# SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters

Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee
Center for Manycore Programming
School of Computer Science and Engineering
Seoul National University, Seoul 151-744, Korea
{jungwon, sangmin, jun, jeongho, gangwon}@aces.snu.ac.kr, jlee@cse.snu.ac.kr
http://aces.snu.ac.kr

## ABSTRACT

In this paper, we propose SnuCL, an OpenCL framework for heterogeneous CPU/GPU 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. The target cluster architecture consists of a designated, single host node and many compute nodes. They are connected by an interconnection network, such as Gigabit Ethernet and InfiniBand switches. Each compute node is equipped with multicore CPUs and multiple GPUs. A set of CPU cores or each GPU becomes an OpenCL compute device. The host node executes the host program in an OpenCL application. SnuCL provides a system image running a single operating system instance for heterogeneous CPU/GPU 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. SnuCL also provides collective communication extensions to OpenCL to facilitate manipulating memory objects. 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.

## Categories and Subject Descriptors

D.1.3 [**PROGRAMMING TECHNIQUES**]: Concurrent Programming; D.3.4 [**PROGRAMMING LANGUAGES**]: Processors – Code generation, Compilers, Optimization, Runtime environments

## General Terms

Algorithm, Design, Experimentation, Languages, Measurement, Performance


*ICS'12,* June 25–29, 2012, San Servolo Island, Venice, Italy.


## Keywords

OpenCL, Clusters, Heterogeneous computing, Programming models

## 1. INTRODUCTION

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)[9] 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. Some industry-leading hardware vendors such as AMD[1], IBM[6], Intel[8], NVIDIA[20], and Samsung[22] have provided OpenCL implementations for their hardware. This makes OpenCL a standard parallel programming model for general-purpose, heterogeneous computing systems.

However, 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[12]. 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 distribution of the workload and data (across nodes and across compute devices in a node). MPI functions are used

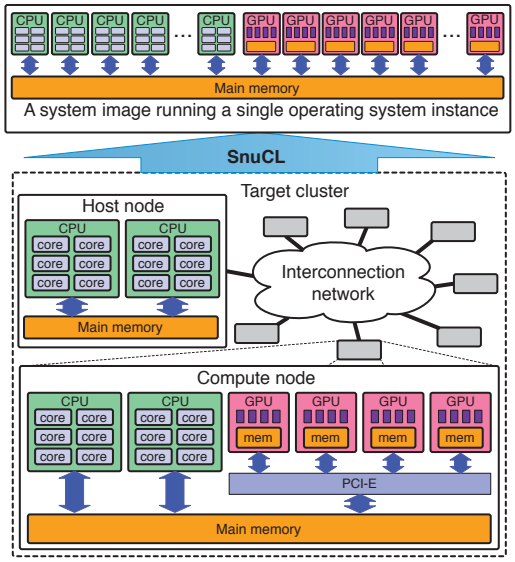


**Figure 1: Our approach.**

to communicate between the nodes in the cluster. As a result, the resulting application may not be executed in a single node.

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

The rest of the paper is organized as follows. Section 2 describes the design and implementation of the SnuCL runtime. Section 3, Section 4, and Section 5 describe memory management techniques, collective communications extensions to OpenCL, and code transformation techniques used in SnuCL, respectively. Section 6 discusses and analyzes the evaluation results of SnuCL. Section 7 surveys related work. Finally, Section 8 concludes the paper.

# 2. THE SNUCL RUNTIME

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

## 2.1 The OpenCL Platform Model

The OpenCL platform model[9] consists of a *host* connected to one or more *compute devices*. A compute device is divided into one or more *compute units* (CUs) which are further divided into one or more *processing elements* (PEs).

An OpenCL application consists of a host program and kernels. A host program executes on the host and submits commands to perform computations on a compute device or to manipulate memory objects. There are three different types of commands: kernel-execution, memory, and synchronization. A *kernel* is a function and written in OpenCL C. It executes on a compute device. It is submitted to a *command-queue* in the form of a kernel-execution command by the host program. A command-queue is created and attached to a specific compute device 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.

When a kernel-execution command is enqueued, an abstract index space is defined. The index space called NDRange is an N-dimensional space, where N is equal to 1, 2, or 3. An NDRange is defined by an N-tuple of integers and specifies the extent of the index space (the dimension and the size). An instance of the kernel, called a *work-item*, executes for each point in this index space. A work-item is uniquely identified by a global ID (N-tuples) defined by its point in the index space. Each work-item executes the same code but the specific pathway and accessed data can vary.

One or more work-items compose a *work-group*, which provides more coarse-grained decomposition of the index space. Each work-group has a unique work-group ID in the work-group index space and assigns a unique local ID to each work-item. Thus a work-item is identified by its global ID or by a combination of its local ID and work-group ID. The work-items in a given work-group execute concurrently on the PEs in a single CU.

## 2.2 Mapping Components

SnuCL defines a mapping between the OpenCL platform components and the target architecture components. A

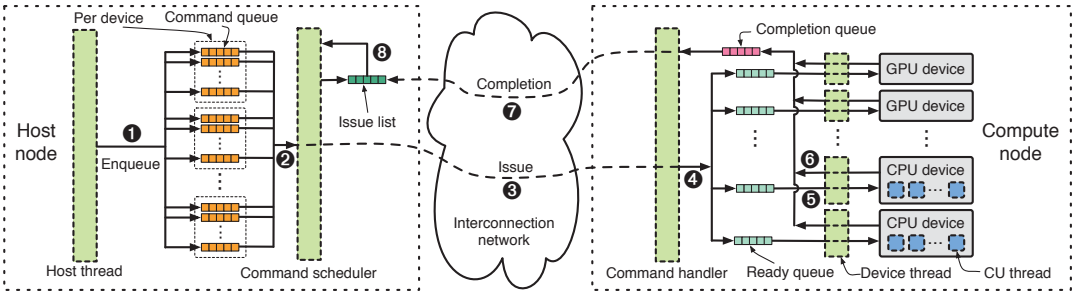


**Figure 2: The organization of the SnuCL runtime.**

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.

Since OpenCL has a strong CUDA heritage[12], 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[15].

## 2.3 Organization of the SnuCL Runtime

Figure 2 shows the organization of the SnuCL runtime. It consists of two different parts for the host node and a compute node.

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 2. The host thread enqueues commands to the command-queues (❶ in Figure 2). 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.

## 2.4 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[15]. 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.*[17] 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.

## 2.5 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.

# 3. MEMORY MANAGEMENT

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

## 3.1 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.

## 3.2 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[9]. 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.

## 3.3 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

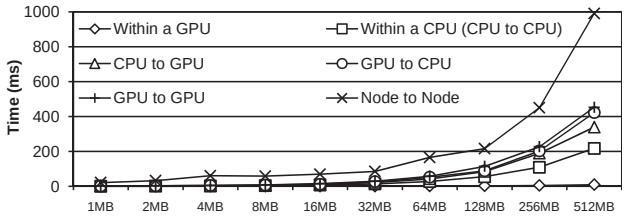


**Figure 3: Memory copy time.**

**Table 1: 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 |

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 3 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 1. Based on this metric, the runtime selects the nearest compute device in the device list of the buffer and copies the buffer contents to the target device from the selected device.

## 3.4 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 runtime 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 1), 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.

## 3.5 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[2].

One solution to this problem is introducing a multiple-writers protocol[2] 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.

## 3.6 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

**Table 2: 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 |

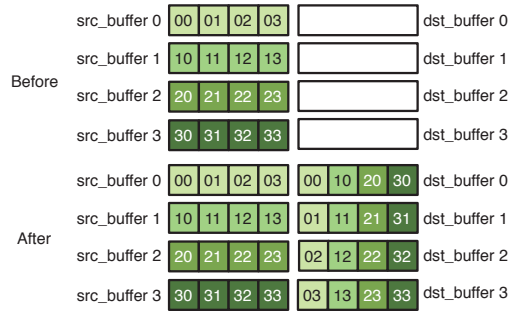


**Figure 4: `clEnqueueAlltoAllBuffer()` operation for four source buffers and four destination buffers.**

`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.

# 4. EXTENSIONS TO OPENCL

A buffer copy command (`clEnqueueCopyBuffer()`) is available in OpenCL[9]. Although this can be used for point-to-point communication in the cluster environment, OpenCL does not provide any 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 2 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 4.

# 5. CODE TRANSFORMATIONS

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

```
__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 5: (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.**

## 5.1 Detecting Buffers Written by a Kernel

To keep shared buffers consistent, the SnuCL runtime performs consistency management as described in Section 3. 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 con-

servative pointer analysis[24] is enough to obtain the necessary information because OpenCL imposes a restriction on the usage of global memory pointers used in a kernel[9]. Specifically, a pointer to address space A can 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 5 (b) shows the information generated from the OpenCL kernel in Figure 5 (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 5 (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.

## 5.2 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[15] 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 5 (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)[15]. 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[15] 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.

## 5.3 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 5 (c) and Figure 5 (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.

# 6. EVALUATION

This section describes the evaluation methodology and results for SnuCL.

**Table 3: 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)**

## 6.1 Methodology

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

**Benchmark applications.** We use eleven OpenCL applications from various sources: AMD[1], NAS[18], NVIDIA[19], Parboil[25], and PARSEC[3]. The characteristics of the applications and their input sets are summarized in Table 4. The applications from Parboil and PARSEC are translated to OpenCL applications manually. The applications from NAS are from SNU NPB Suite[21] 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[19]. We have implemented SnuCL source-to-source translators by modifying clang that is a C front-end for the LLVM[13] compiler infrastructure. The runtime uses Intel ICC 11.1[7], and NVIDIA's NVCC 4.0[19] to compile the translated kernels for CPU devices and GPU devices, respectively.

## 6.2 Results

Figure 6 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 4). Each application from NAS is shown with its input set. The CPUs in the

**Table 4: 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 | |

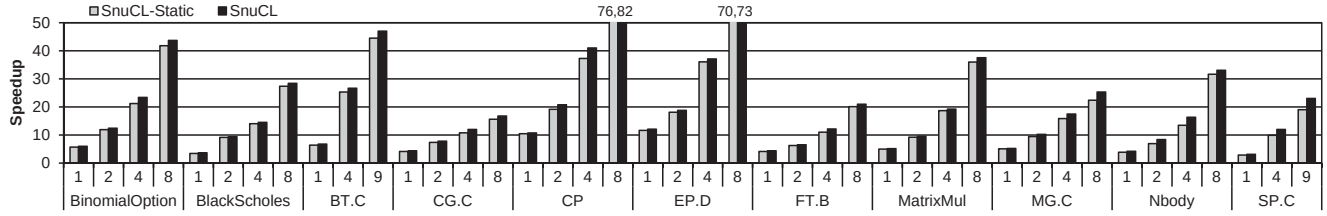


**Figure 6: Speedup over a single CPU core using CPU devices on Cluster A. The numbers on x-axis represent the number of CPU compute devices.**

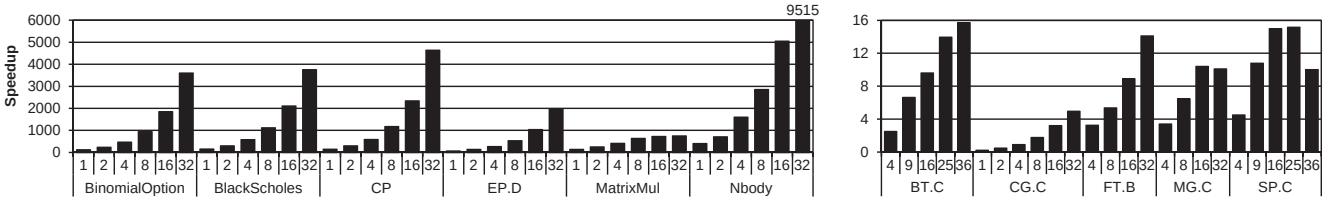


**Figure 7: Speedup over a single CPU core using GPU devices on Cluster A. The numbers on x-axis represent the number of GPU compute devices.**

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 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 2.4 is denoted by SnuCL in Figure 6.

All applications scale well in Figure 6. 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 7 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

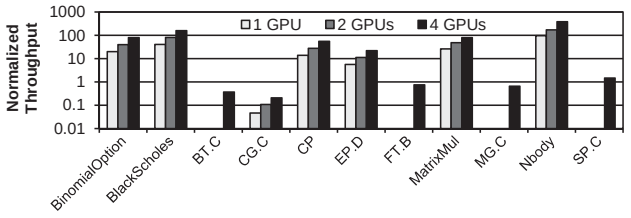


**Figure 8: Normalized throughput of GPU devices over a CPU device.**

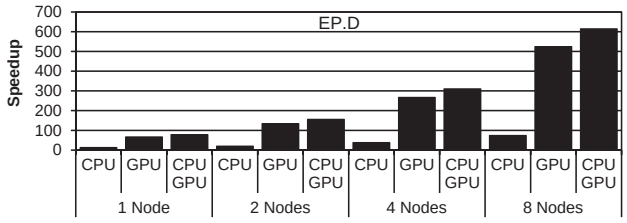


**Figure 9: Exploiting both CPU and GPU devices for EP.**

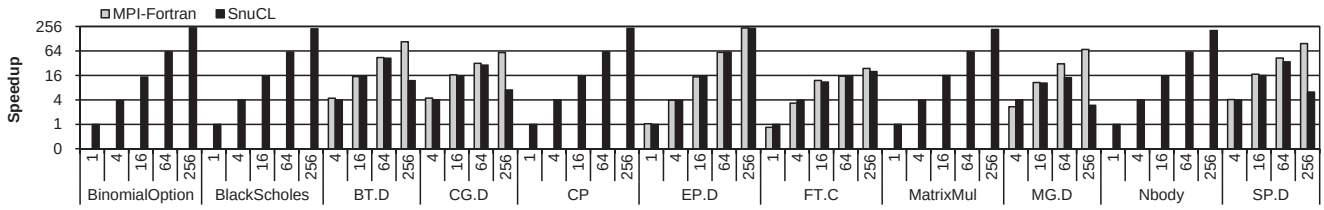


**Figure 10: Speedup over a single node (a CPU compute device with four CPU cores) on Cluster B. The numbers on x-axis represent the number of nodes (CPU compute devices).**

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 7, 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[16].

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 multiprocessor (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.

**Exploiting both types of devices**. Figure 8 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 9 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 9.

**Scalability**. To show the scalability of SnuCL, Figure 10 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 unmodified original MPI-Fortran versions (MPI-Fortran) from NPB. We build

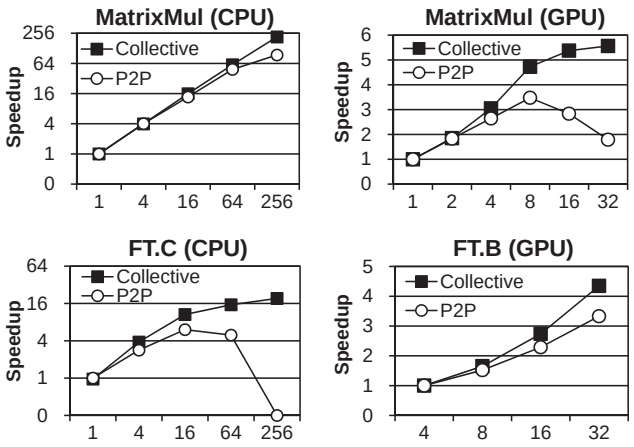


**Figure 11: The performance of collective communication extensions. X-axis shows the number of compute devices.**

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 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 operations. In addition, we use a depth tree to implement the broadcasting mechanism[23] 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 11 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 4, `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.

# 7. RELATED WORK

There are some previous proposals for OpenCL frameworks[5, 14, 10, 11, 15]. Gummaraju *et al.*[5] 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 lightweight `setjmp()` and `longjmp()` system calls to reduce the context switching overhead. Lee *et al.*[15] 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.*[14] 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[10, 4, 11]. Kim *et al.*[10] 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.*[4] 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.

The work most similar to ours is an OpenCL framework presented by Kim *et al.*[11] in that it exploits remote GPUs in a GPU cluster without MPI APIs. SnuCL focuses on the heterogeneity available in the heterogeneous CPU/GPU cluster environment. Supporting both CPUs and GPUs in the cluster raises many challenging issues, such as dynamic scheduling, performance portability, buffer management, and minimizing data transfer overhead. They did not address these issues. Moreover, SnuCL provides collective

communication extensions to OpenCL to achieve high performance and ease of programming. To our knowledge, our work is the first that shows OpenCL's portability and scalability on a heterogeneous CPU/GPU cluster.

# 8. CONCLUSIONS AND FUTURE WORK

In this paper, 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. Our future work is to improve the scalability of SnuCL for large-scale clusters by introducing an effective distributed task scheduling mechanism.

# 9. ACKNOWLEDGEMENTS

This work was supported in part by grant 2009-0081569 (Creative Research Initiatives: Center for Manycore Programming) from the National Research Foundation of Korea. This work was also supported in part by the Ministry of Education, Science and Technology of Korea under the BK21 Project. ICT at Seoul National University provided research facilities for this study.

# 10. REFERENCES

[1] AMD. *AMD Accelerated Parallel Processing (APP) SDK*, 2011. http://developer.amd.com/sdks/amdappsdk/pages/default.aspx.

[2] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *Computer*, 29:18–28, February 1996.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. 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.

[4] J. Duato, A. Pena, F. Silla, R. Mayo, and E. 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.

[5] J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. 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.

[6] IBM. *OpenCL Development Kit for Linux on Power*, 2011. http://www.alphaworks.ibm.com/tech/opencl.

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

[8] Intel. *Intel OpenCL SDK*, 2011. http://software.intel.com/en-us/articles/vcsource-tools-opencl-sdk/.

[9] Khronos OpenCL Working Group. *The OpenCL Specification Version 1.1*, 2010. http://www.khronos.org/opencl.

[10] J. Kim, H. Kim, J. H. Lee, and J. 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.

[11] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. 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.

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

[13] C. Lattner and V. 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.

[14] J. Lee, J. Kim, J. Kim, S. Seo, and J. 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.

[15] J. Lee, J. Kim, S. Seo, S. Kim, J. Park, H. Kim, T. T. Dao, Y. Cho, S. J. Seo, S. H. Lee, S. M. Cho, H. J. Song, S.-B. Suh, and J.-D. 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.

[16] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. 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.

[17] H. Li, S. Tandri, M. Stumm, and K. 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.

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

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

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

[21] S. Seo, G. Jo, and J. Lee. Performance Characterization of the NAS Parallel Benchmarks in OpenCL. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, IISWC '11, pages 137–148, 2011.

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

[23] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.

[24] B. 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.

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