack of scalability. The master can become a communication bottleneck if there are many slaves[24]. Second, OpenCL uses shared memory in the compute devices. An OpenCL memory object can be shared between multiple compute devices. When multiple compute devices in different nodes share an OpenCL memory object, the case is similar to the concept of distributed shared memory systems[19]. For these reasons, it has seemed that OpenCL is not a natural candidate as a programming model for clusters.

I.3 Related Work

Heterogeneous computing has been drawn much attention due to its parallelism, energy efficiency and cost effectiveness. Recently, there have been many studies done on GPU clusters. Fan *et al.*[11] are the first to develop a scalable GPU cluster for high performance scientific computing and large-scale simulation using graphics program-

ming APIs. They build a GPU cluster with 32 nodes. Each node consists of a dual-core CPU with an NVIDIA GeForce FX 5800 Ultra. They implement a Lattice-Boltzmann solver on the cluster to simulate the transport of airborne contaminants in the Times Square area of New York City. They achieve a speedup of 4.6 on the GPU cluster over the same model implemented on a CPU cluster. Phillips *et al.*[38] advance GPU clusters using a multi-GPU capable host system and CUDA, a general purpose programming language for GPU. They build a GPU cluster for simulation of large bio-molecular systems and achieve performance that nearly matches the performance of 330 CPU cores with fifteen 4-GPU nodes. Chen *et al.*[8] implement large-scale FFT on a GPU cluster. They achieve 5 times speedup with respect to Intel MKL for 4096 3D double-precision FFT on the 16-node cluster with 32 GPUs. They exploit an all-to-all collective communication operation to distribute data across nodes efficiently. In all of these studies, the authors develop their applications using the MPI library to implement communication between nodes in the cluster. Our approach, on the other hand, does not require any communication APIs, such as MPI, in the application code.

In addition to GPU clusters, there are some studies that exploit CPU/GPU clusters. Fatica[12] and Yang *et al.*[44] propose heterogeneous CPU/GPU clusters to accelerate Linpack. They distribute the workload across CPUs and GPUs based on their throughput. Especially, Yang *et al.*'s TianHe-1 system ranked No. 5 in the TOP500

list[1] published in November 2009. Moreover, its upgraded version, TianHe-1A was ranked as the world's first fasted supercomputer in the TOP500 list released in November 2010.

Chen *et al.*[7] propose new language extensions to Unified Parallel C (UPC) in order to take advantage of GPU clusters. They extend UPC with hierarchical data distribution and introduce the implicit thread hierarchy. They implement the compiler and runtime system, and show that their model has better programmability than the mixed MPI/CUDA approach, and the model is effective to achieve good performance on GPU clusters. We, on the other hand, show that the original OpenCL semantics naturally fits to the GPU clusters, and present the OpenCL framework for such clusters.

Fan *et al.*[11] are the first to develop a scalable GPU cluster for high performance scientific computing and large-scale simulation using graphics programming APIs. They build a GPU cluster with 32 nodes. Each node consists of a dual-core CPU with an NVIDIA GeForce FX 5800 Ultra. They implement a Lattice-Boltzmann solver on the cluster to simulate the transport of airborne contaminants in the Times Square area of New York City. They achieve a speedup of 4.6 on the GPU cluster over the same model implemented on a CPU cluster. Phillips *et al.*[38] advance GPU clusters using a multi-GPU capable host system and CUDA. They build a GPU cluster for simulation of large bio-molecular systems and achieve performance that nearly matches the performance of 330 CPU cores with fifteen 4-GPU

nodes. Chen *et al.*[8] implement large-scale FFT on a GPU cluster. They achieve 5 times speedup with respect to Intel MKL for 4096 3D double-precision FFT on the 16-node cluster with 32 GPUs. In all of these studies, the authors develop their applications using the MPI library to implement communication between nodes in the cluster. Our approach, on the other hand, does not require any communication APIs, such as MPI, in the application code.

Kim *et al.*[22] propose an OpenCL framework for multiple GPUs in a system. The OpenCL framework provides an illusion of a single compute device to the programmer for the multiple GPUs available in the system. It automatically partitions the work-group index space of the kernel at run time. To find an optimal partition that minimizes data transfer through the PCI-E bus between the host and GPUs, they use a sampling technique that analyzes the buffer access ranges in the kernel. To achieve a single compute device image, the runtime maintains a virtual device memory and copies them to each device memory when required. While their proposed OpenCL framework provides an illusion of a single compute device to the programmer, our OpenCL framework provides an illusion of a single system to the user for the multiple compute nodes available in GPU clusters.

OpenMP is another platform-independent programming model. It is an industry standard language, widely used for parallel programming on shared memory multiprocessors. There are some proposals[30, 29] to make OpenMP programs portable to the GPGPU systems. Lee

et al.[30] present a compiler framework for translation OpenMP programs into CUDA programs. It converts the loop-level parallelism of the OpenMP programs into the data parallelism of the CUDA programs. OpenMPC[29] extends Lee *et al.*'s model by adding new OpenMP directives and environment variables, extended for CUDA. Thus, it offers not only portability but also tunability, to GPU programming. They extend OpenMP programs' portability to the GPGPU systems with compiler techniques, while our work extends OpenCL programs' portability to the heterogeneous CPU/GPU systems with both of compiler and runtime system.

There are some previous proposals for OpenCL frameworks[14, 28, 22, 23, 27]. Gummaraju *et al.*[14] present an OpenCL framework named Twin Peaks that handles both CPUs and GPUs in a single node. Twin Peaks executes SPMD style OpenCL kernels on a CPU core by switching contexts between work-items. They use their own light-weight `setjmp()` and `longjmp()` system calls to reduce the context switching overhead. Lee *et al.*[27] propose an OpenCL framework for heterogeneous multicores with local memory, such as Cell BE processors. They present work-item coalescing technique and show that it significantly reduces context switching overhead of executing an OpenCL kernel on multiple SPEs. Lee *et al.*[28] present an OpenCL framework for homogeneous manycore processors with no hardware cache coherence mechanism, such as the Single-chip Cloud Computer (SCC). Their OpenCL runtime exploits the SCC's dynamic memory

mapping mechanism together with the symbolic array bound analysis to preserve coherence and consistency between CPU cores.

Some other prior work proposes GPU virtualization[22, 10, 23]. Kim *et al.*[22] propose an OpenCL framework for multiple GPUs in a single node. The OpenCL framework provides an illusion of a single compute device to the programmer for the multiple GPUs available in the system. Duato *et al.*[10] presents a CUDA framework named rCUDA. The framework enables multiple clients to share GPUs in a remote server. These approaches are similar to our work in that OpenCL or CUDA is used as an abstraction layer to provide ease of programming.

A Single System Image (SSI) operating system is a physical or logical mechanism giving the illusion that a set of distributed systems forms a unique single system. Morin *et al.*[33] present an operating system called Kerrighed. Kerrighed provides a view of a single SMP machine on top of a cluster. Walker [6] provides OpenSSI that allows to dynamically balance the cluster CPU load by using a process migration scheme. Both of Kerrighed and OpenSSI cannot virtualize the accelerators such as GPUs in the remote nodes. Moreover, these implementations required modified operating systems whereas SnuCL uses unmodified operating systems.

Garland *et al.*[13] propose a new programming model named Phalanx. Phalanx presents the programmers a unified programming model

for heterogeneous machines, in both a single node and multiple nodes in a distributed system. Phalanx allows the programmers to control the placement of tasks and data across the entire machine. Phalanx defines its interface in terms of a set of new generic functions and template types. In contrast, we show that the original OpenCL semantics naturally fits to the distributed systems, and extend OpenCL semantics to the heterogeneous cluster environment.

The OpenCL Common Runtime from IBM[16] integrates multiple OpenCL implementations into a single OpenCL programming environment. The main goal of this runtime is similar to our OpenCL runtime. It improves application portability and resource usages, and reduces programming complexity. However, unlike our OpenCL runtime, this runtime is limited to a single node. Even though, it automatically manages the resources across multiple platforms in a system, it requires programmers to write specialized code to manage and synchronize the resources across multiple nodes. On the other hand, all OpenCL resources are seamlessly shared and managed across all devices on the multiple nodes in the cluster.

I.4 Contributions

In this thesis, we propose an OpenCL framework called SnuCL[24] and show that OpenCL can be a unified programming model for heterogeneous CPU/GPU clusters. The target cluster architecture is shown

in Figure I.1. It consists of a single host node and multiple compute nodes. The nodes are connected by an interconnection network, such as Gigabit Ethernet and InfiniBand switches. The host node executes the host program in an OpenCL application. Each compute node consists of multiple multicore CPUs and multiple GPUs. A set of CPU cores or a single GPU becomes an OpenCL compute device. A GPU has its own device memory, up to several gigabytes. Within a compute node, data is transferred between the GPU device memory and the main memory through a PCI-E bus.

SnuCL provides a system image running a single operating system instance for heterogeneous CPU/GPU clusters to the user as shown in Figure I.1. It allows the application to utilize compute devices in a compute node as if they were in the host node. The user can launch a kernel to a compute device or manipulate a memory object in a remote node using only OpenCL API functions. This enables OpenCL applications written for a single node to run on the cluster without any modification. That is, with SnuCL, an OpenCL application becomes portable not only between heterogeneous computing devices in a single node, but also between those in the entire cluster environment.

The major contributions of this thesis are the following:

- We show that the original OpenCL semantics naturally fits to the heterogeneous cluster environment.
- We extend the original OpenCL semantics to the cluster envi-

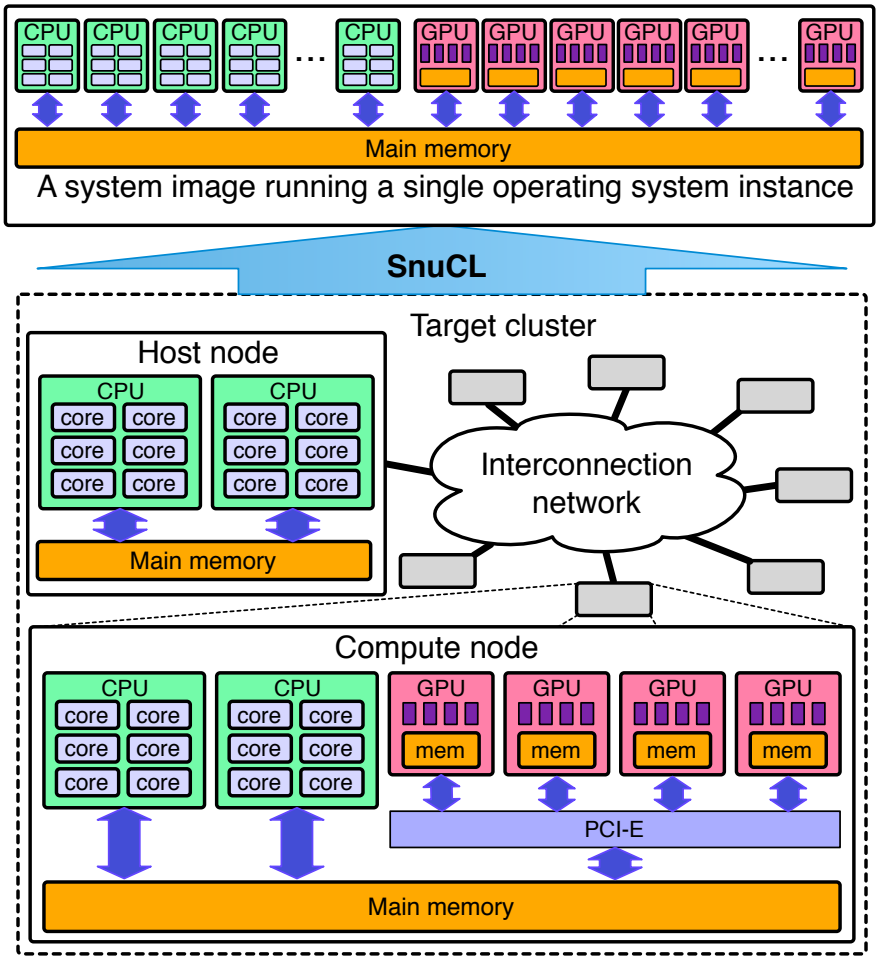

Figure I.1: Overview of SnuCL.

ronment to make communication between nodes faster and to achieve ease of programming.

- We describe the design and implementation of SnuCL (the runtime and source-to-source translators) for the heterogeneous CPU/GPU cluster.
- We develop an efficient memory management technique for the SnuCL runtime for the heterogeneous CPU/GPU cluster.
- We show the effectiveness of SnuCL by implementing the runtime and source-to-source translators. We experimentally demonstrate that SnuCL achieves high performance, ease of programming, and scalability for medium-scale heterogeneous clusters.

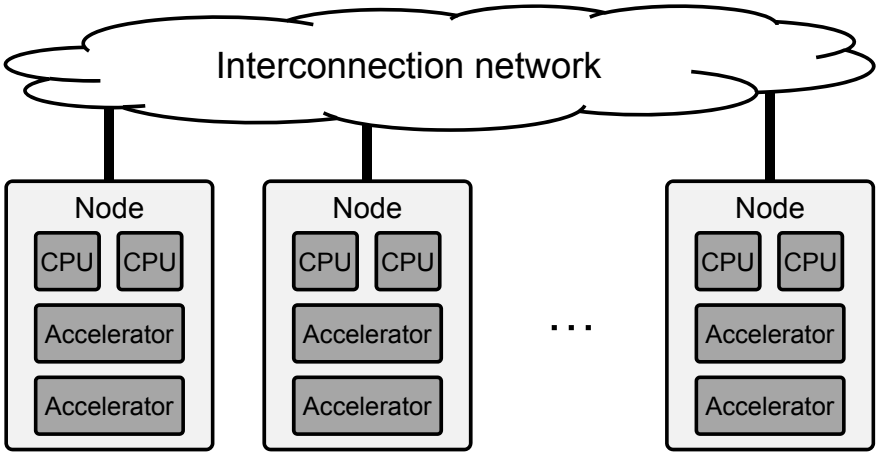

Figure I.2: Target cluster architecture of SnuCL-D.

To overcome the limitations encountered when using SnuCL, we propose two techniques. First, the framework provides an illusion that

each node has all available compute devices in the cluster to every node in the cluster. With this illusion, every node runs an OpenCL application. Secondly, the framework provides an efficient memory management technique by exploiting OpenCL's relaxed memory consistency model.

In this thesis, we propose an OpenCL framework called SnuCL-D for heterogeneous clusters. The target cluster architecture is shown in Figure I.2. It consists of multiple nodes, and they are connected by an interconnection network, such as Gigabit Ethernet and InfiniBand switches. Each node consists of one or more CPUs and one or more accelerators. The architecture of the nodes can be identical or different.

SnuCL-D extends the OpenCL platform model to the heterogeneous clusters. SnuCL-D enables the host to execute kernels and manipulate memory objects on the devices located in not only local node, but also remote nodes. With the framework, the programmer can write OpenCL applications for heterogeneous clusters using OpenCL only.

I.5 Organization of this Thesis

The rest of the thesis is organized as follows. Chapter II briefly describes OpenCL and its features. Chapter III describes the design and implementation of the SnuCL framework. Chapter IV introduces

the limitations of SnuCL and proposes a novel distributed execution model for SnuCL to overcome the limitations. Chapter V discusses and analyzes the evaluation results of our implementations. Finally, Chapter VI concludes the thesis.

Chapter II

The OpenCL Architecture

In this chapter, we briefly describe the OpenCL platform and its execution semantics.

II.1 Platform Model

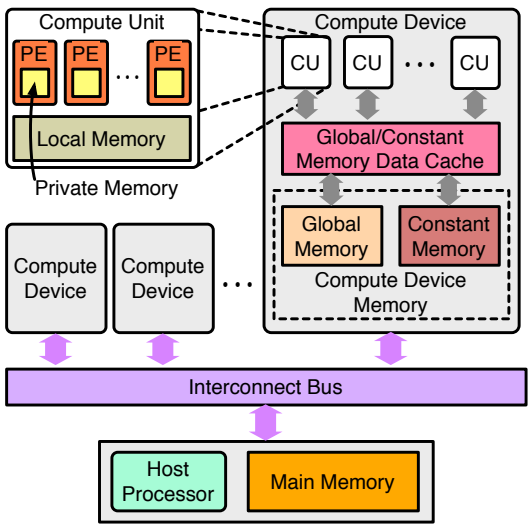

Figure II.1: The OpenCL platform model.

Figure II.1 shows the OpenCL platform model specified in the OpenCL specification[21]. The OpenCL platform consists of a host (host processor) connected to one or more *compute devices*, each of which contains one or more *compute units* (CUs). Each CU contains one or more *processing elements* (PEs). A PE is a *virtual* scalar processor. The host runs an operating system. A GPU, multicore CPU, Cell BE processor, Intel Xeon Phi coprocessor, or an accelerator can be a compute device. In addition, the host processor itself can be a compute device. Compute devices except CPUs communicate with the host processor using a peripheral interconnect, such as PCI-E buses.

II.2 Execution Model

An OpenCL application consists of a *host program* and *kernels*. The host program executes on the host processor and submits *commands* to perform computations on the PEs within a compute device or to manipulate memory objects. There are three types of commands: kernel execution, memory, and synchronization. A kernel is a function and written in OpenCL C. It executes on a single compute device. It is submitted to a *command-queue* in the form of a kernel execution command by the host program. A compute device may have one or more command-queues. Commands in a command-queue are issued in-order or out-of-order depending on the queue type. Commands are then scheduled onto compute devices. There are three different types

of commands: kernel-execution, memory, and synchronization. Commands enqueued in a command-queue are executed asynchronously with the host.

When the host program submit a kernel execution command to a command-queue, it defines an N-dimensional abstract index space, called NDRange for the kernel, where $1 \leq N \leq 3$. Each point in NDRange is specified by an N-tuple of integers with each dimension starting at 0. Each point is associated with an execution instance of the kernel, which is called *work-item*. Thus, the N-tuple becomes the global ID of the associated work-item. Each work-item performs a different task based on its ID in an SPMD[9] manner. Before enqueueing a kernel command, the host program defines an integer array of length N (i.e., the dimension of the NDRange) that specifies the number of work-items in each dimension of the NDRange. Each work-item executes the same code, but the specific pathway and data operated on can vary.

A *work-group* contains one or more work-items. Each work-group has a unique ID that is also an N-tuple. An integer array of length N specifies the number of work-groups in each dimension of the index space. A work-item in a work-group is assigned to a unique local ID within the work-group, treating the entire work-group as an index space. The index space is called a local index space. The global ID of a work-item can be computed with its local ID, work-group ID, and work-group size. Work-items in a work-group execute concurrently on

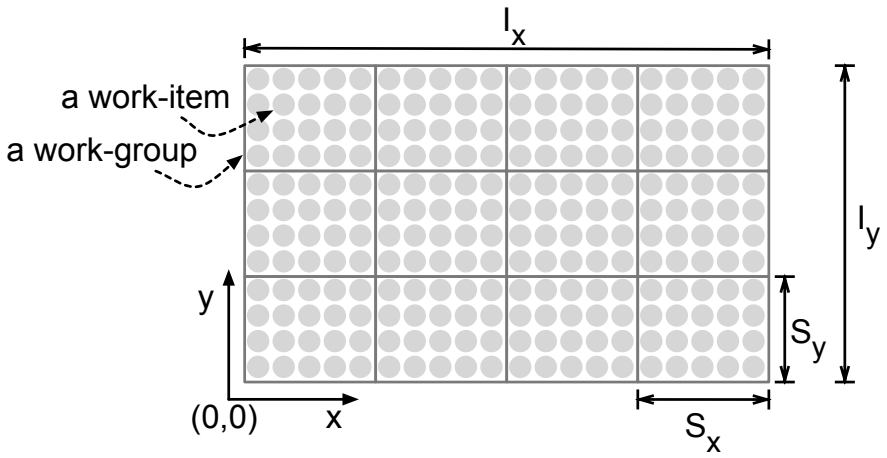

Figure II.2: A two-dimensional index space.

the PEs of a single CU.

For example, Figure II.2 shows a two-dimensional index space whose sizes in dimensions x and y are I_x and I_y respectively. Suppose that the work-group size in dimension x is S_x and in dimension y is S_y. Let $(ID_x^{global}, ID_y^{global})$, $(ID_x^{local}, ID_y^{local})$, and $(ID_x^{group}, ID_y^{group})$ be the global ID, local ID, and work-group ID of a work-item in the index space, respectively. The number of work-groups in dimension x is computed by I_x/S_x and in dimension y is by I_y/S_y. Each work-group contains $S_x \times S_y$ work-items. The global ID is computed with the work-group size, work-group ID and local ID,

$$ID_x^{global} = S_x \cdot ID_x^{group} + ID_x^{local}$$

$$ID_y^{global} = S_y \cdot ID_y^{group} + ID_y^{local}$$

II.3 Memory Model

OpenCL defines four distinct memory regions: global, constant, local and private. *Compute device memory* consists of the global and constant memory regions. The local memory is shared by all PEs in the same compute unit. The private memory is local to a PE. Accesses to the global memory or the constant memory may be cached in the global/constant memory data cache if there is such a cache in the device. The OpenCL runtime maps each OpenCL platform component to a component in the target GPGPU system. The entire GPU card becomes an OpenCL compute device.

The host program enqueues memory commands that operate on memory objects in the device memory. Only the host program can dynamically create global or constant memory objects with OpenCL API functions. Pointers to the memory objects are passed as arguments to a kernel that accesses the objects. A memory object in the device memory is typically a *buffer object*, called in short as a *buffer*. A buffer stores a one-dimensional collection of elements that can be a scalar data type, a vector data type, or a user-defined structure.

II.4 OpenCL Applications

For an example of the OpenCL kernel, consider a C function that performs matrix multiplication in Figure II.3 (a). It computes $A \times B$

```
void matrixMul(float* C, float* A, float* B, int WA, int HA, int WB)
{
1:   for (int j = 0;  j < HA; j++) {
2:       for (int i = 0; i < WB; i++) {
3:           float acc = 0.0f;
4:           for (int k = 0; k < WA; k++) {
5:               acc += A[k + j * WA] * B[i + k * WB];
6:           }
7:           C[i + j * WB] = acc;
8:       }
9:   }
}
```

(a)

```
__kernel void matrixMul( __global float* C, __global float* A,
                         __global float *B, int WA, int WB)
{
1:   int i = get_global_id(0);
2:   int j = get_global_id(1);
3:   float acc = 0.0f;
4:   for (int k = 0; k < WA; k++)
5:       acc += A[k + j * WA] * B[i + k * WB];
6:   C[i + j * WB] = acc;
}
```

(b)

```
#define BLOCK_SIZE 16
void main()
{
1:   ...
2:   cl_mem A_gpu, B_gpu, C_gpu;
3:   float *A_cpu, *B_cpu, *C_cpu;
4:   int WA, WB, WC;  // widths of matrices
5:   int HA, HB, HC;    // heights of matrices
6:                      // HB=WA, WC=WB, and HC=HA
7:   ...
8:   // Initialize the OpenCL runtime
9:   ...
10:  // Build and create the kernel
11:  ...
12:  // Allocate the main memory space for the matrices
13:  A_cpu = (float*) malloc(sizeA);
14:  B_cpu = (float*) malloc(sizeB);
15:  C_cpu = (float*) malloc(sizeC);
16:  // Initialize A_cpu, B_cpu, and C_cpu
17:  ...
18:  // Allocate the device memory for the matrices
19:  A_gpu = clCreateBuffer(context, CL_MEM_READ_ONLY,
20:                         sizeA, NULL, NULL);
21:  B_gpu = clCreateBuffer(context, CL_MEM_READ_ONLY,
22:                         sizeB, NULL, NULL);
23:  C_gpu = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
24:                         sizeC, NULL, NULL);
25:  ...
26:  // Copy the matrices from the main memory to the device memory
27:  clEnqueueWriteBuffer(command_queue, A_gpu, CL_TRUE, 0, sizeA,
28:                       A_cpu, 0, NULL, NULL);
29:  clEnqueueWriteBuffer(command_queue, B_gpu, CL_TRUE, 0, sizeB,
30:                       B_cpu, 0, NULL, NULL);
31:  ...
32:  // Set up kernel arguments
33:  clSetKernelArg(kernel, 0, sizeof(cl_mem), &C_gpu);
34:  clSetKernelArg(kernel, 1, sizeof(cl_mem), &A_gpu);
35:  clSetKernelArg(kernel, 2, sizeof(cl_mem), &B_gpu);
36:  clSetKernelArg(kernel, 3, sizeof(int), &WA);
37:  clSetKernelArg(kernel, 4, sizeof(int), &WB);
38:  // Set the size of the global index space
39:  size_t globalWorkSize[] = {WC, HC};
40:  // Set the size of the local index space
41:  size_t localWorkSize[] = {BLOCK_SIZE, BLOCK_SIZE};
42:  // Execute the kernel
43:  clEnqueueNDRangeKernel(command_queue, kernel, 2, 0,
44:                         globalWorkSize, localWorkSize, 0, NULL, NULL);
45:  // Copy the result from the device memory to the main memory
46:  clEnqueueReadBuffer(command_queue, C_gpu, CL_TRUE, 0, sizeC,
47:                      C_cpu, 0, NULL, NULL);
}
```

(c)

Figure II.3: (a) A C function for matrix multiplication. (b) The OpenCL kernel for the C function in (a). (c) The OpenCL host program for matrix multiplication.

and stores the result in C. Figure II.3 (b) shows an OpenCL kernel function that implements the C function. It is written in OpenCL C. OpenCL C has four address space qualifiers to distinguish different memory regions: `__global`, `__constant`, `__local` and `__private`. The `__global` qualifier is used in the argument declaration of the kernel. It tells the OpenCL C compiler that the buffers of matrices A, B, and C are allocated in the global memory. The kernel has a two-dimensional index space. OpenCL functions `get_global_id(0)` and `get_global_id(1)` return the first and second elements of the global ID of the work-item that executes the kernel, respectively. Each work-item computes an element of C.

Figure II.3 (C) shows the host program for the matrix multiplication. At the beginning, the host initializes the OpenCL runtime by creating an OpenCL context and a (in-order) command-queue for the compute device. Then, it builds and creates the kernel by invoking the OpenCL C compiler. The host allocates spaces for matrices A, B, and C (subscripted with `_cpu`) in the main memory (lines 12 – 15) and initializes them (lines 16 – 17). It also allocates buffer objects for matrices A, B, and C (subscripted with `_gpu`) in the device memory (lines 18 – 24). After allocating the buffers, the host copies matrices A and B from the main memory to the buffers by enqueueing memory commands to the command-queue (lines 26 – 30). After setting up kernel arguments to be passed to the kernel (lines 32 – 37), it specifies the dimension and size of the global and local index spaces (lines 38 –

41). The host executes the kernel by enqueueing a kernel command to the command-queue (lines 42 – 44). To copy matrix C from the buffer to the main memory, the host enqueues a memory command to the command-queue (lines 45 – 47). The command-queue is an in-order queue. Thus, the enqueued memory and kernel commands are issued to the compute device in order.

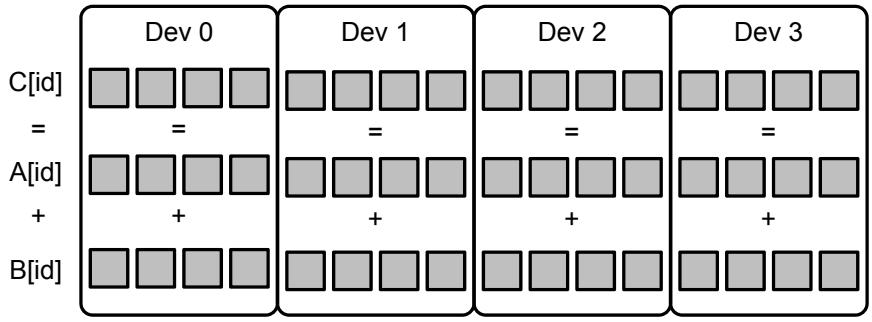

Figure II.4: Vector addition (C = A + B) with multiple compute devices.

To write an OpenCL application for multiple compute devices, the programmer needs to distribute workload across the multiple devices and manage data between the host memory and multiple device memories. For example, Figure II.4 shows an OpenCL application that performs vector addition with multiple compute devices. It adds vector A and B and places the result in vector C (i.e., C=A+B). Figure II.4 shows that there are total of 16 elements in vector A, B, and C, respectively. There are four compute devices and we distribute the workload evenly across the devices. Each device processes four elements.

```
1: __kernel void vecadd(__global float* C, __global float* A,
2:                      __global float* B) {
3:   int id = get_global_id(0);
4:   C[id] = A[id] + B[id];
5: }
```

(a)

```
1: #define SIZE      16
2: #define MAX_DEV   4
3:
4: int main(int argc, char** argv) {
5: cl_platform_id    platform;
6: cl_context        context;
7: cl_device_id      dev[MAX_DEV];
8: cl_command_queue  command_queue[MAX_DEV];
9: cl_mem            bufferA[MAX_DEV];
10: cl_mem           bufferB[MAX_DEV];
11: cl_mem           bufferC[MAX_DEV];
12: cl_kernel        kernel;
13: cl_uint          num_dev;
14: size_t           cb = SIZE * sizeof(float);
15: size_t           global, local, offset;
16: ...
17: // Allocate the main memory space for the vector
18: float *hostA = (float*) malloc(cb);
19: float *hostB = (float*) malloc(cb);
20: float *hostC = (float*) malloc(cb);
21: // Initialize hostA, hostB
22: ...
23: // Initialize the OpenCL objects
24: clGetPlatformIDs(1, &platform, NULL);
25: clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 0, NULL, &num_dev);
26: clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, num_dev, dev, NULL);
27: context = clCreateContext(0, num_dev, dev, NULL, NULL, NULL);
28: for (i = 0; i < num_dev; i++) {
29:   command_queue[i] = clCreateCommandQueue(context, dev[i], 0, NULL);
30:   bufferA[i] = clCreateBuffer(context, CL_MEM_READ_ONLY,
31:                               cb / num_dev, NULL, NULL);
32:   bufferB[i] = clCreateBuffer(context, CL_MEM_READ_ONLY,
33:                               cb / num_dev, NULL, NULL);
34:   bufferC[i] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
35:                               cb / num_dev, NULL, NULL);
36: }
37: ...
38: // Copy the vectors from the main memory to the device memory
39: for (i = 0; i < num_dev; i++) {
40:   offset = (cb / num_dev) * i;
41:   clEnqueueWriteBuffer(command_queue[i], bufferA[i], CL_FALSE, 0,
42:                        cb / num_dev, hostA + offset, 0, NULL, NULL);
43:   clEnqueueWriteBuffer(command_queue[i], bufferB[i], CL_FALSE, 0,
44:                        cb / num_dev, hostB + offset, 0, NULL, NULL);
45: }
46: // Set up kernel arguments and launch the kernel
47: kernel = clCreateKernel(program, "vecadd", NULL);
48: global = { SIZE / num_dev };
49: local = { 1 };
50: for (i = 0; i < num_dev; i++) {
51:   clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*) &bufferC[i]);
52:   clSetKernelArg(kernel, 1, sizeof(cl_mem), (void*) &bufferA[i]);
53:   clSetKernelArg(kernel, 2, sizeof(cl_mem), (void*) &bufferB[i]);
54:   clEnqueueNDRangeKernel(command_queue[i], kernel, 1, NULL, global,
55:                          local, 0, NULL, NULL);
56: }
57: // Copy the results from the device memory to the main memory
58: for (i = 0; i < num_dev; i++) {
59:   offset = (cb / num_dev) * i;
60:   clEnqueueReadBuffer(command_queue[i], bufferC[i], CL_TRUE,  0,
61:                       cb / num_dev, hostC + offset, 0, NULL, NULL);
62: }
63:
64: }
```

(b)

Figure II.5: (a) The OpenCL kernel for vector addition. (b) The OpenCL host program for vector addition.

Figure II.5 (a) shows the OpenCL kernel of the vector addition. When a work-item calls `get_global_id(0)`, it returns the work-item's global ID. The work-item then uses the global ID to index the data assigned to it.

Figure II.5 (b) shows the host program of the application. At the beginning, the host initializes OpenCL objects. The host gets an OpenCL platform in the system (line 24) and obtains an array of compute devices available in the platform (lines 25 – 26). After it creates a context with the devices (line 27), it creates a command-queue (line 29), and three memory objects for vector A, B, and C (lines 30 – 35) for each device. The size of each memory object is set to the vector size in bytes divided by the number of devices (`cb / num_dev`). The host copies vectors A and B from the main memory to the memory objects by enqueueing memory write commands to the command-queues (lines 41 – 44). After setting up kernel arguments (lines 51 – 53), it executes the kernels by enqueueing kernel commands to the command-queues (lines 54 – 55). The index space size for each kernel is set to the vector size divided by the number of devices (`SIZE / num_dev`). To copy vector C from the memory objects to the main memory, the host enqueues memory read commands to the command-queues (lines 60 – 61).

Chapter III

The SnuCL Framework

In this chapter, we describe the design and implementation of the SnuCL framework for the heterogeneous CPU/GPU cluster.

III.1 The SnuCL Runtime

III.1.1 Mapping Components

SnuCL defines a mapping between the OpenCL platform components and the target architecture components. A CPU core in the host node becomes the OpenCL host processor. A GPU or a set of CPU cores in a compute node becomes a compute device. Thus, a compute node may have multiple GPU devices and multiple CPU devices. The remaining CPU cores in the host node other than the host core can be configured as a compute device. Table III.1 summarizes the mapping. The host node executes the host program and compute nodes execute kernels in an OpenCL application. SnuCL runtime for GPU devices

Table III.1: Mapping the OpenCL platform to the target architecture.

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

OpenCL platform	Target architecture (compute node)
Compute device	A set of CPU cores
Compute unit	A 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 hardware coherence mechanism

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

is implemented using CUDA framework[36].

Since OpenCL has a strong CUDA heritage[25], the mapping between the components of an OpenCL compute device to those in a GPU is straightforward. For a compute device composed of multiple CPU cores, SnuCL maps all of the memory components in the compute device to disjoint regions in the main memory of the compute node where the device resides. Each CPU core becomes a CU, and the core emulates the PEs in the CU using the work-item coalescing technique[27].

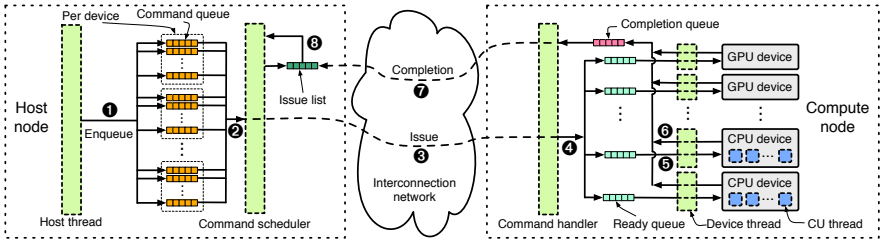

Figure III.1: The organization of the SnuCL runtime.

III.1.2 Organization of the SnuCL Runtime

Figure III.1 shows the organization of the SnuCL runtime. It consists of two different parts for the host node and a compute node. Execution of an OpenCL application in SnuCL runtime occurs in two parts: the host program that executes on the host node and kernels that execute on the compute devices in the compute nodes.

The runtime for the host node runs two threads: *host thread* and

command scheduler. When a user launches an OpenCL application in the host node, the host thread in the host node executes the host program in the application. The host thread and command scheduler share the OpenCL command-queues. A compute device may have one or more command-queues as shown in Figure III.1. The host thread enqueues commands to the command-queues (❶ in Figure III.1). The command scheduler schedules the enqueued commands across compute devices in the cluster one by one (❷).

When the command scheduler in the host node dequeues a command from a command-queue, the command scheduler *issues* the command by sending a *command message* (❸) to the target compute node that contains the target compute device associated with the command-queue. A command message contains the information required to execute the original command. To identify each OpenCL object, the runtime assigns a unique ID to each OpenCL object, such as contexts, compute devices, buffers (memory objects), programs, kernels, events, etc. The command message contains these IDs.

After the command scheduler sends the command message to the target compute node, it calls a non-blocking receive communication API function to wait for the completion message from the target node. The command scheduler encapsulates the receive request in the command event object and adds the event object in the *issue list*. The issue list contains event objects associated with the commands that have been issued but have not completed yet.

The runtime for a compute node runs a *command handler thread*. The command handler receives command messages from the host node and executes them across compute devices in the compute node. It creates a command object and an associated event object from the message. After extracting the target device information from the message, the command handler enqueues the command object to the *ready-queue* of the target device (❹). Each compute device has a single ready-queue. The ready-queue contains commands that are issued but not launched to the associated compute device yet.

The runtime for a compute node runs a *device thread* for each compute device in the node. If a CPU device exists in the compute node, each core in the CPU device runs a *CU thread* to emulate PEs. The device thread dequeues a command from its ready-queue and launches the kernel to the associated compute device when the command is a kernel-execution command and the compute device is idle (❺). If it is a memory command, the device thread executes the command directly.

When the compute device completes executing the command, the device thread updates the status of the associated event to *completed*, and then inserts the event to the *completion queue* in the compute node (❻). The command handler in each compute node repeats handling commands and checking the completion queue in turn. When the completion queue is not empty, the command handler dequeues the event from the completion queue and sends a completion message to the host node (❼).

The command scheduler in the host node repeats scheduling commands and checking the event objects in the issue list in turn until the OpenCL application terminates. If the receive request encapsulated in an event object in the issue list completes, the command scheduler removes the event from the issue list and updates the status of the dequeued event from *issued* to *completed* (❽).

The command scheduler in the host node and command handlers in the compute nodes are in charge of communication between different nodes. This communication mechanism is implemented with a lower-level communication API, such as MPI. To implement the runtime for each compute node, an existing CUDA or OpenCL runtime for a single node can be used.

III.1.3 Processing Kernel-execution Commands

When a device thread dequeues a kernel-execution command from its ready-queue, it launches the kernel to the target device when the device is idle. When the target device is a GPU, the device thread launches the kernel using the vendor-specific API, such as CUDA if the GPU vendor is NVIDIA. When the target is a CPU device, CU threads (i.e., CPU cores) in the device emulate the PEs using a kernel transformation technique, called work-item coalescing[27]. Basically, the work-item coalescing technique makes the CU thread execute each

work-item in a work-group one by one sequentially using a loop that iterates over the local index space in the work-group. This transformation is provided by the SnuCL OpenCL-C-to-C translator.

The CPU device thread dynamically distributes the kernel workload across the CU threads and achieves workload balancing between the CU threads. The unit of workload distribution is a work-group. The problem of work-group scheduling across the CU threads is similar to that of parallel loop scheduling for the conventional multiprocessor system because each work-group is essentially a loop due to the work-item coalescing technique. Thus, we modify the conventional parallel loop scheduling algorithm proposed by Li *et al.*[32] and use it in the SnuCL runtime.

In the SnuCL runtime, one or more work-groups are grouped together and dynamically assigned to a currently idle CU thread. The set of work-groups assigned to a CU thread is called a *work-group assignment*. To minimize the scheduling overhead, the size of each work-group assignment is large at the beginning, and the size decreases progressively. When there are N remaining work-groups, the size S of next work-group assignment to an idle CU thread is computed by $S{=}\lceil N/(2P)\rceil$, where P is the number of all CU threads in the CPU device. The CPU device thread repeatedly schedules the remaining work-groups until N is equal to zero.

III.1.4 Processing Synchronization Commands

OpenCL supports synchronization between work-items in a work-group using a *work-group barrier*. Every work-item in the work-group must execute the barrier and cannot proceed beyond the barrier until all other work-items in the work-group reach the barrier. Between work-groups, there is no synchronization mechanism available in OpenCL.

Synchronization between commands in a single command-queue can be specified by a *command-queue barrier* command. To synchronize commands between different command-queues, *events* are used. Each OpenCL API function that enqueues a command returns an event object that encapsulates the command status. Most of OpenCL API functions that enqueue a command take an *event wait list* as an argument. This command cannot be issued for execution until all the commands associated with the event wait list complete.

The command scheduler in the host node honors the type (in-order or out-of-order) of each command-queue and (event) synchronization enforced by the host program. When the command scheduler dequeues a synchronization command, the command scheduler uses it for determining execution ordering between queued commands. It maintains a data structure to store the events that are associated with queued commands and bookkeeps the ordering between the commands. When

there is no event for which a queued command waits, the command is dequeued and issued to its target node that contains the target device.

III.2 Memory Management

In this section, we describe how the SnuCL runtime manages memory objects and executes memory commands.

III.2.1 The OpenCL Memory Model

OpenCL defines four distinct memory regions in a compute device: global, constant, local and private. To distinguish these memory regions, OpenCL C has four address space qualifiers: `__global`, `__constant`, `__local`, and `__private`. They are used in variable declarations in the kernel code. Since OpenCL treats these memory regions as logically distinct regions, they may overlap in physical memory.

An OpenCL memory object is a handle to a region of the global memory. The host program dynamically creates a memory object and enqueues commands to read from, write to, and copy the memory object. A memory object in the global memory is typically a *buffer object*, called a *buffer* in short. A buffer stores a one-dimensional collection of elements that can be a scalar data type, vector data type, or user-defined structure.

OpenCL defines a relaxed memory consistency model. An update to a memory location by a work-item does not need to be visible to other work-items at all times. Instead, the local view of memory from each work-item is guaranteed to be consistent at synchronization points. Synchronization points include work-group barriers, command-queue barrier, and events. Especially, the device global memory is consistent across work-items in a single work-group at a work-group barrier, but there are no guarantees of memory consistency between different work-groups executing the kernel. For other synchronization points, such as command-queue barriers and events, the state of the global memory should be consistent across all work-items in the kernel index space.

III.2.2 Space Allocation to Buffers

In OpenCL, the host program creates a buffer object by invoking an API function `clCreateBuffer()`. Even though the space for a buffer is allocated in the global memory of a specific device, the buffer is not bound to the compute device in OpenCL[21]. Binding a buffer and a compute device is implementation dependent. As a result, `clCreateBuffer()` has no parameter that specifies a compute device. This implies that when a buffer is created, the runtime has no information about which compute device accesses the buffer.

The SnuCL runtime does not allocate any memory space to a

buffer when the host program invokes `clCreateBuffer()` to create it. Instead, when the host program issues a memory command that manipulates the buffer or a kernel-execution command that accesses the buffer to a compute device, the runtime checks if a space is allocated to the buffer in the target device's global memory. If not, it allocates a space to the buffer in the global memory.

III.2.3 Minimizing Memory Copying Overhead

To efficiently handle buffer sharing between multiple compute devices, the SnuCL runtime maintains a *device list* for each buffer. The device list contains compute devices that have the same latest copy of the buffer in their global memory. It is empty when the buffer is created. When the command that accesses the buffer completes, the host command scheduler updates the device list of the buffer. If the buffer contents are modified by the command, it empties the list and adds the device that has the modified copy of the buffer in the list. Otherwise, it just adds in the list the device that has recently obtained a copy of the buffer because of the command.

When the host command scheduler dequeues a memory command or kernel-execution command, it checks the device list of each buffer that is accessed by the command. If the target compute device is in the device list of a buffer, the compute device has a copy of the

buffer. Otherwise, the runtime checks whether a space is allocated to the buffer in the target device's global memory. If not, the runtime allocates a space for the buffer in the global memory of the target device. Then it copies the buffer contents from a device in the device list of the buffer to the allocated space.

To minimize the memory copying overhead, the runtime selects a source device in the device list that incurs the minimum copying overhead. Figure III.2 shows an example of the memory copy time in a node (Within a GPU, Within a CPU, CPU to CPU, CPU to GPU, GPU to CPU, and GPU to GPU) or between different nodes (Node to Node) of the target cluster. We vary the buffer size from 1 MB to 512 MB.

As the source of copying, the runtime prefers a device that has a latest copy of the buffer and resides in the same node as that of the target device. If there are multiple such devices, a CPU device is preferred. When all of the potential source devices reside in other nodes, a CPU device is also preferred to a GPU device. This is because the lower-level communication API does not typically support reading directly from the GPU device memory. It costs one more copying step from the GPU device memory to a temporary space in the node main memory.

To avoid such an unnecessary memory copying overhead, we define a distance metric between compute devices as shown in Table III.2. Based on this metric, the runtime selects the nearest compute device

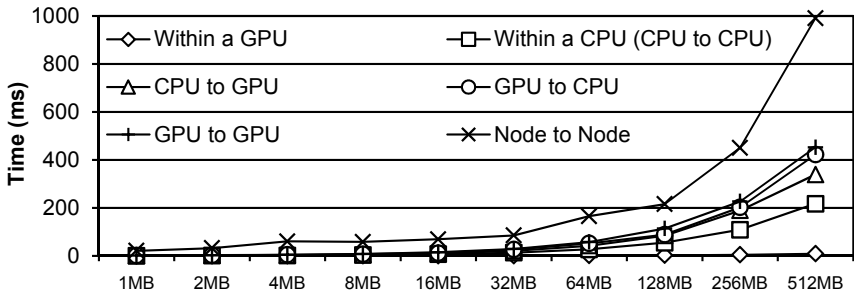

Figure III.2: Memory copy time.

Table III.2: Distance between compute devices

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

in the device list of the buffer and copies the buffer contents to the target device from the selected device.

III.2.4 Processing Memory Commands

There are three representative memory commands in OpenCL: write (`clEnqueueWriteBuffer()`), read (`clEnqueueReadBuffer()`), and copy (`clEnqueueCopyBuffer()`). When the runtime executes a write command, it copies the buffer contents from the host node's main memory to the global memory of the target device. When the run-

time executes a read command, it copies the buffer contents from the global memory of a compute device in the device list of the buffer to the host node's main memory. A CPU device is preferred to avoid the unnecessary memory copying overhead. When the runtime executes a copy command, based on the distance metric (Table III.2), it selects a nearest device in the device list of the source buffer from the target device. Then it copies the buffer contents from the global memory in the source device to the global memory in the target device.

III.2.5 Consistency Management

In OpenCL, multiple kernel-execution and memory commands can be executed simultaneously, and each of them may access a copy of the same buffer. If they update the same set of locations in the buffer, we may choose any copy as the last update for the buffer according to the OpenCL memory consistency model. However, when they update different locations in the same buffer, the case is similar to the false sharing problem that occurs in a traditional, page-level software shared virtual memory system[3].

One solution to this problem is introducing a multiple-writers protocol[3] that maintains a twin for each writer and updates the original copy of the buffer by comparing the modified copy with its twin. Each node that contains a writer device performs the comparison and sends the result (e.g., diffs) to the host who maintains the

original buffer. The host updates the original buffer with the result. However, this introduces a significant communication and computation overhead in the cluster environment if the degree of buffer sharing is high.

Instead, the SnuCL runtime solves this problem by executing kernel-execution and memory commands atomically in addition to keeping the most up-to-date copies using the device list. When the host command scheduler issues a memory command or kernel-execution command, it records the buffers that are written by the command in a list called *written-buffer list*. When the host command scheduler dequeues a command, and the command writes to any buffer in the written-buffer list, it delays issuing the command until the buffers accessed by the dequeued command are removed from the written-buffer list. This mechanism is implemented by adding the commands that write to the buffers and have not completed their execution yet into the event wait list of the dequeued command. Whenever a kernel-execution or memory command completes its execution, the host command scheduler removes the buffers written by the command from the written-buffer list.

III.2.6 Ease of Programming

Assume that a user uses a mix of MPI and OpenCL as a programming model for the heterogeneous cluster. When the user wants to launch

a kernel to an OpenCL-compliant compute device, and the kernel accesses a buffer having been written by another compute device in a different compute node, the user explicitly inserts necessary communication and data transfer operations in the MPI-OpenCL program. First, the user makes the source device copy the buffer into the main memory of its node using `clEnqueueReadBuffer()`, and sends the data to the target node using `MPI_Send()`. The target node receives the data from the source node using `MPI_Recv()`. Then, the user copies the data into the device memory by invoking `clEnqueueWriteBuffer()`. Finally, the user invokes `clEnqueueNDRangeKernel()` to execute the kernel.

On the other hand, SnuCL hides the communication and data transfer layer from the user and manages memory consistency all by itself. Thus, with SnuCL, the user executes the kernel by invoking only `clEnqueueNDRangeKernel()` without any additional data movement operations (`clEnqueueReadBuffer()`, `MPI_Send()`, `MPI_Recv()`, and `clEnqueueWriteBuffer()`). This improves software developers' productivity and increases portability.

III.3 Extensions to OpenCL

A buffer copy command (`clEnqueueCopyBuffer()`) is available in OpenCL[21]. Although this can be used for point-to-point communication in the cluster environment, OpenCL does not provide any

Table III.3: Collective communication extensions

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

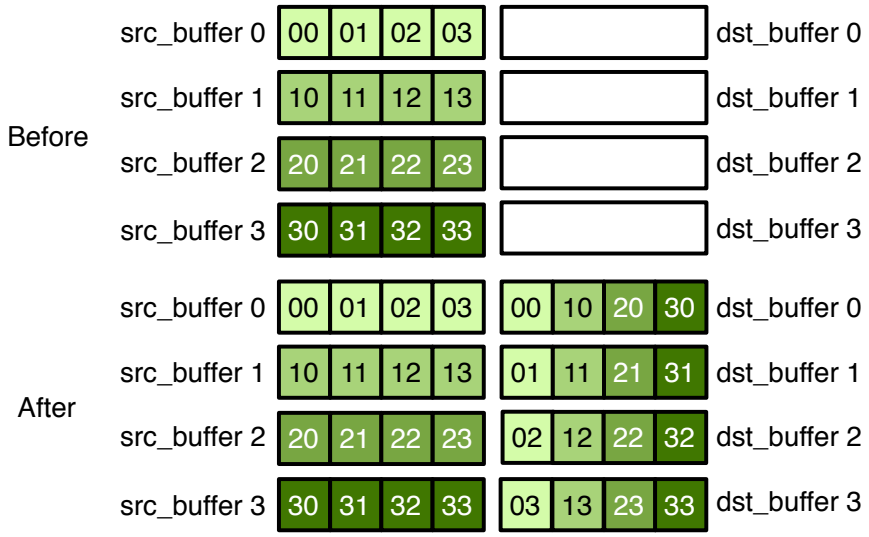

Figure III.3: `clEnqueueAlltoAllBuffer()` operation for four source buffers and four destination buffers.

collective communication mechanisms that facilitate exchanging data between many devices. SnuCL provides collective communication operations between buffers. These are similar to MPI collective communication operations. They can be efficiently implemented with the lower-level communication API or multiple `clEnqueueCopyBuffer()` commands. Table III.3 lists each collective communication operation and its MPI equivalent.

For example, the format of `clEnqueueAlltoAllBuffer()` operation is as follows:

```
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)
```

The API function `clEnqueueAlltoAllBuffer()` is similar to the MPI collective operation `MPI_Alltoall()`. The first argument `cmd_queue_list` is the list of command-queues that are associated with the compute devices where the destination buffers (`dst_buffer_list`) are located. The command is enqueued to the first command-queue in the list. The meaning of this API function is the same as enqueueing N independent `clEnqueueCopyBuffer()`s to each command-queue in `cmd_queue_list`, where N is the number

of buffers. The meaning of this operation is illustrated in Figure III.3.

III.4 Code Transformations

In this section, we describe compiler analysis and transformation techniques used in SnuCL.

III.4.1 Detecting Buffers Written by a Kernel

To keep shared buffers consistent, the SnuCL runtime performs consistency management as described in Section III.2. This requires detecting buffers that are written by an OpenCL kernel. In OpenCL, each memory object has a flag that represents its read/write permission: `CL_MEM_READ_ONLY`, `CL_MEM_WRITE_ONLY`, and `CL_MEM_READ_WRITE`. Thus, the runtime may use the read/write permission of each buffer object to obtain the necessary information. However, this may be too conservative. When the memory object has `CL_MEM_READ_WRITE` and the kernel does not write to the buffer at all, the runtime cannot detect this.

Thus, SnuCL performs a conservative pointer analysis on the kernel source when the kernel is built. A simple and conservative pointer analysis[42] is enough to obtain the necessary information because OpenCL imposes a restriction on the usage of global memory pointers used in a kernel[21]. Specifically, a pointer to address space A can

```
__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];
}
```

(a)

```
int vec_add_memory_flags[3] = {
    CL_MEM_READ_ONLY, // A
    CL_MEM_READ_ONLY, // B
    CL_MEM_WRITE_ONLY // C
};
```

(b)

```
__global__ void vec_add(float *A, float *B, float *C) {
  int id = blockDim.x * blockIdx.x + threadIdx.x;
  C[id] = A[id] + B[id];
}
```

(c)

```
#define get_global_id(N) \
  (__global_id[N] + (N == 0 ? __i : (N == 1 ? __j : __k)))

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];
      }
    }
  }
}
```

(d)

Figure III.4: (a) An OpenCL kernel. (b) The buffer access information of kernel `vec_add` for the runtime. (c) The CUDA C code generated for a GPU device. (d) The C code for a CPU device.

only be assigned to a pointer to the same address space A. Casting a pointer to address space A to a pointer to address space B ($\neq$ A) is illegal.

When the host builds a kernel by invoking `clBuildProgram()`, the SnuCL OpenCL-C-to-C translator at the host node generates the buffer access information for the runtime from the OpenCL kernel code. Figure III.4 (b) shows the information generated from the OpenCL kernel in Figure III.4 (a). It is an array of integer for each kernel. The i^{th} element of the array represents the access information of the i^{th} buffer argument of the kernel. Figure III.4 (b) indicates that the first and second buffer arguments (A and B) are read and the third buffer argument (C) is written by kernel `vec_add`. The runtime uses this information to manage buffer consistency.

III.4.2 Emulating PEs for CPU Devices

In a CPU device, the SnuCL runtime makes each CU thread emulate the PEs in the CU using a kernel transformation technique, called work-item coalescing[27] provided by the SnuCL OpenCL-C-to-C translator. The work-item coalescing technique makes the CPU core execute each work-item in the work-group one by one sequentially using a loop that iterates over the local work-item index space. The triply nested loop in Figure III.4 (d) is such a loop after the work-item coalescing technique has been applied. The size of the local

work-item index space is determined by the array `__local_size` provided by the runtime. The runtime also provides an array `__global_id` that contains the global ID of the first work-item in the work-group.

When there are work-group barriers in the kernel, the work-item coalescing technique divides the code into work-item coalescing regions (WCRs)[27]. A WCR is a maximal code region that does not contain any barrier. Since a work-item private variable whose value is defined in one WCR and used in another needs a separate location for each work-item to transfer the variable's value between different WCRs, the variable expansion technique[27] is applied to WCRs. Then, the work-item coalescing technique executes each WCR using a loop that iterates over the local work-item index space. After work-item coalescing, the execution ordering of WCRs preserves the barrier semantics.

III.4.3 Distributing the Kernel Code

When the host builds a kernel by invoking `clBuildProgram()`, the SnuCL OpenCL-C-to-CUDA-C translator (we assume that the runtime in a compute node is implemented with the CUDA runtime) generates the code for a GPU and OpenCL-C-to-C translator generates the code for a CPU device. Figure III.4 (c) and Figure III.4 (d) show the code generated for a GPU and a CPU device, respectively. Then, the host command scheduler sends a message that contains the

translated kernels to each compute node. The compute node stores the kernels in separate files and builds them with the native compiler for each compute device in the system.

Chapter IV

Distributed Execution Model for SnuCL

IV.1 Two Problems in SnuCL

This chapter addresses two problems. First, we try to solve the scalability problem of SnuCL. Second, we would like to abstract away the vendor heterogeneity in the underlying cluster architecture. When a system has multiple compute devices from different vendors, it is the programmers' burden to manage multiple OpenCL platforms from different vendors in a single OpenCL application. For example, assume that a system has three different accelerators: an AMD GPU, an NVIDIA GPU, and an Intel Xeon Phi coprocessor. In this case, the programmer needs to manage three different OpenCL platforms from three different vendors in a single application when the programmer wants to use all the accelerators in the application.

However, what makes it more difficult for the programmer is that

OpenCL objects, such as memory and event objects, cannot be shared between different platforms. Extra copy processes between them are required to share memory and event object. Furthermore, when it comes to a heterogeneous cluster, the programmer is responsible for managing memory and event objects between different platforms and different nodes. This requires additional host-side code for memory movement, event synchronization, inter-node communication, etc. In turn, the scalability of the application worsens because of the extra overhead.

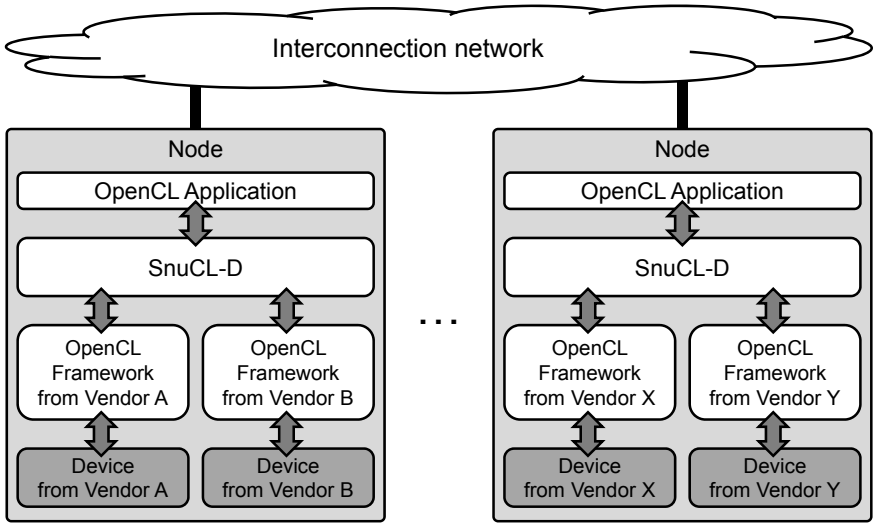

Figure IV.1: Overview of SnuCL-D.

We redesign SnuCL to overcome the above two problems. We call the redesigned framework SnuCL-D. Figure IV.1 shows an overview of SnuCL-D on a heterogeneous cluster. SnuCL-D is built on top of different OpenCL implementations for a single operating system in-

stance, and uses them to control different compute devices from different vendors in each node in the cluster.

The user executes an OpenCL application on the cluster using SnuCL-D. Then, SnuCL-D runtime executes the host program in the application on each node in the cluster. These host program instances run asynchronously and in parallel. Every node runs the same host program and follows the same execution path. However, each program instance accesses different data using different compute devices. It is a completely distributed execution mechanism and requires no centralized control. The execution process is transparent to the user and accomplished by a technique called *remote device virtualization*.

As described in Section II, the host program obtains an array of compute devices available from the target platform. It distributes the workload across the compute devices in the array and manages data between them. In SnuCL-D, the host program is written for all compute devices available in the cluster as if they were local compute devices. SnuCL-D runtime executes the host program on every node in the cluster, and it provides an illusion to the node that the node has all compute devices available in the cluster. The runtime determines which device is local to the node transparently to the host program.

In addition, SnuCL-D hides the multiple underlying OpenCL implementations from the user by integrating them into a single unified OpenCL framework on top of them. This integration relieves the pro-

grammer's burden of managing multiple OpenCL platforms and sharing OpenCL objects across different platforms and different nodes.

IV.2 Remote Device Virtualization

To make the illusion, the runtime performs the following operations. When a user launches an OpenCL application in a cluster, the runtime on each node finds out all compute devices available in the local node. Then, the runtime sends the information about the devices to a specific node. The node is specified by the user and called root node. The root node gathers the device information from all nodes in the cluster, and then it organizes an array of compute devices using the gathered information. The root node broadcasts a message that indicates the array of compute devices to all nodes in the cluster.

The runtime on each node organizes an array of compute devices using the message received from the root node. The message contains the attributes of compute devices, such as their type, name, located node, and etc. For each device in the array, the runtime obtains a real compute device from the local platform if the device is located in the local node. Otherwise, if the device is in a remote node, the runtime creates a virtual compute device using the information. All nodes organize their own array of compute devices using the same message, and thus the sequence of the compute devices is identical in all nodes. We call the virtualization technique Remote Device Virtualization.

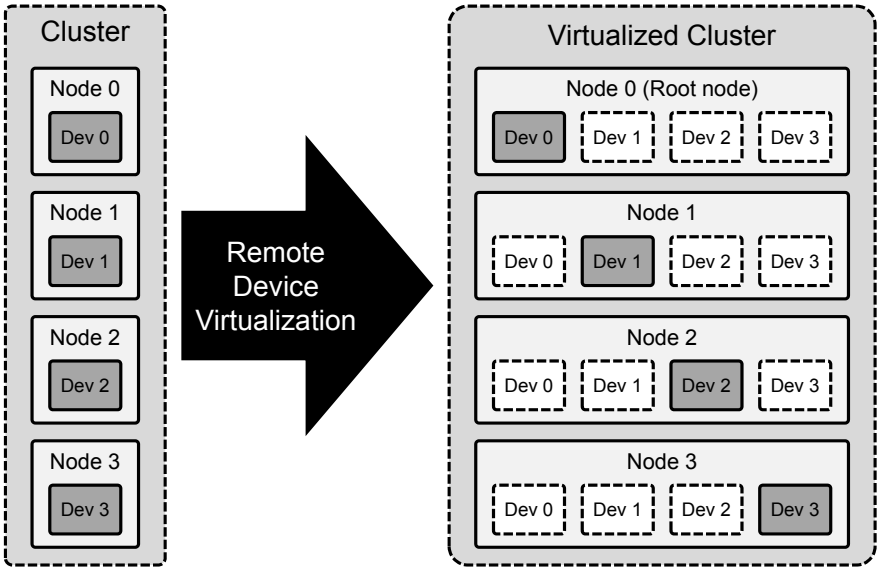

Figure IV.2: Remote Device Virtualization.

Figure IV.2 shows an example. Here is a cluster that consists of four nodes, and Node 0 is specified the root node. Each node has a single compute device, and thus there are total four compute devices in the cluster. Remote device virtualization provides an illusion to each node that the node has four compute devices. In Figure IV.2, a gray-colored device represents a real device, and a dotted-lined device represents a virtual device. For all nodes, the sequences of the compute devices are same and represented in Dev0, Dev1, Dev2, and Dev3.

When the host requests an array of compute devices available on the platform, the runtime provides a device array that consists of real and virtual devices to the host. An OpenCL operation is submitted to a command-queue in the form of an OpenCL command. A command-

queue is created and attached to a specific compute device. When a command is submitted to a command-queue that is attached to a real device, the runtime submits the command to the command-queue by calling the corresponding API function in the underlying OpenCL implementation. Otherwise, if the target command-queue is attached to a virtual device, the runtime does not submit the command. Instead, the runtime checks the OpenCL objects such as memory and event that are needed to execute the command. We describe this checking process in Section IV.3.

Finally, we show how SnuCL-D executes the OpenCL application as shown in Figure II.5 on the cluster of Figure IV.2. The host program in the application runs in every node on the cluster. For each node, when the host requests all compute devices available on the platform, the runtime returns four compute devices in a sequence of Dev0, Dev1, Dev2, and Dev3. The host creates four command-queues with the four compute devices. The host enqueues a write memory command to each command-queue. The runtime submits the command to the command-queue, only when the command-queue is associated with a real compute device. Thus, Node 0, Node 1, Node2, and Node 3 process the command that assigned to Dev0, Dev1, Dev2, and Dev3 respectively. And then, the runtime processes kernel-execution commands in the same manner as the write memory commands. Finally, the host reads memory objects from the devices by enqueuing read memory commands. For a read memory command, the runtime keeps

all nodes in the cluster have the same content in their host main memory, because all node runs the host program and they should follow the same execution path. Thus, a node that has the content of memory object broadcasts the content to other nodes. And at the same time, it receives content of the memory object that does not reside in the local node from another node. For example, in Node 2, the runtime sends the content of bufferC[2] to Node0, Node1, and Node3. And it receives the contents of bufferC[0], bufferC[1], and bufferC[3] from Node0, Node1, and Node3 respectively.

IV.2.1 Exclusive Execution on the Host

Since every node runs the host program in SnuCL-D, there are some side effects from the execution. First, the host can generate random input set data and use it. With SnuCL-D every node can make different random input sets, and thus the application may behave incorrectly or fail. Alternatively, the host writes the generated random input set in the host main memory to a single device memory, and then it can share or copy the memory object between compute devices. This programming style make the application run correctly with random generated input sets.

Second, the host may require IO operations such as writing a file. In this case, only one single host should write the file to avoid the duplicate IO operations and race conditions. SnuCL-D offers a wrapped

system library set. It wraps the system calls such as write(), and executes exclusively on the root node only. Programmer can indicate the system calls that runs exclusively in linking time.

Finally, some host codes cannot be covered with above cases. In this case, the programmer can specify the exclusive execution region in the application using our built-in global variable named IS_ROOT_NODE. SnuCL-D runtime set IS_ROOT_NODE to 1 in the root node and 0 in other nodes. The programmer moves the code that should run exclusively with an if clause using IS_ROOT_NODE. However, the application that uses IS_ROOT_NODE is not a standard OpenCL application and cannot run with other OpenCL frameworks.

IV.3 OpenCL Framework Integration

IV.3.1 OpenCL Installable Client Driver (ICD)

The OpenCL Installable Client Driver (ICD) enables multiple OpenCL implementations from different hardware vendors to coexist in a single system. With help of ICD, programmers can select between multiple OpenCL implementations at run time.

Every OpenCL object such as platform, device, and etc has a dispatch table pointer (KHRicdVendorDispatch) as shown in Figure IV.3. KHRicdVendorDispatch is a function pointer dispatch ta-

```
typedef struct KHRicdVendorDispatchRec KHRicdVendorDispatch;

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

typedef struct _cl_platform_id *    cl_platform_id;
typedef struct _cl_device_id *      cl_device_id;
...

struct _cl_platform_id
{
  KHRicdVendorDispatch *dispatch;
  ...
};

struct _cl_device_id
{
  KHRicdVendorDispatch *dispatch;
  ...
};
```

Figure IV.3: OpenCL ICD.

ble which is used to direct calls to a particular vendor OpenCL platform[20]. Almost all OpenCL implementations from various hardware vendors are implemented as dynamic shared libraries and ICD compatible. The ICD Loader in SnuCL-D loads all available OpenCL dynamic shared libraries in the system and gets the respective OpenCL platforms from them. OpenCL platform object contains KHRicdVendorDispatch, and thus SnuCL-D can call OpenCL API functions in all underlying OpenCL implementations in the system.

IV.3.2 Event Synchronization

When the host calls an OpenCL API function that enqueues a command, it can obtain an event object. The event object encapsulates the status of the enqueued command. An event object can notify the host that the associated command completes its execution. In addition, it can be used to define an ordering between commands. A user event is a special form of event object and not associated with any command's execution. A user event is created and set its status by the host.

When the host waits a command identified by an event object to complete, there are two cases to be considered. One is the node that has the device that executes the command; we call the node the source node. The other is other nodes that do not have the device. For the source node, the runtime calls the corresponding OpenCL API

function of the underlying implementation associated with the target device and waits until the completion of the command. When the runtime receives notification of completion, it broadcasts a completion message to all other nodes in the cluster through the interconnection network. For all nodes except source node, the runtime just waits for a completion message from the source node.

These processes can be achieved by the distributed execution model of SnuCL-D. As shown in Section IV.1, SnuCL-D runtime executes the same host program on every node in the cluster with an illusion that each node has all compute devices available in the cluster. Therefore, the runtime in each node can make a distinction if an event object is associated with a real device or a virtual device. If it is the former, the self-node becomes the source node. Otherwise, it identifies the source node that has the device associated with the event, and then it waits a completion message from the source node. This is a completely distributed mechanism and requires no centralized control. The distributed execution model in SnuCL-D makes all of the following techniques possible.

Every OpenCL API function that enqueues a command takes a wait list made up of event objects as an argument. It is called event wait list. The enqueued command cannot be issued for execution until all events in event wait list has completed. One restriction is that the platform associated with the target compute device and events in event wait list should be the same. Even though SnuCL-D provides a

single OpenCL platform to the users, it internally uses multiple underlying OpenCL implementations. Thus, some events in the wait list can be associated with different OpenCL platforms from that of the target compute device. The events can be from the different platforms in the local node or from a remote node.

If the target device and an event in the event wait list are associated with different platforms within a same node, then the runtime creates a user event using the platform of the target device. The runtime substitutes it with the original event in the event wait list, and it registers a callback function for the original event. The registered callback function will be called when the original event completes, and it changes the user event's status to complete.

When a command waits an event from a remote node, the runtime identifies the remote node that has a device associated with the event. We call the remote node the source node. The runtime creates a user event using the platform of the target device and substitutes it with the original event. Then the runtime calls a non-blocking receive communication API to receive a completion message from the source node. The runtime encapsulates the receive request in the request object and adds it in the request list. Each node has a single request list, and the request thread in the node repeats checking the list. When the receive communication function completes, that is, the node receives the completion message, the runtime changes the user event's status to complete. For the source node, the node identifies the target node

that has the target device associated with the command and registers a callback function for the event. The callback function will be called when the event completes and it sends a completion message to the target node.

IV.3.3 Memory Sharing

OpenCL requires that memory objects are shareable between multiple devices within a platform. On the other hand, they cannot be shared between different platforms. SnuCL-D runtime presents the programmer a single OpenCL platform, and thus the runtime makes it possible to share memory objects between multiple devices that are from different underlying vendor implementations or different nodes.

SnuCL-D runtime creates memory objects on demand. A memory object is a handle to a reference region of device memory. When the host creates a memory object, the runtime does not allocate any memory space to the memory object. Instead, the runtime allocates a space to the memory object when the memory object eventually is used such as execution for a memory command or kernel execution command. When the host enqueues a command to a command-queue, the runtime checks the memory objects needed by the command. If the target device is a real device and a memory object has not been allocated on the target device's memory, then the runtime allocates a space to the memory object in the device memory.

After checking the memory allocation, the runtime updates the contents of the memory object to maintain memory consistency. The runtime transfers contents of the memory from a source device which has the latest copy to the target device that needs the latest copy. In SnuCL-D, all nodes executes the same host program, then all nodes know that what command is executed on what device. If a command that modifies a memory object is executed on a compute device, then the device has the latest copy of the memory object and the device becomes the single source device. Otherwise, the command only reads the memory object, the device becomes one of the source devices. With this information, the runtime transfers the memory contents to the target device from one of the source devices.

There are three cases to be considered for selecting a source device. First, a source device can be associated with the same underlying platform in the same node as the target device. In that case, the runtime just enqueues the command without any additional operations. Then, the underlying OpenCL implementation transparently transfers the memory between devices.

Second, a source device can be in the same node but different underlying platform as the target device. The runtime creates a user event by using the underlying platform of the target device. The runtime enqueues a non-blocking memory read command to a command-queue associated with the source device. The memory command reads the content of the memory from the source device and writes the con-

tent to a temporary space in the node main memory. Then the runtime registers a callback function for the enqueued memory read command. The callback function will be called when the memory read command completes and it changes the user event's status to complete. Then the runtime enqueues a non-blocking memory write command to the target command-queue, and adds the user event into the memory write command's event wait list. The memory write command writes the content of the temporary space in the main memory to the device memory. After enqueueing the write memory command, the runtime enqueues the original command with a modified event wait list containing the write memory command's event.

Last, a source device can be in the different node as the target device. For this case, the runtimes in the source node and the target node transfer the contents of the memory object through the interconnection network. The source node sends data to the target node and the target node receives the data from the source node. Their requests must be matched to each other. If there is only one source device, then the node that has the source device becomes the source node and sends data to the target node. Otherwise, if there are multiple source nodes, the runtime must select one node among them.

Each node has a unique integer rank in the cluster. The target node selects a source device in the devices that have the latest copy of the memory object by using the rank. A node has the shortest distance between its node and the target node's rank is selected to the source

node. If there are more than one nodes that have the same shortest distance, the runtime selects a node has lower rank. According to this rule, the target node selects a source node and receives the memory contents from the node. On the other hand, every node except the target node in the cluster checks whether oneself is the source node or not. When the host enqueues a command to a command-queue associated with a virtual device, the runtime checks the latest copy of the memory objects needed in the command are in the local node. If then, the runtime calculates all node distance of source nodes and checks the local node has the lowest distance.

In order to minimize the memory copying overhead, the runtime selects a source device that incurs the minimum copying overhead. For above three cases, the runtime prefers preceding cases.

Chapter V

Experimental Results

V.1 SnuCL Evaluation

This section describes the evaluation methodology and results for SnuCL.

Table V.1: The target clusters

	Host node	**Compute node**	
Processors	2 × Intel Xeon X5680	2 × Intel Xeon X5660	4 × 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		

Cluster A **(a 10-node heterogeneous CPU/GPU cluster)**

	Host node	**Compute node**
Processors	2 × Intel Xeon X5570	2 × 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	

Cluster B **(a 257-node CPU cluster)**

V.1.1 Methodology

Target cluster architecture. We evaluate SnuCL using two cluster systems (Cluster A and Cluster B). Table V.3 summarizes the target clusters.

Table V.2: Applications used

Application	Source	Description	Input	Global memory size (MB)	Extension 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 (162x162x162) or Class D (408x408x408)	1982.1 or 30686.7	
CG	NAS	Conjugate gradient	Class C (150000) or Class D (1500000)	1102.6 or 20399.1	
CP	Parboil	Coulombic potential	16384x16384, 10000 atoms	4.1	
EP	NAS	Embarrassingly parallel	Class D (2^36)	0.8	
FT	NAS	3-D FFT PDE	Class B (512x256x256) or Class C (512x512x512)	2816.0 or 11264.0	AlltoAll
MatrixMul	NVIDIA	Matrix multiplication	10752x10752 or 16384x16384	1323.0 or 3072.0	Broadcast
MG	NAS	Multigrid	Class C (512x512x512) or Class D (1024x1024x1024)	3575.3 or 28343.7	
Nbody	NVIDIA	N-Body simulation	1048576 bodies	64.0	
SP	NAS	Pentadiagonal solver	Class C (162x162x162) or Class D (408x408x408)	1477.9 or 19974.4	

Benchmark applications. We use eleven OpenCL applications from various sources: AMD[2], NAS[34], NVIDIA[35], Parboil[43], and PARSEC[5]. The characteristics of the applications and their input sets are summarized in Table V.2. The applications from Parboil and PARSEC are translated to OpenCL applications manually. The applications from NAS are from SNU NPB Suite[39] that contains OpenCL versions of the original NAS Parallel Benchmarks for multiple OpenCL compute devices. For an OpenCL application written for a single compute device, we modify the application to distribute workload across multiple compute devices available. Especially, FT and MatrixMul use SnuCL collective communication extensions to OpenCL. Some applications are evaluated with two input sets. The smaller input set is used for Cluster A because a GPU has relatively small device memory and allows only the smaller input set for those applications. The larger

input set is used for Cluster B to show its scalability in a large-scale cluster. All applications are portable across CPU and GPU devices. That is, we can run the applications either on CPU devices or GPU devices without any source code modification.

Runtime and source-to-source translators. We have implemented the SnuCL runtime and source-to-source translators. The SnuCL runtime uses Open MPI 1.4.1 as the lower-level communication API. The GPU part of the runtime is implemented with CUDA Toolkit 4.0[35]. We have implemented SnuCL source-to-source translators by modifying clang that is a C front-end for the LLVM[26] compiler infrastructure. The runtime uses Intel ICC 11.1[17], and NVIDIA's NVCC 4.0[35] to compile the translated kernels for CPU devices and GPU devices, respectively.

V.1.2 Results

Figure V.1 shows the speedup (over a single CPU core) of each application with SnuCL when we use only CPU devices in Cluster A. The sequential CPU version of each application is obtained from the same source (Table V.2). Each application from NAS is shown with its input set. The CPUs in the compute nodes support simultaneous multithreading (SMT) that enables two logical cores per physical core. Thus, each compute node contains 24 logical CPU cores. In each compute node, two logical CPU cores are dedicated to the command

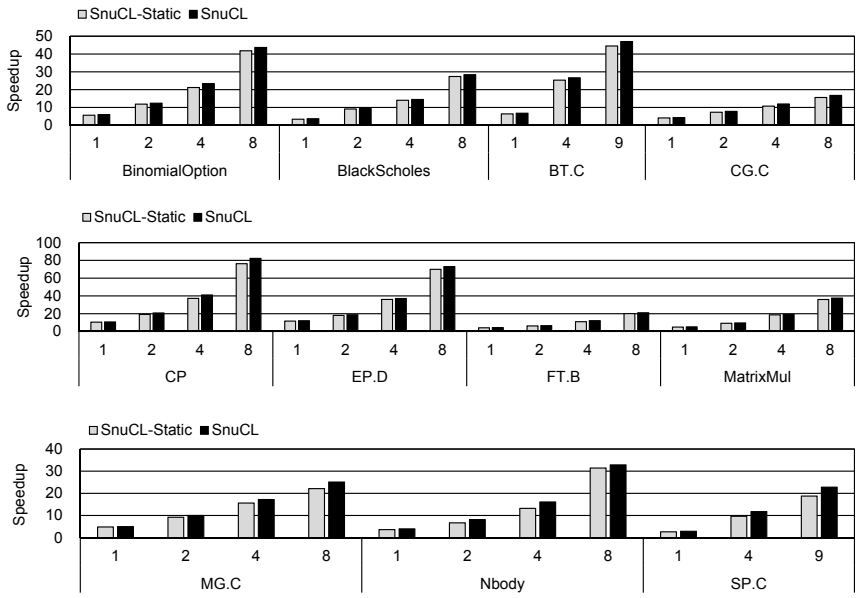

Figure V.1: Speedup over a single CPU core using CPU devices on Cluster A. The numbers on x-axis represent the number of CPU compute devices.

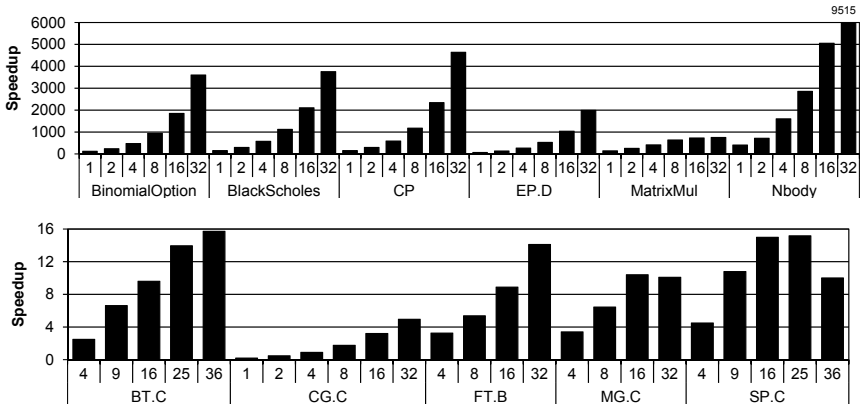

Figure V.2: Speedup over a single CPU core using GPU devices on Cluster A. The numbers on x-axis represent the number of GPU compute devices.

handler and the CPU device thread. The remaining 22 logical CPU cores are configured as a CPU device. We vary the number of compute nodes (i.e., the total number of CPU devices in the cluster) from 1 to 8 in powers of two for all applications but BT and SP. We set the number of CPU devices to square numbers (1, 4, 9) for BT and SP because of their algorithms.

We also implement another SnuCL runtime (SnuCL-Static) that exploits a static scheduling algorithm (conventional block scheduling) for the kernel workload distribution for CPU compute devices. The device thread divides the entire work-groups into sets of $\lceil N/P \rceil$ work-groups, where N is the number of work-groups and P is the number of CU threads in the CPU device. The SnuCL runtime that uses the dynamic scheduling algorithm described in Section III.1.3 is denoted by SnuCL in Figure V.1.

All applications scale well in Figure V.1. Our dynamic scheduling algorithm is quite effective. Static scheduling mechanisms used in SnuCL-Static ignore workload imbalance that occurs at run time due to variations in CPU cores. In addition, when the total workload is not evenly divisible for all CPU cores, some CPU cores are not fully utilized resulting in load imbalance. Our dynamic scheduling mechanism solves this load imbalancing problem.

Figure V.2 shows the speedup (over a single CPU core) when we use only GPU devices in Cluster A. Each compute node contains four

GPU devices. We vary the number of GPU devices in the cluster from 1 to 36 in powers of two or square numbers. For BT, FT, MG, and SP, we cannot use fewer than four GPUs because of their memory requirement. Note that we use the same OpenCL application source code for both CPU devices and GPU devices.

When we use only GPU devices in the cluster, all applications but MatrixMul, MG and SP scale well. Since the communication overhead due to data movement (e.g., read/write and copy operations of buffers) dominates the performance of MatrixMul and SP, they do not scale well. The speedup of MG at 32 GPU devices is smaller than that at 16 GPU devices because its index space is not large enough to fully utilize all the 32 GPU devices and the communication overhead increases at 32 GPU devices.

When an application has enough data parallelism, its performance with GPU devices is better than that with CPU devices. On the other hand, the cost of data transfer between GPU devices is higher than that between CPU devices because of extra data transfer between the node main memory and the GPU's device memory via the PCI-E bus. We see that applications with a low communication-to-computation ratio scales better in our cluster environment. A GPU device executes the kernel in MatrixMul 30 times faster than a CPU device but the GPU device has 30% longer data transfer time than the CPU device. With one GPU device, MatrixMul has a communication-to-computation ratio of 13%. As the number of GPU devices increases

to 8 and 32, the ratio increases to 44% and 257%, respectively. On the other hand, a CPU device has 0.4% communication-to-computation ratio due to its lower communication overhead and slower computation than a GPU device. When the number of CPU devices increases to 8, the ratio increases to only 4.6%. This is the reason why MatrixMul scales better with CPU devices than with GPU devices.

Performance portability. In Figure V.2, with GPU devices, the speedup of BT, CG, FT, MG and SP is three orders of magnitude smaller than that of other applications. These applications are from the NAS Parallel Benchmark suite that is originally targeting CPU systems, and note that we use the same OpenCL source code for both CPU devices and GPU devices. Since performance tuning factors, such as data placement, memory access patterns (e.g., non-coalesced memory accesses), the number of work-groups in the kernel index space, the work-group size (the number of work-items in a work-group), compute-device-specific algorithms, etc., between CPU devices and GPU devices are significantly different, an optimization for one type of device may not perform well on another type of device[31].

The kernels in BT, CG, FT, MG, and SP make the GPU devices suffer from non-coalesced memory accesses. Furthermore, they have many buffer-copy memory commands. The data transfer cost between GPU devices is much higher than that between CPU devices. Since the kernels in CG and MG have small work-group sizes that are not big enough to make all scalar processors (PEs) of a streaming multi-

processor (CU) in GPU devices busy, resulting in poor performance. On the other hand, for CPU devices, each CPU core (CU) emulates the PEs. It executes each work-item in a work-group one by one sequentially using the work-item coalescing technique. Thus, the small work-group size does not affect the performance of the CPU devices.

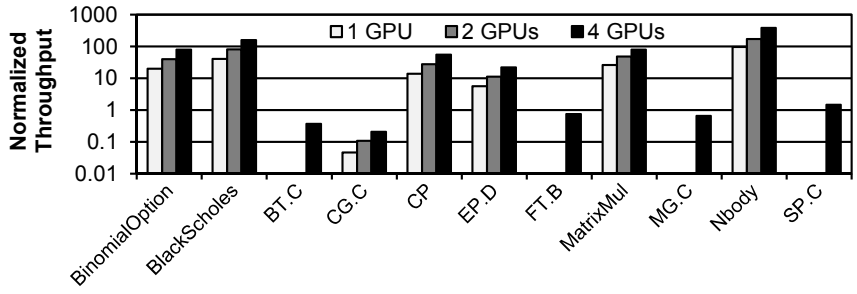

Figure V.3: Normalized throughput of GPU devices over a CPU device.

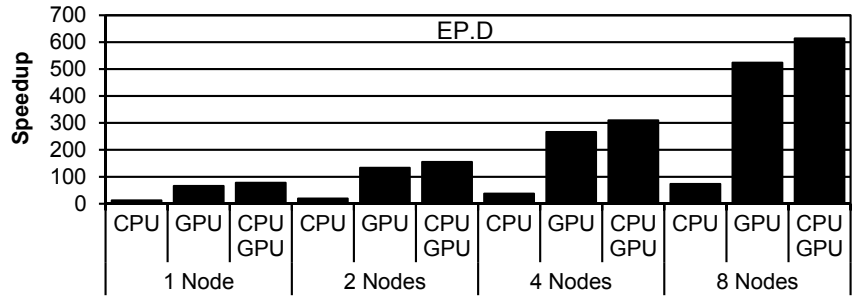

Figure V.4: Exploiting both CPU and GPU devices for EP.

Exploiting both types of devices. Figure V.3 shows the normalized throughput (CPU execution time divided by GPU execution time) of one, two, and four GPU devices over a single CPU device within the same compute node for each application. The y-axis is in the logarithmic scale. BT, FT, MG, and SP have no throughput at

one and two GPU devices because of their memory requirement.

An application that has similar performance between a CPU device and a GPU device can profit from exploiting both CPU devices and GPU devices in the cluster because our OpenCL implementation of the application is portable across both types of devices. If the user wishes more than 10 percent performance improvement using both types of devices, the normalized throughput between the faster device and slower device should be less than nine. However, one restriction is that the application should allow changing the number of devices used and the amount of workload distributed to each device.

Among those applications, only EP satisfies the condition. We distribute its workload between CPU devices and GPU devices based on the throughput. Figure V.4 shows the speedup of EP when both types of devices are used. We vary the number of compute nodes from 1 to 8. Only one GPU device within each compute node (including one CPU device) is used for evaluation to manifest the difference. As we expected from the throughput, compared to the case of GPU devices only, the performance improvement of EP is 11.4% on average in Figure V.4.

Scalability. To show the scalability of SnuCL, Figure V.5 shows the speedups of all applications on Cluster B (a 257-node homogeneous cluster). For the applications from the NAS Parallel Benchmark (NPB) suite, it compares our OpenCL implementations with the un-

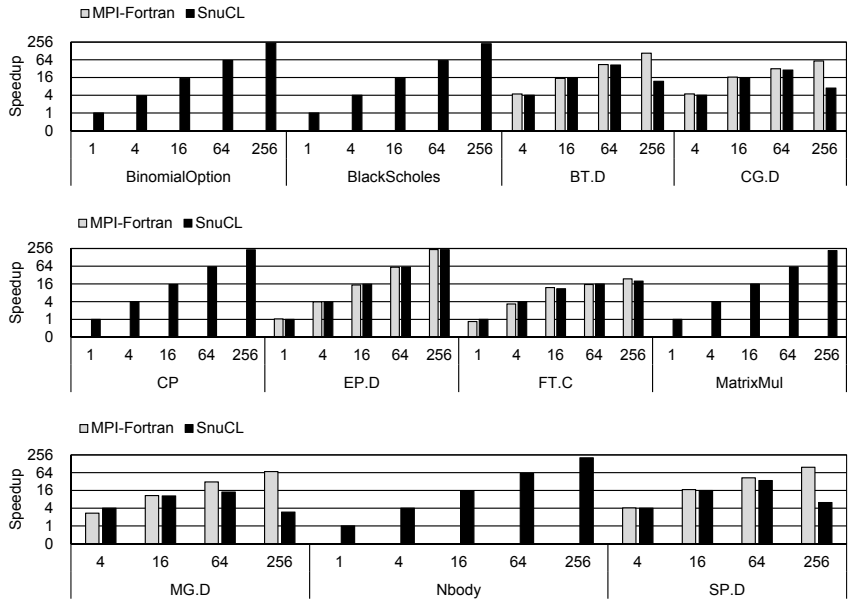

Figure V.5: Speedup over a single node (a CPU compute device with four CPU cores) on Cluster B. The numbers on x-axis represent the number of nodes (CPU compute devices).

modified original MPI-Fortran versions (MPI-Fortran) from NPB. We build the MPI-Fortran applications using Intel IFORT 11.1. Since BT and SP require the number of MPI tasks to be a square number while CG, FT, and MG require the number of tasks to be a power of two, we run 4 MPI processes per node for all applications (the Hyper-Threading mechanism is disabled for the CPU cores of Cluster B). For fair comparison, the SnuCL runtime configures a CPU device with 4 CPU cores per compute node. We vary the number of compute nodes from 1 to 256 in powers of four on x-axis. Y-axis shows the speedup in logarithmic scale over the OpenCL version on a single compute node.

All OpenCL applications scale well up to 64 nodes. The OpenCL applications from NPB show competitive performance with MPI versions. However, when the number of compute nodes increases to 256, some OpenCL applications from NPB show poor performance while MPI-Fortran versions still show good scalability. BinomialOption, BlackScholes, CP, EP, FT, MatrixMul, and Nbody still scale well on 256 nodes. They have a small number of commands that take long time to execute. Their communication-to-computation ratios are small and their scheduling overhead is negligible.

On the other hand, BT, CG, MG, and SP show performance degradation on 256 nodes. They have a large number of commands that take very short time to execute. For example, SP has the largest number of commands to be executed among the applications. Total 90,234,368 commands are enqueued and executed (with the Class D input) on

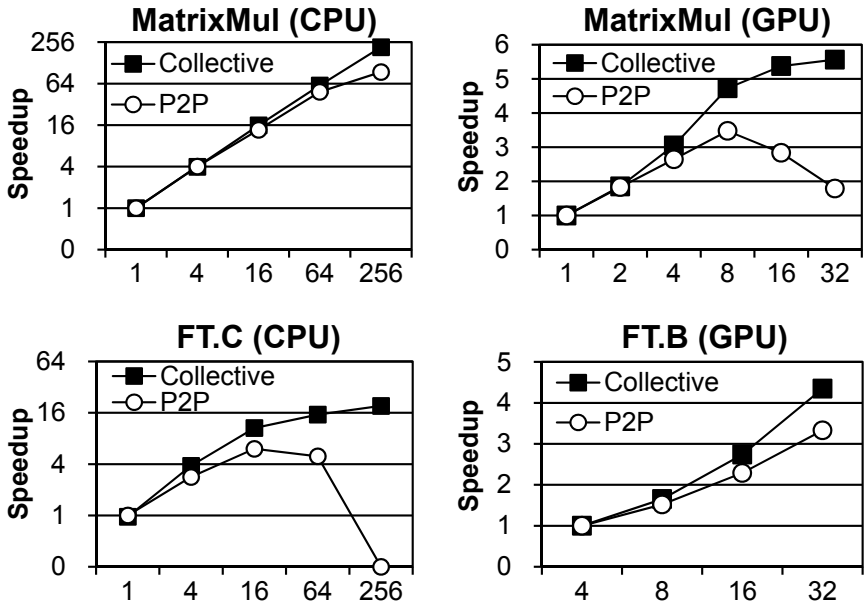

Figure V.6: The performance of collective communication extensions. X-axis shows the number of compute devices.

256 compute nodes while its total execution time is 1,813 seconds. This means that the host command scheduler schedules about 50,000 commands in a second. As the number of nodes increases, workload to be executed on each compute device decreases. However, the idle time of a compute device increases because command scheduling is centralized to a single host node and it takes time for the host node to schedule a new command when there are many nodes in the cluster. This makes compute devices less efficient, resulting in overall performance degradation on 256 nodes.

Collective communication extensions. The collective communication APIs in SnuCL are implemented with MPI collective op-

erations. In addition, we use a depth tree to implement the broadcasting mechanism[41] for GPU devices, rather than sending data directly from the source to the destination. Among the applications, MatrixMul uses `clEnqueueBroadcastBuffer()` and FT uses `clEnqueueAlltoAllBuffer()`. To compare performance, we implement another version (P2P) that uses `clEnqueueCopyBuffer()` instead of using the extensions. We evaluate the performance using Cluster A for GPUs and Cluster B for CPUs. Figure V.6 shows the performance of SnuCL collective communication extensions to OpenCL (Collective). We see that Collective achieves much better performance than P2P as the number of compute devices increases.

As described in Section III.3, `clEnqueueAlltoAllBuffer()` has an equivalent meaning of performing N independent `clEnqueueCopyBuffer()` to each device, where N is the number of buffers. To execute N independent commands concurrently, either the command queue should be out-of-order type or there should be N command queues per compute device. This makes the OpenCL program more complex and increases the scheduling overhead. Thus, SnuCL collective communication extensions provide the programmer with both high performance and ease of programming in the cluster environment.

V.2 SnuCL-D Evaluation

This section presents the evaluation methodology and results for SnuCL-D.

Table V.3: The target clusters.

	Node
Processors	2 × Intel Xeon X5570
Clock frequency	2.93GHz
Number of cores	4
Memory size	24GB
Quantity	256
OS	Red Hat Enterprise Linux Server 5.3
Interconnection	Mellanox InfiniBand QDR

Cluster A (CPU Cluster)

	Node		
Processors	2 × Intel Xeon E5-2650	Intel Xeon Phi 5110P	NVIDIA GTX 580
Clock frequency	2.0GHz	1.0GHz	1.5GHz
Number of cores	8	60	512
Memory size	128GB	8GB	1.5GB
Quantity	4		
OS	Red Hat Enterprise Linux Server 6.3		
Interconnection	Mellanox InfiniBand QDR		

Cluster B (Heterogeneous CPU/Phi/GPU Cluster)

V.2.1 Methodology

Target cluster architecture. We evaluate the performance of SnuCL-D using two cluster systems (Cluster A and Cluster B). Table V.3 summarizes the target clusters.

Underlying OpenCL Implementations. SnuCL-D uses the underlying OpenCL implementations. We use AMD OpenCL SDK for CPU devices in Cluster A. In Cluster B, we use Intel OpenCL SDK

for CPU devices and Intel Xeon Phi coprocessors, and use NVIDIA OpenCL SDK for GPU devices. SnuCL-D uses Open MPI 1.4.2 in Cluster A, and Open MPI 1.6.4 in Cluster B as the lower-level communication API.

Application	Input	Global memory size (MB)
BinomialOption	2097152 samples, 512 steps, 100 iterations	64.0
BlackScholes	33538048 options, 100 iterations	895.6
BT	Class D (408x408x408)	30686.7
CG	Class D (1500000) or Class E (9000000)	20399.1 or 151536.2
CP	16384x16384, 10000 atoms	4.1
EP	Class D (2^36)	0.8
FT	Class C (512x512x512)	11264.0
MatrixMul	16384x16384	3072.0
MG	Class D (1024x1024x1024) or Class E (2048x2048x2048)	28343.7 or 226749.6
Nbody	1048576 bodies	64.0
SP	Class D (408x408x408)	19974.4

Table V.4: Applications used.

Benchmark Applications. We use eleven OpenCL applications from various sources: AMD[2], NAS[34], NVIDIA[35], Parboil[43], and PARSEC[5]. The characteristics of the applications and their input sets are summarized in Table V.4.

V.2.2 Results

Scalability. Figure V.7 shows the speedup of all applications on Cluster A (a 256-node cluster) to show the scalability of our approach. We compare SnuCL-D with SnuCL[24]. SnuCL is another OpenCL framework for clusters. Unlike SnuCL-D, SnuCL adopts the master/slave execution model based on the original OpenCL execution model. The

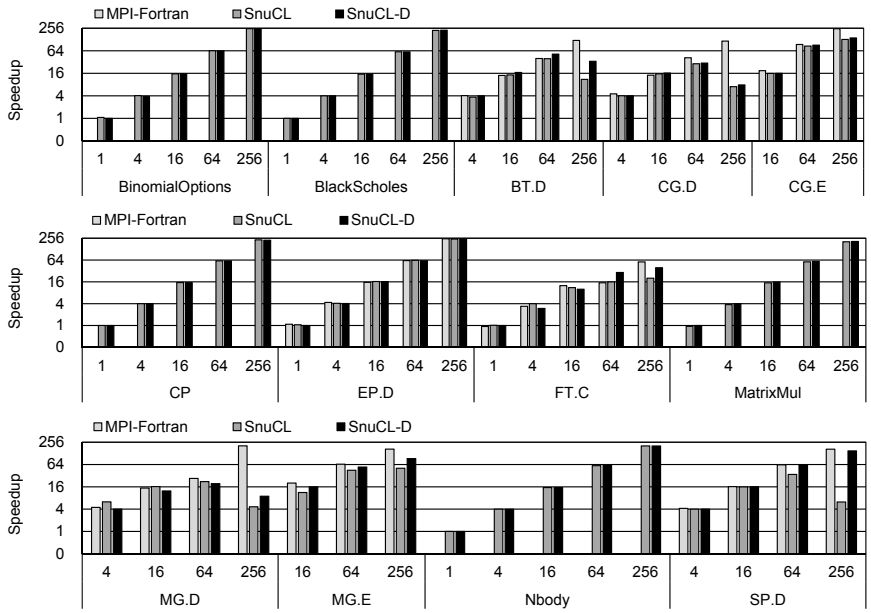

Figure V.7: Speedup over a single CPU device on Cluster A.

target cluster architecture of SnuCL consists of a single host node and multiple compute nodes. The host node executes the host program in an OpenCL application and manages all compute nodes in the cluster. The compute nodes execute the kernel programs in the application.

For the applications from the NAS Parallel Benchmark (NPB) suite, it compares our OpenCL implementations with the unmodified original MPI_Fortran versions (MPI-Fortran) from NPB. Since SP requires the number of MPI tasks to be a square number, we run four MPI processes per node. For fair comparison, the runtimes of SnuCL and SnuCL-D configure a CPU device with four CPU cores per node. We vary the number of nodes from 1 to 256 in powers of four on X-

axis. Y-axis shows the speedup in logarithmic scale over SnuCL-D on a single node.

BinomialOption, BlackScholes, CP, EP.D, FT, C, MatrixMul, Nbody show similar performance and scale well up to 1024 nodes in both of SnuCL and SnuCL-D. The applications have a small number of commands that take long time to execute. The runtime overhead of SnuCL and SnuCL-D are negligible, and thus they show good scalability.

For the memory limitation, SP.D requires minimum four nodes to run. To show scalability, we assume that the speedup with four nodes is four times than a single node. SP.D have a large number of commands that take very short time to execute. In SnuCL, command scheduling is centralized to a single host node and it takes time for the host to schedule a new command when there are many nodes in the cluster. This makes compute devices less efficient, resulting in overall performance degradation. SnuCL-D scales up to 256 nodes while SnuCL does not. With 256 nodes, SnuCL-D shows 24 times better performance than SnuCL, and 10 percentage worse performance than MPI-Fortran.

Runtime overhead. Figure V.8 shows the runtime overhead of SnuCL and SnuCL-D for all applications on Cluster A with 256 nodes. The runtime overhead for BinomialOptions, BlackScholes, CP, EP.D, MatrixMul, and Nbody are negligible. These applications have a small

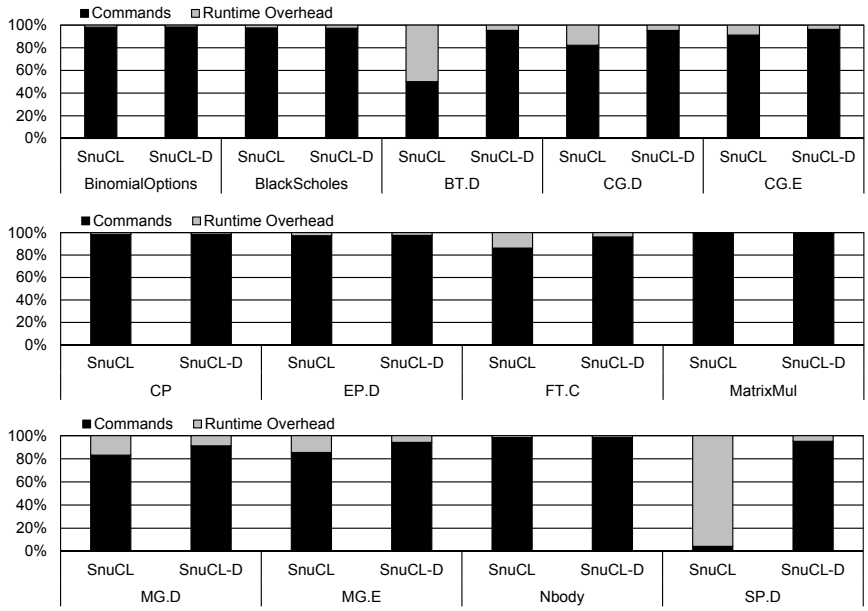

Figure V.8: Runtime overhead on Cluster A.

number of commands that take long time to execute.

On the other hand, BT.D, CG.D, CG.E, FT.C, MG.D, MG.E, and SP.D have a large number of commands that take very short time to execute. The main overhead in SnuCL runtime is inter-node communication between host node and compute nodes to transfer command messages and completion messages. Because SnuCL is designed to centralized execution model, as the number of commands increases, the host node becomes the communication bottleneck. For example, SP.D schedules about 50,000 commands in a second. The scheduling command overhead in SP.D takes about 95.7% in its execution time. The main overhead in SnuCL-D runtime is checking the event objects

and memory objects in handling commands for remote devices. As the number of commands increases, these overhead increases too. SnuCL-D is designed in completely distributed execution model and checking the events and memories for remote devices is negligible. SnuCL-D shows only 4.9% runtime overhead of its total execution time in SP.D with 256 nodes.

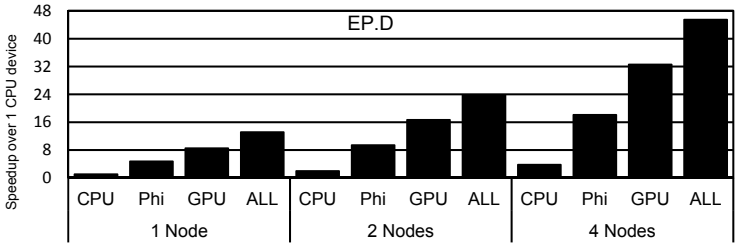

Figure V.9: Exploiting CPU, Phi and GPU devices for EP on Cluster B.

Unified OpenCL framework. Cluster B is a heterogeneous cluster that contains CPUs, GPUs and Phi coprocessors. Two OpenCL implementations are installed in each node in Cluster B. Intel OpenCL SDK for CPUs and Phi coprocessors, and NVIDIA OpenCL SDK for GPUs. Figure V.9 shows the speedup (over a single CPU device) of EP when CPUs only, Phi coprocessors only, GPUs only, and all of them are used. We distribute its workload across CPUs, GPUs, and Phi coprocessors based on their throughput. We vary the number of nodes from 1 to 4. Note that we use only single OpenCL platform presented by SnuCL-D. This alleviates the programmers from the burden of managing multiple OpenCL implementations across different platforms and moreover different nodes.

Chapter VI

Conclusions and Future Directions

VI.1 Conclusions

In this thesis, we introduce the design and implementation of SnuCL that provides a system image running a single operating system instance for heterogeneous CPU/GPU clusters to the programmer. It allows the OpenCL application to utilize compute devices in a remote compute node as if they were in the host node. The user launches a kernel to any compute device in the cluster and manipulates memory objects using standard OpenCL API functions. Our work shows that OpenCL can be a unified programming model for heterogeneous CPU/GPU clusters. Moreover, our collective communication extensions to standard OpenCL facilitate ease of programming. SnuCL enables OpenCL applications written for a single node to run on the cluster that consists of multiple such systems without any modification. It

also makes the application portable not only between heterogeneous devices in a single node, but also between all heterogeneous devices in the cluster environment. The experimental result indicates that SnuCL achieves high performance, ease of programming, and scalability for medium-scale clusters. For large scale clusters, SnuCL may lead to performance degradation due to its centralized task scheduling model.

To overcome this limitation, we have designed and implemented SnuCL-D, a scalable and unified OpenCL framework for heterogeneous clusters. The remote device virtualization technique provides an illusion to the node that the node has all compute devices available in the cluster. SnuCL-D executes the host program in an OpenCL application on every node in the cluster using the illusion. SnuCL-D integrates multiple underlying OpenCL implementations in the system into a single OpenCL programming environment. This relieves the programmer from the burden of managing multiple OpenCL implementations across different platforms and different nodes. The experimental results indicates that SnuCL-D achieves high performance, ease of programming, and good scalability. To the best of our knowledge, this is the first work that runs OpenCL applications in a completely distributed fashion on the clusters and integrates underlying OpenCL implementations across multiple nodes on the clusters.

VI.2 Future Directions

While heterogeneous cluster systems with multiple compute devices are widening their user base, the users still manually write code from scratch to exploit the multiple compute devices available in the system. In addition, converting an application written for a single compute device to run on multiple compute devices generally requires rewriting the code, and sometimes fairly extensive modifications are required. The programmer needs to insert code for distributing workload across multiple compute devices and managing data between the host main memory and multiple compute device memories. Guaranteeing consistency between the multiple copies of data makes this process more difficult for the programmer.

An interesting future direction is to introduce an OpenCL framework that provides an illusion of a single compute device to the programmer for the multiple compute devices available in the heterogeneous cluster. It enables OpenCL applications written for a single compute device to run on the system with multiple compute devices without any modification.

Bibliography

[1] *TOP500 Supercomputing Sites*, 2010. `http://www.top500.org`.

[2] AMD. *Accelerated Parallel Processing (APP) SDK*, 2012. `http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk`.

[3] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *Computer*, 29:18–28, February 1996.

[4] Amnon Barak, Tal Ben-nun, Ely Levy, and Amnon Shiloh. A Package for OpenCL Based Heterogeneous Computing on Clusters with Many GPU Devices. In *Proceedings of the Workshop on Parallel Programming and Applications on Accelerator Clusters*, PPAAC '10, 2010.

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 72–81, 2008.

[6] Bruce J. Walker. Open Single System Image (openSSI) Linux Cluster Project. `http://www.openssi.org/ssi-intro.pdf`.

[7] Li Chen, Lei Liu, Shenglin Tang, Lei Huang, Zheng Jing, Shixiong Xu, Dingfei Zhang, and Baojiang Shou. Unified parallel C for

GPU clusters: language extensions and compiler implementation. In *Proceedings of the 23rd international conference on Languages and compilers for parallel computing*, LCPC'10, pages 151–165, 2011.

[8] Yifeng Chen, Xiang Cui, and Hong Mei. Large-scale FFT on GPU clusters. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 315–324, 2010.

[9] Frederica Darema. The SPMD Model: Past, Present and Future. *Lecture Notes in Computer Science*, 2131(1):1–1, January 2001.

[10] J. Duato, A.J. Pena, F. Silla, R. Mayo, and E.S. Quintana-Orti. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *Proceedings of the International Conference on High Performance Computing and Simulation* , HPCS '11, pages 224 –231, 28 2010-july 2 2010.

[11] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. GPU Cluster for High Performance Computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, pages 47–58, 2004.

[12] Massimiliano Fatica. Accelerating linpack with cuda on heterogenous clusters. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 46–51, 2009.

[13] Michael Garland, Manjunath Kudlur, and Yili Zheng. Designing a unified programming model for heterogeneous machines. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 67:1–67:11, 2012.

[14] Jayanth Gummaraju, Laurent Morichetti, Michael Houston, Ben Sander, Benedict R. Gaster, and Bixia Zheng. Twin peaks: a software platform for heterogeneous computing on general-purpose

and graphics processors. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 205–216, 2010.

[15] IBM. *OpenCL - The open standard for parallel programming of heterogeneous systems*, 2011. `http://www.alphaworks.ibm.com/tech/opencl`.

[16] IBM. *OpenCL Common Runtime for Linux on x86 Architecture*, 2011. `http://www.alphaworks.ibm.com/tech/opencl`.

[17] Intel. Intel Composer XE 2011 for Linux. `http://software.intel.com/en-us/articles/intel-composer-xe`.

[18] Intel. *The Intel SDK for OpenCL Applications*, 2012. `http://software.intel.com/en-us/vcsource/tools/opencl-sdk`.

[19] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 10–10, 1994.

[20] The Khronos Group. *OpenCL Installable Client Driver (ICD)*, 2010. `http://www.khronos.org/registry/cl/extensions/khr/cl_khr_icd.txt`.

[21] Khronos OpenCL Working Group. *OpenCL - The open standard for parallel programming of heterogeneous systems*, 2011. `http://www.khronos.org/opencl`.

[22] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. Achieving a single compute device image in OpenCL for multiple GPUs. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 277–288, 2011.

[23] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. OpenCL as a Programming Model for GPU Clusters. In *Proceedings of the 24th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '11, 2011.

[24] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 341–352, 2012.

[25] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010.

[26] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–86, 2004.

[27] 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. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 193–204, 2010.

[28] Jun Lee, Jungwon Kim, Junghyun Kim, Sangmin Seo, and Jaejin Lee. An OpenCL Framework for Homogeneous Manycores with no Hardware Cache Coherence. In *Proceedings of the 20th international conference on Parallel architectures and compilation techniques*, PACT '11, 2011.

[29] Seyong Lee and Rudolf Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, 2010.

[30] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 101–110, 2009.

[31] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 451–460, 2010.

[32] Hui Li, Sudarsan Tandri, Michael Stumm, and Kenneth C. Sevcik. Locality and Loop Scheduling on NUMA Multiprocessors. In *Proceedings of the 1993 International Conference on Parallel Processing - Volume 02*, ICPP '93, pages 140–147, 1993.

[33] Morin, C. and Lottiaux, R. and Vallee, G. and Gallard, P. and Margery, D. and Berthou, J.-Y. and Scherson, I. D. Kerrighed and data parallelism: cluster computing on single system image operating systems. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, CLUSTER '04, pages 277–286, 2004.

[34] NASA Advanced Supercomputing Division. *NAS Parallel Benchmarks version 3.3.* `http://www.nas.nasa.gov/Resources/Software/npb.html`.

[35] NVIDIA. NVIDIA CUDA Toolkit 4.0. `http://developer.nvidia.com/cuda-toolkit-40`.

[36] NVIDIA. *NVIDIA CUDA C Programming Guide 4.0*, 2011.

[37] NVIDIA. *NVIDIA OpenCL*, 2011. `http://developer.nvidia.com/opencl`.

[38] James C. Phillips, John E. Stone, and Klaus Schulten. Adapting a message-driven parallel application to GPU-accelerated clusters. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 8:1–8:9, 2008.

[39] Sangmin Seo, Gangwon Jo, and Jaejin Lee. Performance Characterization of the NAS Parallel Benchmarks in OpenCL. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, IISWC '11, 2011.

[40] Seoul National University and Samsung. *SNU-SAMSUNG OpenCL Framework*, 2010. `http://opencl.snu.ac.kr`.

[41] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.

[42] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 32–41, 1996.

[43] The IMPACT Research Group. *Parboil Benchmark suite*. `http://impact.crhc.illinois.edu/parboil.php`.

[44] Canqun Yang, Feng Wang, Yunfei Du, Juan Chen, Jie Liu, Huizhan Yi, and Kai Lu. Adaptive Optimization for Petascale Heterogeneous CPU/GPU Computing. In *Proceedings of IEEE*

International Conference on Cluster Computing, IEEE Cluster '10, pages 19–28, 2010.

초 록

OpenCL은 이종 컴퓨팅 시스템의 다양한 계산 장치를 위한 통합 프로그래밍 모델이다. OpenCL은 다양한 이기종의 계산 장치에 대한 공통된 하드웨어 추상화 레이어를 프로그래머에게 제공한다. 프로그래머가 이 추상화 레이어를 타깃으로 OpenCL 어플리케이션을 작성하면, 이 어플리케이션은 OpenCL을 지원하는 모든 하드웨어에서 실행 가능하다. 하지만 현재 OpenCL은 단일 운영체제 시스템을 위한 프로그래밍 모델로 한정된다. 프로그래머가 명시적으로 MPI와 같은 통신 라이브러리를 사용하지 않으면 OpenCL 어플리케이션은 복수개의 노드로 이루어진 클러스터에서 동작하지 않는다. 요즘 들어 여러 개의 멀티코어 CPU와 가속기를 갖춘 이종 클러스터는 그 사용자 기반을 넓혀가고 있다. 이에 해당 이종 클러스터를 타깃으로 프로그래밍하기 위해서는 프로그래머는 MPI-OpenCL 같이 여러 프로그래밍 모델을 혼합하여 어플리케이션을 작성해야 한다. 이는 어플리케이션을 복잡하게 만들어 유지보수가 어렵게 되며 이식성이 낮아진다.

본 논문에서는 이종 클러스터를 위한 OpenCL 프레임워크, SnuCL을 제안한다. 본 논문은 OpenCL 모델이 이종 클러스터 프로그래밍 환경에 적합하다는 것을 보인다. 이와 동시에 SnuCL이 고성능과 쉬운 프로그래밍을 동시에 달성할 수 있음을 보인다. SnuCL은 타깃 이종

클러스터에 대해서 단일 운영체제가 돌아가는 하나의 시스템 이미지를 사용자에게 제공한다. OpenCL 어플리케이션은 클러스터의 모든 계산 노드에 존재하는 모든 계산 장치가 호스트 노드에 있다는 환상을 갖게 된다. 따라서 프로그래머는 MPI 라이브러리와 같은 커뮤니케이션 API를 사용하지 않고 OpenCL 만을 사용하여 이종 클러스터를 타깃으로 어플리케이션을 작성할 수 있게 된다. SnuCL의 도움으로 OpenCL 어플리케이션은 단일 노드에서 이종 디바이스간 이식성을 가질 뿐만 아니라 이종 클러스터 환경에서도 디바이스간 이식성을 가질 수 있게 된다. 본 논문에서는 총 열한 개의 OpenCL 벤치마크 어플리케이션의 실험을 통하여 SnuCL의 성능을 보인다.

주 요 어 : OpenCL, 클러스터, 이종 컴퓨팅, 프로그래밍 모델, 런타임 시스템, 병렬화

학 번 : 2006-23153