A Distributed OpenCL Framework using Redundant Computation and Data Replication

Junghyun Kim Gangwon Jo Jaehoon Jung Jungwon Kim Jaejin Lee

Center for Manycore Programming
Department of Computer Science and Engineering
Seoul National University, Seoul 08826, Korea
{junghyun, gangwon, jaehoon, jungwon}@aces.snu.ac.kr, jaejin@snu.ac.kr
http://aces.snu.ac.kr

Abstract

Applications written solely in OpenCL or CUDA cannot execute on a cluster as a whole. Most previous approaches that extend these programming models to clusters are based on a common idea: designating a centralized host node and coordinating the other nodes with the host for computation. However, the centralized host node is a serious performance bottleneck when the number of nodes is large. In this paper, we propose a scalable and distributed OpenCL framework called SnuCL-D for large-scale clusters. SnuCL-D's remote device virtualization provides an OpenCL application with an illusion that all compute devices in a cluster are confined in a single node. To reduce the amount of control-message and data communication between nodes, SnuCL-D replicates the OpenCL host program execution and data in each node. We also propose a new OpenCL host API function and a queueing optimization technique that significantly reduce the overhead incurred by the previous centralized approaches. To show the effectiveness of SnuCL-D, we evaluate SnuCL-D with a microbenchmark and eleven benchmark applications on a large-scale CPU cluster and a medium-scale GPU cluster.

Categories and Subject Descriptors D.3.4 [*PROGRAMMING LANGUAGES*]: Processors—Optimization, Runtime environments

Keywords OpenCL, clusters, heterogeneous computing, programming models, runtime systems, redundant computation, data replication

1. Introduction

A heterogeneous system is a system that contains different types of processors, such as general-purpose CPUs, GPUs, FPGAs, DSPs, and accelerators of other types. One of the most popular accelerators is a graphics processing unit (GPU), and several programming models have been proposed to use it efficiently. Among others, CUDA[36] and OpenCL[22] have been used widely. Unlike CUDA, OpenCL provides a common abstraction layer, the OpenCL platform model, across different processor architectures including general-purpose CPUs. Its specification is maintained by Khronos group, and many hardware vendors, such as Altera, AMD, Apple, ARM, IBM, Imagination, Intel, MediaTek, NVIDIA, Qualcomm, Samsung, TI, and Xilinx, provide OpenCL platforms for their hardware.

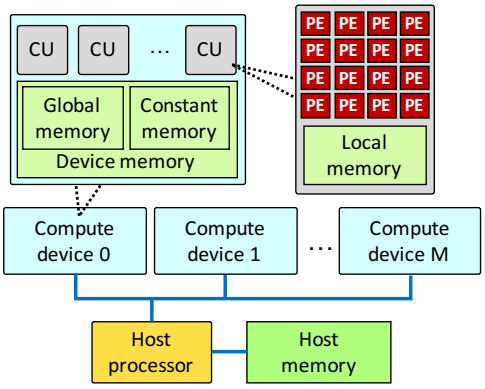

Figure 1. OpenCL platform model.

OpenCL platform model. Figure 1 shows the OpenCL platform model. It consists of a single *host processor* and one or more *compute devices* (*i.e.*, accelerators). The host processor has the host memory, and each compute device has its own device memory. The device memory is not visible to other compute devices and consists of global memory and read-only constant memory. An OpenCL application consists of a *host program* written in C and a set of *kernels*

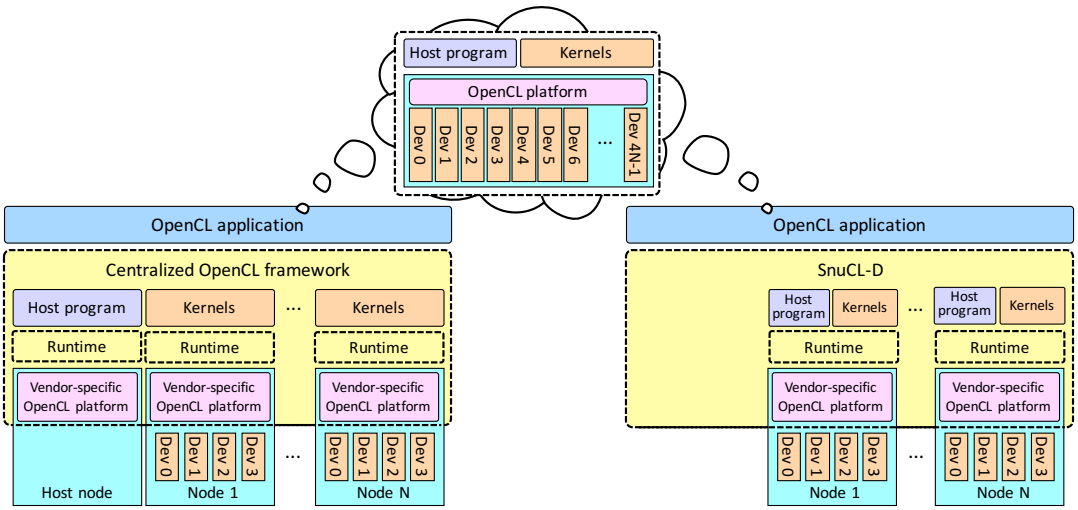

Figure 2. Comparison between previous OpenCL frameworks for clusters and SnuCL-D.

written in OpenCL C. OpenCL C is based on C99 with some extensions and restrictions. The host processor executes the host program, and compute devices execute kernels. Kernels are built online or offline. At least one *command-queue* is created and attached to a compute device by the host program. The host program submits a command to the compute device through the command-queue using an OpenCL host API function prefixed by `clEnqueue`.

OpenCL commands. There are three different types of OpenCL commands: *kernel execution*, *device memory access*, and *synchronization*. While a compute device executes a kernel-execution command, the OpenCL runtime executes a synchronization command to maintain correct execution order between commands. Since there is no shared address space between the host and a device or between a device and another device, a device-memory-access command exchanges data between them through OpenCL memory objects. The runtime and a device cooperate each other to execute a device-memory-access command. Based on the command-queue type, the enqueued commands are issued to the compute device *in order* or *out of order* by the runtime.

Absence of cluster abstraction. Since both OpenCL and CUDA have no cluster abstraction, each of them works only under a single operating system (OS) instance. Applications written solely in CUDA or OpenCL cannot run on a cluster as a whole. To develop OpenCL or CUDA applications for the cluster, a communication library such as MPI[33] must be used to support communication between OpenCL platform instances in different nodes. As a result, programmers are forced to use a mix of two different programming models (*e.g.*, MPI + OpenCL or MPI + CUDA). However, it is cumbersome and error-prone for programmers to switch between the two different programming models in different phases of an application. Moreover, they need to distribute the application workload hierarchically in two levels: across nodes and inside a node. Thus, even if heterogeneous computing under a single OS instance has been widely spread, more research is needed for a cluster running multiple OS instances, one in each node.

1.1 Problems of Previous Approaches

A number of studies address the issue of cluster abstraction for OpenCL[3, 6, 14, 21, 23, 25, 47, 49], and their proposals are based on a common idea: a *centralized host node*. Their differences come from the design and implementation. The left-hand side of Figure 2 illustrates the common idea of the previous approaches. They all have a *centralized* host node that executes the OpenCL host program. Other nodes are compute nodes and perform kernel computations on their compute devices. Each node including the host runs a vendor-specific OpenCL platform. The runtime takes care of communication between different OpenCL platforms. As a result, the centralized framework provides a single OpenCL platform image to the application. This design is natural and intuitive because the OpenCL platform model itself has a centralized host.

Types of overheads. There are three different types of overheads along the execution path of an OpenCL command in the centralized approach: *command queueing*, *command scheduling*, and *command delivery*. The command-queueing overhead occurs when the host enqueues a command using an OpenCL API function prefixed by `clEnqueue`. Actual insertion is done by the runtime in the API function. When the runtime sends a command to its target device through the interconnection network, the command-delivery overhead occurs. The command-scheduling overhead includes all overheads that occur between the point after the runtime enqueues a command and the point before the runtime sends the command to the target device. It includes the overhead of determining the execution order of commands and maintaining consistency for OpenCL memory objects.

Type	Source
Command queueing	When enqueueing a command to a command-queue.
Command scheduling	Between the point after enqueueing a command and the point before sending the command to the target device.
Command delivery	When delivering a command to the target compute device.
Host-data transfer	When transferring data between the host and a compute device.

Table 1. Overhead Types in the Centralized Approach.

To execute a kernel on a compute device, it may be necessary to transfer data accessed by the kernel from the host to the device, or vice versa. *Host-data-transfer overhead* occurs in the centralized approach when the runtime transfers data between the host and a device through the interconnection network. We summarize the four different overhead types in Table 1.

Even though the target device has enough computing power to execute many commands, the host node may not be able to deliver enough commands and data to it because of the overheads described above. This problem becomes more serious as the number of nodes increases.

However, most of the previous approaches do not address the scalability issue of their solutions. They evaluate their OpenCL frameworks only for a small-scale cluster. An exception is SnuCL[23, 24]. It is an open-source centralized OpenCL framework for clusters and uses a large-scale cluster with 256 nodes for evaluation. SnuCL does not scale well with more than 64 nodes for some applications. The single host node becomes a significant performance bottleneck for the large-scale cluster because it schedules and delivers commands and data to a large number of compute nodes.

1.2 Proposed Techniques

In this paper, we propose a distributed OpenCL framework called *SnuCL-D* to solve the scalability problem of the previous centralized approaches. It overcomes the performance bottleneck incurred by the four types of overheads.

Exploiting redundant computation and data replication. Instead of making the host node to execute the OpenCL host program, SnuCL-D executes the same host program in every node redundantly as shown in Figure 2. This also makes the data produced by the host program to be replicated in every node. As a result, the OpenCL application uses the compute devices as if they were confined in a single node. However, no code modification is necessary to run an OpenCL application under SnuCL-D. In addition, SnuCL-D does not need a host node. Thus, it requires one less node than the centralized approach.

When the execution of the host program is not deterministic, replicating the host program in each node introduces non-determinacy in interactions with the OS (*e.g.*, file I/O) as well as command scheduling in multithreaded host programs. We implement a limited form of determinacy for common file I/O operations and function calls (*e.g.*, `srand`) in SnuCL-D. However, we do not address subtler sources of non-determinacy including multithreaded host programs because deterministic execution is a well-known problem[10, 11, 32, 38].

Remote device virtualization. The *remote device virtualization* (RDV) technique proposed in SnuCL-D enables a node to see not only its own compute devices, called *actual devices*, but also those in other nodes, called *virtual devices*. A command enqueued to a command-queue attached to an actual device is called an *actual command*. Otherwise, the command is called a *virtual command*. Actual commands are executed normally, but virtual commands are discarded by the runtime and not actually executed. This eliminates the inter-node command-delivery overhead and also significantly reduces the command-scheduling overhead. While the host node in the centralized approach delivers all the commands to their target nodes, each node in SnuCL-D knows exactly which commands are for its actual devices, and it just locally executes the commands for its actual devices.

The host data replication makes SnuCL-D outperforms the centralized approach because SnuCL-D does not have the host-data-*send* overhead (*i.e.*, the overhead occurred when the host sends data to a compute device) that is unavoidable in the centralized approach. On the other hand, SnuCL-D has more severe host-data-*receive* overhead (*i.e.*, the overhead occurred when a compute device sends data to the host) for device-memory-read commands than the centralized approach. However, device-memory-read commands occur less frequently than other types of commands. We discuss this in detail in Section 3.1.

New OpenCL API function and queueing optimization. A memory object in OpenCL is not bound to a specific compute device. That is, its location is not fixed to a specific device, and it moves around different devices. When a memory object is passed to a kernel by the host and the kernel executes on a device, the object resides on the device. This requires a sophisticated memory consistency management mechanism for memory objects, resulting in increasing the command-scheduling overhead. We propose a new OpenCL API function that makes a memory object to be bound to a specific device. It further reduces the command-scheduling overhead by alleviating the consistency management overhead for the memory object. We also propose a queueing optimization technique. Queueing optimization together with the new API function significantly reduces the queueing overhead that is unavoidable in the centralized approach.

Note that both of the centralized approach and SnuCL-D cannot execute an OpenCL application that requires more memory space than is available in the main memory of

each node. This implies that SnuCL-D will not work well when the primary motivation for using a cluster is exploiting more memory space across the entire cluster rather than exploiting more computing power (*e.g.*, more GPUs running in parallel). While the memory footprint of each device memory remains the same, SnuCL-D sacrifices the main memory footprint for performance. However, exploiting redundant computation and data replication is sometimes a preferable common practice in large-scale computing to improve performance even though it requires more memory footprint[2, 20, 26, 41, 42, 48].

The implementation of SnuCL-D is based on SnuCL[23, 24]. We show the effectiveness of SnuCL-D by comparing it with SnuCL and MPI-Fortran. We run eleven benchmark applications on a large-scale CPU cluster with 512 nodes (4096 CPU cores in total) and a medium-scale GPU cluster with 36 nodes (144 GPU devices in total).

The rest of the paper is organized as follows. The next section describes programming models of OpenCL and SnuCL-D. Section 3 explains the design and implementation of SnuCL-D. Section 4 evaluates SnuCL-D. Section 5 describes related work. Finally, Section 6 concludes the paper.

2. Programming Models

In this section, we describe programming models of OpenCL and SnuCL-D.

2.1 OpenCL Programming Model

We have introduced the OpenCL platform model in the previous section. We start with the OpenCL execution model.

Execution model. An execution instance of a kernel is called a *work-item*. An index space (up to three dimensional) specifies the total number and shape of work-items that execute a kernel. A point in the space corresponds to a work-item, and the Cartesian coordinates of the point is the unique global ID of the work-item. One or more work-items are grouped in a *work-group*. Each work-group also has a unique ID, and each work-item in a work-group has a unique local ID. The host program specifies the total number of work-items and the work-group size for a kernel before it submits the kernel command.

As shown in Figure 1, each compute device contains one or more *compute units* (CUs). Each CU contains one or more *processing elements* (PEs) and local memory that is not visible to other CUs. A PE is a processor and has its own private memory (*e.g., registers*). The OpenCL runtime distributes work-groups to CUs. Work-items in a work-group execute *concurrently* on PEs in an SPMD (Single Program, Multiple Data)[15] manner.

Memory regions and objects. There are four distinct memory regions in a compute device (Figure 1): `__global`, `__constant`, `__local` and `__private`. These regions are accessible to kernels. While the host program accesses the compute device memory via device-memory-access commands, a kernel may not access the host memory.

OpenCL has three different types of memory objects: *buffers*, *images*, and *pipes*. For simplicity and without loss of generality, we consider only buffer objects in this paper. A buffer object is similar to a byte array in C and contains any type of data. To read, write, or copy a buffer, a device-memory-access command is submitted to a command-queue by the host program.

Synchronization mechanisms. A *work-group barrier* is used to synchronize work-items in the same work-group. There is no synchronization mechanism available between work-groups. Synchronization between commands in the same command-queue can be specified by a *command-queue barrier*. To synchronize commands in different command-queues, *events* are used. An API function prefixed by `clEnqueue` returns an event object associated with the command enqueued. The event object can be queried to track the execution status of the command. For example, when the command is enqueued, the state of the event becomes `CL_QUEUED`, and when the command has completed, it becomes `CL_COMPLETE`. In addition, the API function takes an *event wait list* as an argument. The enqueued command cannot be scheduled for execution until all commands associated with the events in the wait list have completed.

Memory consistency model. Since a memory object in OpenCL is not bound to a specific compute device, a new copy of a memory object may be created when a different device accesses the same memory object. For example, if two different kernel-execution commands K_A and K_B executing on different devices modify the same memory object M, a different copy of M resides in each compute device. To guarantee memory consistency, the runtime needs to take some action for the two different copies of M.

OpenCL defines a relaxed memory consistency model in which memory is only guaranteed to be consistent after each synchronization point. There are several types of synchronization points defined in OpenCL[22]. For example, a work-group barrier is a synchronization point for work-items in the same work-group. The point when a kernel is issued to a device for execution is another example of a synchronization point.

2.2 Sample OpenCL Application

Figure 3 shows an OpenCL application adding two vectors (`C = A + B`) using multiple GPUs. For simplicity, we assume that both the number of available GPUs in the system (`ndevs`) and the size of the vectors (`N`) are a power of two. We also assume `N` is much bigger than `ndevs`.

Kernel code. The kernel source code (`vec_add`) is embedded in the host program as a string (line 1 - line 11). The kernel binary is built online in the application. OpenCL C built-in

```
1  const char *kernel_source =
2  "__kernel void vec_add( __global float *A,     \n" \
3  "                       __global float *B,     \n" \
4  "                       __global float *C,     \n" \
5  "                       const unsigned int n)  \n" \
6  "{                                             \n" \
7  "  int id = get_global_id(0);                  \n" \
8  "                                              \n" \
9  "  if (id < n)  // Bound check                 \n" \
10 "    C[id] = A[id] + B[id];                    \n" \
11 "}                                             \n";
12
13 int main() {
14   float *h_A, *h_B, *h_C;
15   ... // Initialize h_A, h_B, and h_C
16
17   cl_platform platform;
18   clGetPlatformIDs(1, &platform, NULL);
19
20   int ndevs;
21   cl_device_id devs[MAX_NDEVS];
22   clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU,
23                  MAX_NDEVS, devs, &ndevs);
24
25   cl_context ctxt = clCreateContext(0, ndevs, devs, ...);
26
27   cl_command_queue cmdqs[MAX_NDEVS];
28   for(int i = 0; i < ndevs; i++)
29     cmdqs[i] = clCreateCommandQueue(ctxt, devs[i], ...);
30
31   cl_kernel kls[MAX_NDEVS];
32   ... // Compile and build ndevs kernel objects
33
34   unsigned int n = N/ndevs; // Vector size per device
35   unsigned int size = n*sizeof(float);
36
37   cl_mem d_A[MAX_NDEVS], d_B[MAX_NDEVS], d_C[MAX_NDEVS];
38   for(int i = 0; i < ndevs; i++) {
39     d_A[i] = clCreateBuffer(ctxt, ..., size, ...);
40     d_B[i] = clCreateBuffer(ctxt, ..., size, ...);
41     d_C[i] = clCreateBuffer(ctxt, ..., size, ...);
42   }
43
44   for(int i = 0; i < ndevs; i++) {
45     clEnqueueWriteBuffer(cmdqs[i], d_A[i], ...,
46                          size, h_A + i*n, ...);
47     clEnqueueWriteBuffer(cmdqs[i], d_B[i], ...,
48                          size, h_B + i*n, ...);
49   }
50
51   ... // Define the index space (gws and lws)
52
53   for(int i = 0; i < ndevs; i++) {
54     clSetKernelArg(kls[i], 0, sizeof(cl_mem), &d_A[i]);
55     clSetKernelArg(kls[i], 1, sizeof(cl_mem), &d_B[i]);
56     clSetKernelArg(kls[i], 2, sizeof(cl_mem), &d_C[i]);
57     clSetKernelArg(kls[i], 3, sizeof(unsigned int), &n);
58     clEnqueueNDRangeKernel(cmdqs[i], kls[i], ...,
59                            gws, lws, ...);
60   }
61
62   for(int i = 0; i < ndevs; i++)
63     clFinish(cmdqs[i]);
64
65   for(int i = 0; i < ndevs; i++)
66     clEnqueueReadBuffer(cmdqs[i], d_C[i], ...,
67                         size, h_C + i*n, ...);
68   ...
69 }
```

Figure 3. An OpenCL application for multiple GPUs.

function `get_global_id(0)` returns the global ID of the work-item that executes the kernel. The index space is one dimensional. The first three kernel arguments are pointers to OpenCL buffers in the device global memory.

Host program. At the beginning, the host program (`main`) allocates memory spaces for arrays `h_A`, `h_B`, and `h_C` in the host main memory. Then, they are initialized (lines 14 - 15).

After obtaining an available OpenCL platform in the system at line 18, the host obtains the list of available GPUs in the platform (lines 22 - 23). Array `devs` and variable `ndevs` contain the list and the number of available GPUs, respectively. We assume `MAX_NDEVS` $\geq$ `ndevs`.

The host creates an OpenCL *context* at line 25. The OpenCL runtime uses the context to manage compute devices and OpenCL objects, such as command-queues, memory objects, and kernel objects. Then, the host creates a command-queue for each GPU in the context at line 29. At line 32, the host builds the embedded kernel source code for the GPUs and creates `ndevs` kernel objects.

To transfer the contents of `h_A` and `h_B` to GPUs, the host creates two buffer objects `d_A[i]` and `d_B[i]` for each GPU `i` (lines 39 - 40). The size of `d_A[i]` and `d_B[i]` is `n*sizeof(float)`. In addition, the host creates a buffer `d_C[i]` for each GPU `i` to transfer the result from the GPU to the host (line 41). Note that locations of the buffers are not known yet at this point because `clCreateBuffer` does not have any argument from which the target GPU can be inferred. The host copies the contents of `h_A` and `h_B` to buffers `d_A[i]` and `d_B[i]`, respectively, at lines 45 - 48. At this point, the OpenCL runtime allocates memory spaces in GPU `i` to `d_A[i]` and `d_B[i]`. Using the argument `cmdqs[i]` in `clEnqueueWriteBuffer` as a clue, the runtime finds that the target device is GPU `i`. After allocating the memory spaces, the runtime copies the contents.

After setting up kernel arguments, the host enqueues a kernel command using the non-blocking API function `clEnqueueNDRangeKernel` for each GPU at lines 54 - 59. The work-group size and the total number of work-items for the kernel are given by `lws` and `gws`, respectively. Since the kernel index space is one dimensional, both `lws` and `gws` are a one-dimensional integer array. Then, the runtime schedules each kernel command to its target GPU, and the GPU executes the kernel command.

The synchronization host API function `clFinish` at line 63 blocks until the kernel command in `cmdqs[i]` has completed. After completion, a device-memory-read command `clEnqueueReadBuffer` reads the contents of `d_C[i]` from the device memory and puts them to `h_C` (lines 66 - 67).

Finally, the host releases OpenCL objects and frees dynamically allocated host memory spaces.

2.3 SnuCL-D Programming Model

SnuCL-D has the same programming model as that of OpenCL. No code modification is necessary. The OpenCL application in Figure 3 runs well under SnuCL-D without any modification. Since SnuCL-D is an OpenCL programming model, it supports any heterogeneous compute devices installed in the OpenCL platform. For example, an OpenCL application, in which a set of CPU compute devices and a set of GPU compute devices collaborate each other, also works well under SnuCL-D without any code modification.

SnuCL-D is a distributed framework in the sense that the host program and data are replicated and the host program executes in every node in the cluster. Each node in the cluster also runs an instance of the SnuCL-D runtime. Since each node executes the same OpenCL host program, the programming model of SnuCL-D looks like an SPMD programming model[15]. It is true at the host side. However, the host-side SPMD programming model is not exposed to the programmer, and each compute device may execute different kernel code. Thus, the entire programming model of SnuCL-D is not an SPMD programming model. In the programmer's perspective, it is just an OpenCL programming model.

SnuCL-D also provides collective communication extensions proposed by SnuCL[23]. The extensions are implemented in SnuCL-D's distributed framework.

3. Design and Implementation of SnuCL-D

In this section, we describe the design and implementation of SnuCL-D and how it overcomes the performance bottleneck incurred in the previous centralized approaches.

As shown in Figure 2, SnuCL-D is laid between the cluster and an OpenCL application. Each node in the cluster executes a copy of the host program and a SnuCL-D runtime instance. Both are threads and belong to an MPI process running in the node. The SnuCL-D runtime instances work together to provide the single OpenCL platform image to the OpenCL application. SnuCL-D uses pairs of coupled asynchronous MPI send and receive calls to implement communication between different SnuCL-D runtime instances.

Each node may have multiple vendor-specific OpenCL platforms for different accelerators. The SnuCL-D runtime instance controls the OpenCL platforms using the OpenCL installable client driver (ICD)[22].

3.1 Observations

Commonly used commands in OpenCL host programs include kernel-execution, device-memory-write (*e.g.*, `clEnqueueWriteBuffer`), device-memory-read (*e.g.*, `clEnqueueReadBuffer`), and device-memory-copy (*e.g.*, `clEnqueueCopyBuffer`) commands.

A device-memory-read command copies the contents of a buffer from a device to the host memory. To run the host program correctly in SnuCL-D, the host memory in each node must be kept up-to-date after executing a device-memory-read command. After a device-memory-read command is executed on the target actual device in a node N_a, the SnuCL-D runtime instance in N_a propagates the data to other nodes. When the corresponding virtual device-memory-read command is scheduled by a runtime instance in another node N_b, N_b receives the contents from N_a who owns the actual device. This makes the host memory in each node be kept up-to-date after each device-memory-read command. Since the data read is broadcast to all other nodes from the actual device, exploiting redundant computation and data replication is not beneficial to device-memory-read commands. They introduce inter-node communication that does not occur in the centralized approach in which only the main memory in the host node needs to be updated.

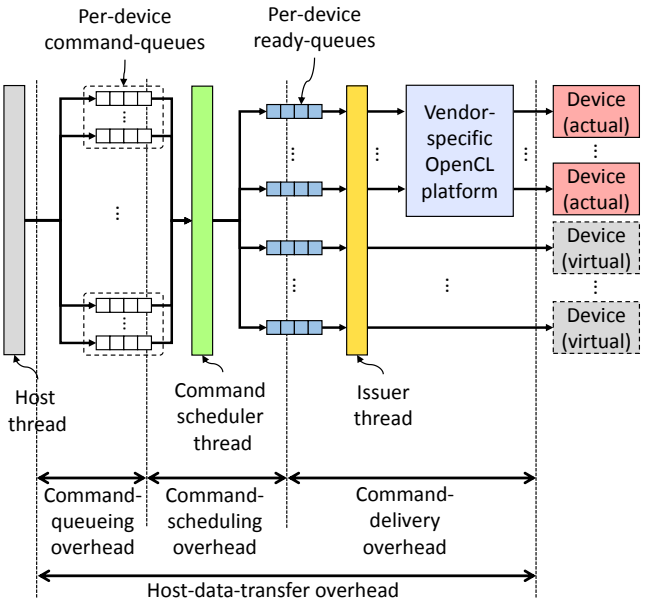

Figure 4. The organization of the SnuCL-D runtime.

However, exploiting redundant computation and data replication in SnuCL-D is do beneficial to kernel-execution, device-memory-write, and device-memory-copy commands. For kernel-execution and device-memory-write commands, SnuCL-D does not need any inter-node command and data communication because of RDV while the host node in the centralized approach needs to send both a command message and data to the target node. For device-memory-copy commands, SnuCL-D directly performs data communication between the source device and the destination device. However, the centralized approach needs one more step: the host node sends control messages to the source and destination devices. Then, these devices communicate with each other to transfer the data.

The design of SnuCL-D is based on an observation that device-memory-read commands are executed much less of-

ten than other commonly-used commands. We show this in Section 4.3. Consequently, the degree of performance degradation by device-memory-read commands is much less than that of improvement by other types of commands.

3.2 Organization of the SnuCL-D Runtime

Figure 4 shows the organization of the SnuCL-D runtime in each node. It also shows the four types of overheads described in Section 1.1. The runtime basically consists of a scheduler thread, an issuer thread, and per-device ready-queues. A command-queue or ready-queue is implemented with a non-blocking lock-free single-producer/single-consumer queue[31, 50] to boost performance. The runtime also maintains a list of event objects that are associated with commands. When an event object is associated with an actual command, it is called an *actual event*. Otherwise, it is called a *virtual event*. Similarly, when a command-queue is attached to an actual device, it is called an *actual command-queue*. Otherwise, it is called a *virtual command-queue*.

RDV implementation. To implement RDV, the runtime instance in each node exchanges its own actual device information with those in other nodes when the host program invokes an OpenCL API function for the first time. A unique device ID is assigned to each device across the cluster. Then, each runtime instance creates virtual devices in its node. Every runtime instance knows which device actually resides in which node.

Command scheduler thread. The role of the command scheduler is enforcing a consistent execution order on actual commands across different nodes. It honors the order enforced by synchronizations in the host program. It also manages memory consistency. The memory consistency management mechanism is described in detail in Section 3.3. The command scheduler continuously visits all the command-queues in a round-robin manner. It schedules commands in a command-queue one by one from the head to the tail.

Scheduling actual commands. For an actual command-queue, the scheduler checks synchronization dependences for each command. If all events (without regards to they are actual or virtual) in the event wait list of an actual command have been set to `CL_COMPLETE`, the command has no synchronization dependence. If so, the scheduler dequeues the command and sends it to the corresponding ready-queue. Otherwise, the scheduler moves on either to the next command in the same queue or to the next queue depending on the command-queue type (*i.e.*, in order or out of order). To determine the execution order of actual commands, the command scheduler directly executes synchronization commands, such as command-queue barriers, without sending them to ready-queues.

Scheduling virtual commands. When the command-queue is virtual, the scheduler checks for each command C if all actual events, say $e_0, e_1, \cdots, e_n$, in its event wait list have been set to `CL_COMPLETE`. If so, the scheduler notifies the owner runtime of the actual command that corresponds to C about the completion of $e_0, e_1, \cdots, e_n$. After receiving the completion message, the owner runtime sets to `CL_COMPLETE` its virtual events that correspond to $e_0, e_1, \cdots, e_n$. Then, C is dequeued and discarded. Otherwise, the scheduler moves on either to the next command in the same queue or to the next queue depending on the command-queue type. An exception is a virtual device-memory-read command. The scheduler sends it to the corresponding ready-queue.

Issuer thread. The *issuer* in the SnuCL-D runtime continuously visits all the ready-queues in a round-robin manner. When there is a command in a ready-queue, the issuer dequeues it and processes it one by one. If a dequeued command is a virtual device-memory-read command, the issuer receives the data from the node that executes the corresponding actual command. Then, the issuer updates the host main memory. If the dequeued command is an actual command, after obtaining the memory object accessed by the command, the issuer invokes an OpenCL API function provided by the vendor-specific OpenCL platform to execute the command or executes it directly. The details of obtaining the memory objects are described in Section 3.3. After completion of the actual command, the issuer sets the associated actual event to `CL_COMPLETE`.

3.3 Consistency Management

Since there is no shared address space between different compute devices, and an OpenCL memory object is not bound to a specific compute device, two compute devices may have different copies of the same memory object after executing a command that accesses the memory object. Thus, the runtime needs to guarantee memory consistency between the multiple copies of the same memory object. There are two different sources of inconsistency: *simultaneous accesses* and *sequential accesses* to the same memory object by multiple commands.

Simultaneous accesses by multiple commands. When the same set of locations in the memory object is updated by multiple commands simultaneously, we may choose any copy as the latest update for the memory object at the next synchronization point. This conforms to the OpenCL memory consistency model. However, if they update different sets of locations in the same memory object, the problem becomes similar to false sharing in the page-level software shared virtual memory (SVM) system[5].

To solve this problem, we may use a multiple-writers protocol that was used in traditional software SVM systems[5]. However, this incurs a significant overhead because of twins (copies of the original) and computations for diffs (differences between the twin and a modified memory object). Instead, SnuCL-D serializes conflicting commands. A *conflict* occurs

between two unordered commands if they access the same memory object, and at least one writes to it[30].

When a command C is enqueued by a host API function prefixed by `clEnqueue`, the runtime checks in the API function if there is a conflict between C and any command C_e that has been enqueued already. If so, the runtime inserts the event object for C_e in the event wait list of C. Consequently, the order enforced by the serialization is the enqueueing order of commands by the host program.

Sequential accesses by multiple commands. Even though we serialize two conflicting accesses to the same memory object, we still have the inconsistency problem caused by sequential accesses to the same memory object by multiple devices. This is because there may exist multiple different copies of the same memory object. To solve this problem, SnuCL-D maintains a *latest device list* for each memory object. It contains compute devices that have the latest copy of the memory object.

OpenCL command	Nearest device	Operation
Actual	Actual	Do nothing (the node that executes the OpenCL command already has the latest copy).
	Virtual	Receive the contents from the node who owns the nearest device.
Virtual	Actual	Send the contents to the node who executes the corresponding actual command.
	Virtual	Do nothing.

Table 2. Consistency Management Command.

When the command scheduler schedules an actual command C, the scheduler inserts a special management command in the corresponding ready-queue for each memory object accessed by C. These management commands are followed by C. The management command brings the most up-to-date copy of the memory object by looking for the nearest device from the latest device list of the memory object. When C is virtual, the scheduler inserts a management command in the corresponding ready-queue for each memory object M_i accessed by C if the nearest device of M_i is actual. Table 2 describes the operation of a management command depending on its type and the latest device.

After scheduling an actual or virtual command, the scheduler updates the latest device list of each memory object accessed by the command. If the command writes to the memory object, the updated list contains only the target device of the command. SnuCL-D uses a distance-based tie-breaking algorithm that considers both inter-node and intra-node memory hierarchy when there are more than one device in the latest device list. For example, the distance between two devices may be defined by *interconnection network latency + PCIe latency*.

Since SnuCL-D pairs an MPI send call with an MPI receive call to implement the consistency management command, we may have a mismatched send or receive call if the command scheduler in each node do not see the same latest device list for each memory object. To prevent this, SnuCL-D also treats two unordered commands reading the same memory object as conflicting commands. Thus, they are ordered by the enqueueing order of commands in the conflict detection phase explained above.

Guaranteeing consistency. When we treat the host and compute devices as threads and each OpenCL memory object as a shared variable between them, serializing conflicting commands is analogous to enforcing sequential consistency for shared-memory parallel programs[28].

Let P be a set of directed edges that represent the program order of shared-variable operations in a multithreaded program, and let C be a set of bidirectional edges that represent the conflict relation on the shared-variable operations. A critical cycle is a cycle in $P \cup C$ and represents the existence of an inconsistent execution order of shared-variable operations. Breaking such a cycle by either enforcing program edges or orienting conflict edges in the cycle guarantees sequential consistency[29, 30, 44, 46].

Serializing commands accessing the same memory object in SnuCL-D is a way of orienting conflict edges. It guarantees sequential consistency on the accesses of OpenCL memory objects. Sequential consistency satisfies the OpenCL's relaxed memory consistency model.

SnuCL also adopts serializing conflicting commands and maintaining the latest device list to manage consistency. However, the distributed version of SnuCL-D significantly reduces the frequency of inter-node communication. To move data in SnuCL, the host node should deliver a control message to each of the source and destination nodes. Then, the source sends the data to the destination. However, the source and destination nodes in SnuCL-D directly communicate with each other to transfer the data without any control message.

3.4 Non-determinacy in Command Scheduling

If there is no synchronization enforced between commands in the host program, the host program execution instance in each node may not follow the same command execution order. As a result, the outcome of an OpenCL application may be different from what the programmer expects.

Single-threaded host programs. When the host program is single threaded, solving the non-determinacy problem is relatively easy compared to multithreaded host programs. In addition, single-threaded host programs are much more common than multithreaded host programs in reality. SnuCL-D enforces the enqueueing order specified by the host program on device-memory-access commands. It also guarantees having the same latest device list for the same memory object across the nodes in SnuCL-D. This is another major difference between SnuCL and SnuCL-D in consistency management.

Multithreaded host programs. All OpenCL API functions but `clSetKernelArg` are thread-safe[22], and OpenCL allows multithreaded host programs. However, unlike single-threaded host programs, the enqueueing order of commands is not deterministic in a multithreaded host program.

A solution to this problem is deterministic multithreading[10, 11, 32, 38]. It enables a multithreaded programs to execute deterministically. This implies that communications between threads in a multithreaded program occur in the same order from a run to another run. Thus, we can guarantee the same enqueueing order of commands for all multithreaded host program instances in the cluster using deterministic multithreading.

3.5 Non-determinacy in the Result of a Function Call

To solve the non-determinacy in the result of a function call, global synchronization between host program instances is required. SnuCL-D exploits wrapper functions and a root node. A function that causes a non-deterministic result across host program instances is implemented as a wrapper. Only a designated root node (*e.g.*, a node whose MPI rank is zero) performs the original function. Other nodes receive the result from the root node in the wrapper if necessary. Synchronization between them are implemented in the wrapper.

For example, if there is a `srand` call to install a seed value to generate random numbers in the host program, SnuCL-D implements a wrapper for `srand`. Each host program instance invokes this wrapper. The root node executes the original `srand` and propagates its result to other nodes in the wrapper. This makes every node have the same seed value. Another example is handling file I/Os. File I/O operations also require synchronization between the host program instances. Only the root node performs a file-write operation. Other nodes do nothing. Before performing a file-read operation, other nodes perform synchronization with the root node. This synchronization guarantees correctness when the host program writes to a file and then reads from it.

OpenCL Host API calls `clWaitForEvents(num_evs, ev_wait_list)` and `clFinish(cmd_q)` also need global synchronization between host program instances. The call `clFinish(cmd_q)` makes the host program block and wait until all commands previously enqueued to `cmd_q` have completed. A node who has the actual device to which `cmd_q` is attached broadcasts a completion message to other nodes in SnuCL-D. The call `clWaitForEvents(num_events, event_wait_list)` makes the host program wait until all commands associated to the events in `event_wait_list` have completed. Thus, a node who owns each actual event in `event_wait_list` broadcasts the completion to other nodes in SnuCL-D.

3.6 New API Function: `clAttachBufferToDevice`

To reduce the consistency management overhead in command scheduling, we propose a new OpenCL host API function:

```
void clAttachBufferToDevice(cl_mem m, cl_device_id d);
```

If this function is called, the SnuCL-D runtime assumes that compute device `d` always has the latest copy of memory object `m`. The runtime does not need to maintain the latest device list for `m` and to search the list to find the latest copy. This API function is useful when a memory object is accessed by only one device in an OpenCL application. In many OpenCL applications, a memory object is accessed by only one device.

At the point of the API function call, the runtime copies the contents of `m` to `d` from a device listed in `m`'s latest device list. Then, it removes the latest device list of `m`. From then on, there will be no latest device list management for `m`. If there is no such device, the runtime write zeros to `m` on `d`.

If a command C enqueued for another device `e` ($\neq$`d`) updates `m`, the command scheduler inserts a special command in the ready-queue for the device `e` to obtain the contents from `d` before it schedules C in the ready-queue. Then, another special command that sends the updated contents to `d` follows C in the ready-queue. Thus, even if a programmer attaches a memory object to a wrong device (*e.g.*, it is not the only device that updates `m`), the SnuCL-D runtime still guarantees correctness with some performance degradation.

```
1  ...
2  int main() {
3    ...
4    for(int i = 0; i < ndevs; i++) {
5      d_A[i] = clCreateBuffer(ctxt, ..., size, ...);
6      clAttachBufferToDevice(d_A[i], devs[i]);
7      d_B[i] = clCreateBuffer(ctxt, ..., size, ...);
8      clAttachBufferToDevice(d_B[i], devs[i]);
9      d_C[i] = clCreateBuffer(ctxt, ..., size, ...);
10     clAttachBufferToDevice(d_C[i], devs[i]);
11   }
12   ...
13 }
```

Figure 5. The OpenCL application in Figure 3 is modified to use `clAttachBufferToDevice`.

For example, since each buffer `d_A[i]` is accessed by only one device `devs[i]` in the OpenCL application in Figure 3, we can add the new API function call for `d_A[i]` in the code. The same thing is true for `d_B[i]` and `d_C[i]`. The modification is shown in Figure 5. Lines 39 - 41 in Figure 3 have been changed to lines 5 - 10 in Figure 5.

3.7 Queueing Optimization

A virtual command sometimes does not even need to be enqueued by the runtime if a memory object accessed by the command is attached to a compute device by `clAttachBufferToDevice`. When the command scheduler schedules a virtual command, it does not deliver the command to the issuer in general. However, before discarding it, the scheduler still needs to process the command to obtain the information that is used to schedule other commands (*e.g.*,

the latest device list and event synchronization). If a virtual command does not contribute to obtaining such information, it can be dropped without inserting it into a command-queue. Note that conflict detection occurs in each `clEnqueue`$\cdots$ call, not by the command scheduler.

Conditions under which a virtual command does not need to be enqueued are as follows:

- It does not have any event in its event wait list.
- Each memory object accessed by it is attached to a virtual device.

If a virtual command meets these conditions, it is discarded by the runtime in its `clEnqueue`$\cdots$ call. However, a device-memory-read command (*e.g.*, `clEnqueueReadBuffer`) always needs to be enqueued to a command-queue.

The queueing optimization significantly reduces the queueing overhead in SnuCL-D.

4. Evaluation

In this section, we evaluate SnuCL-D using a large-scale CPU cluster and a medium-scale GPU cluster. In addition, we compare SnuCL-D with SnuCL and MPI-Fortran.

4.1 Methodology

The implementation of SnuCL-D is based on SnuCL[23, 24] and OpenCL 1.2[22]. SnuCL is an open-source OpenCL framework for clusters and the only centralized framework that has been evaluated with a large-scale cluster. We compare SnuCL-D with SnuCL using a microbenchmark. Then, we show the effectiveness of the proposed techniques. We also compare SnuCL-D with MPI-Fortran.

Systems used. We use two different clusters: a large-scale 512-node CPU cluster and a medium-scale 36-node GPU cluster. Their configurations are summarized in Table 3. No dedicated CPU core to the SnuCL or SnuCL-D runtime exists in the CPU cluster. The runtime runs concurrently with the application on CPU cores.

	Large-scale CPU cluster	**Medium-scale GPU cluster**
Number of nodes	512	36
CPUs per node	2×Intel 2.93Ghz quad-core Xeon x5570	2×Intel 2.0Ghz octa-core Xeon E5-2650
GPUs per node	N/A	4×AMD Radeon HD 7970 (2GB global memory per GPU)
Memory	24GB per node	128GB per node
Operating system	Red Hat Enterprise Linux 5.3	Red Hat Enterprise Linux 6.3
Interconnect	Mellanox Infiniband QDR	
OpenCL	AMD APP SDK v2.9	AMD APP SDK v2.8
MPI	Open MPI 1.6.3	Open MPI 1.6.4
C compiler	GCC 4.4.6	
Fortran compiler	GNU Fortran 3.4.6	N/A

Table 3. System Configurations.

Application	**Source**	**Problem size**	
		Large-scale CPU cluster	**Medium-scale GPU cluster**
blackscholes	PARSEC	64M options	128M options
BinomialOption	AMD SDK	1M samples	1M samples
CP	Parboil	16K×16K	16K×16K
N-body	NVIDIA	2.5M bodies	10M bodies
MatrixMul	NVIDIA	16K×16K	10752×10752
EP	NPB	class E	class E
FT	NPB	class D	class C
CG	NPB	class E	class C
MG	NPB	class E	class C
SP	NPB	class E	class D
BT	NPB	class E	class D

Table 4. Applications Used.

Applications used. We use benchmark applications from different benchmark suites: the SNU NPB suite[43], PARSEC[12], NVIDIA SDK[35], AMD[4], and Parboil[45]. Table 4 lists their sources and problem sizes. The host programs in all the OpenCL applications are single-threaded.

The SNU NPB suite[43] is an OpenCL implementation of the NAS Parallel Benchmarks (NPB) suite[34]. It provides OpenCL implementations of NPB applications for multiple OpenCL compute devices. Some of them contain OpenCL collective communication extension API calls, such as `clEnqueueAlltoAllBuffer`, proposed by SnuCL.

Since original NPB applications are written in Fortran using MPI, some of them require the number of MPI processes be a square number (*i.e.,* the square of an integer). Thus, the corresponding SNU NPB applications require also the number of OpenCL compute devices be a square number. The AMD APP SDK v2.9 on the CPU cluster configures all the CPU cores in a node as a single CPU compute device. To make the number of devices a square number, we divide a CPU device into two sub-devices using a standard OpenCL API function `clCreateSubDevices`. Thus, we use 32 (256 cores, 64 devices), 128 (1024 cores, 256 devices), and 512 (4096 cores, 1024 devices) nodes of the CPU cluster.

`Blackscholes` is a multithreaded C application. We manually translate it to an OpenCL application for multiple devices. Since `BinomialOption`, `CP`, `N-body`, and `MatrixMul` are OpenCL applications originally for a single compute device, we modify them to work for multiple compute devices by distributing their workload across the compute devices.

We also modify each OpenCL application to make a different version that uses the `clAttachBufferToDevice` API function described in Section 3.6.

4.2 Evaluation with a Microbenchmark

To show the effectiveness of our decentralized approach over the centralized approach (*e.g.*, SnuCL), we run an OpenCL

microbenchmark on the large-scale CPU cluster. Assume that the cluster has N nodes, and each node has a device D_i. There are N 16-byte buffers ($B_0 - B_{N-1}$) in the microbenchmark, and it copies the contents of B_i to other buffers B_j ($i \neq j$) on each device D_i using `clEnqueueCopyBuffer` in each iteration. Thus, the total number of device-memory-copy commands in each iteration is $N * (N - 1)$. The total number of iterations is 100.

Unlike the centralized approach, there is no need to deliver control messages to the source and destination nodes in SnuCL-D when executing the device-memory-copy command. They know each other already because they are running the same host program.

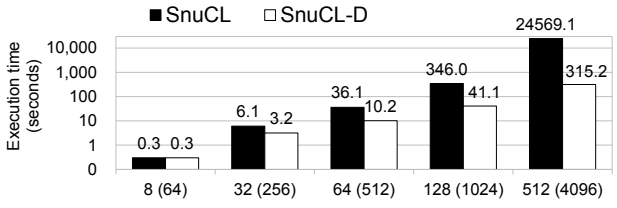

Figure 6. Comparison between the centralized approach (SnuCL) and the decentralized approach (SnuCL-D) using a microbenchmark.

Figure 6 shows the evaluation result. The x-axis shows the number of nodes, and the number in parentheses represents the number of CPU cores. The y-axis in logarithmic scale shows the execution time in seconds. When the number of nodes is small, the performance of SnuCL-D (the decentralized approach) is similar to that of SnuCL (the centralized approach). However, for more than 32 nodes (256 cores), SnuCL-D outperforms SnuCL significantly. SnuCL-D is 78 times faster than SnuCL for 512 nodes with 4096 CPU cores.

	The total number of `clAttachBufferToDevice` calls inserted	The total number of OpenCL memory objects
blackscholes	7	7
BinomialOption	2	2
CP	2	2
N-body	2	2
MatrixMul	3	3
EP	6	6
FT	17	17
CG	25	25
MG	20	20
SP	37	37
BT	46	46

Table 5. Number of `clAttachBufferToDevice` Calls.

4.3 Application Characteristics

clAttachBufferToDevice calls. Table 5 shows how many `clAttachBufferToDevice` calls are inserted in each application. It shows that each memory object is accessed by only one compute device for all applications. Thus, this type of memory object accesses is very common in reality.

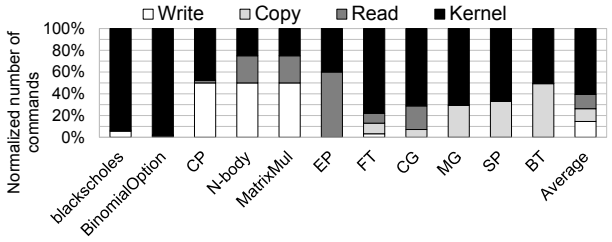

Figure 7. Distribution of frequently used OpenCL commands executed in each application for 512 nodes.

Distribution of commands executed. Figure 7 shows the distribution of frequently used OpenCL commands executed in each application on 512 nodes. They include device-memory-write (Write), device-memory-copy (Copy), device-memory-read (Read), and kernel-execution (Kernel) commands. On average, kernel commands are the most frequently executed commands (60.6%), and device-memory-read commands takes 13.2%.

We see that kernel execution, device-memory-copy, and device-memory-write commands (86.8% on average) are executed more frequently than device-memory-read commands (13.2% on average). As a result, the degree of performance degradation by the device-memory-read commands is much less than the degree of performance improvement by other commands.

4.4 Large-scale CPU Cluster

Figure 8 shows performance comparison between MPI-Fortran, SnuCL, and SnuCL-D on the large-scale CPU cluster. The x-axis shows the number of nodes, and the number in parentheses represents the number of CPU cores. The y-axis shows the speedup over 256 MPI-Fortran processes running on 32 nodes (*i.e.*, 256 CPU cores). E-class NPB applications do not run on less than 32 nodes because of the total memory size. Thus, we run them on 32, 128, and 512 nodes. An exception is FT. We use the D-class input for FT because the E-class input requires more memory space than others. Since `blackscholes`, `BinomialOption`, `CP`, `N-body`, and `MatrixMul` do not have an MPI-Fortran version, we obtain their speedup over SnuCL on 32 nodes.

The bars labeled MPI-Fortran and SnuCL show the performance of MPI-Fortran and SnuCL, respectively. The bar labeled SnuCL-D (decentralization only) shows the speedup of SnuCL-D only with the decentralization technique. The `clAttachBufferToDevice` API function and the queueing optimization technique described in Section 3.7 are not used. The bar labeled SnuCL-D (decentralization + clAttachBufferToDevice) shows the speedup of SnuCL-D with the decentralization technique and `clAttachBufferToDevice` calls in each application. Finally, SnuCL-D represents the speedup of SnuCL-D with all the proposed techniques. It includes the queueing optimization technique in addition to the

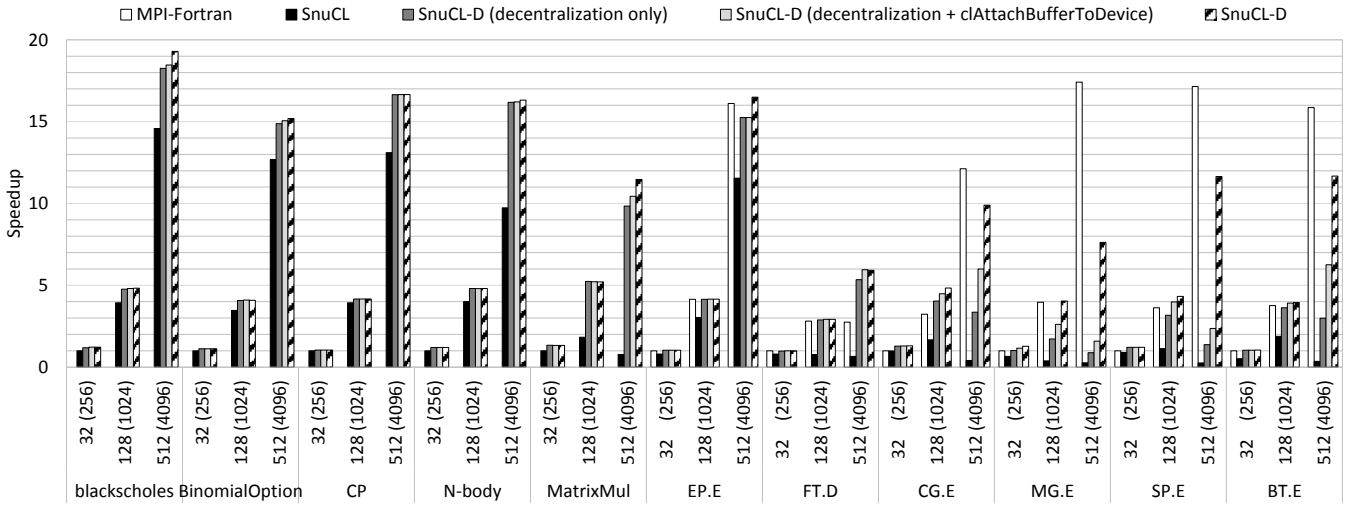

Figure 8. Comparison between MPI-Fortran, SnuCL, and SnuCL-D on the large-scale CPU cluster (speedup over 256 MPI-Fortran processes on 32 nodes with 256 CPU cores).

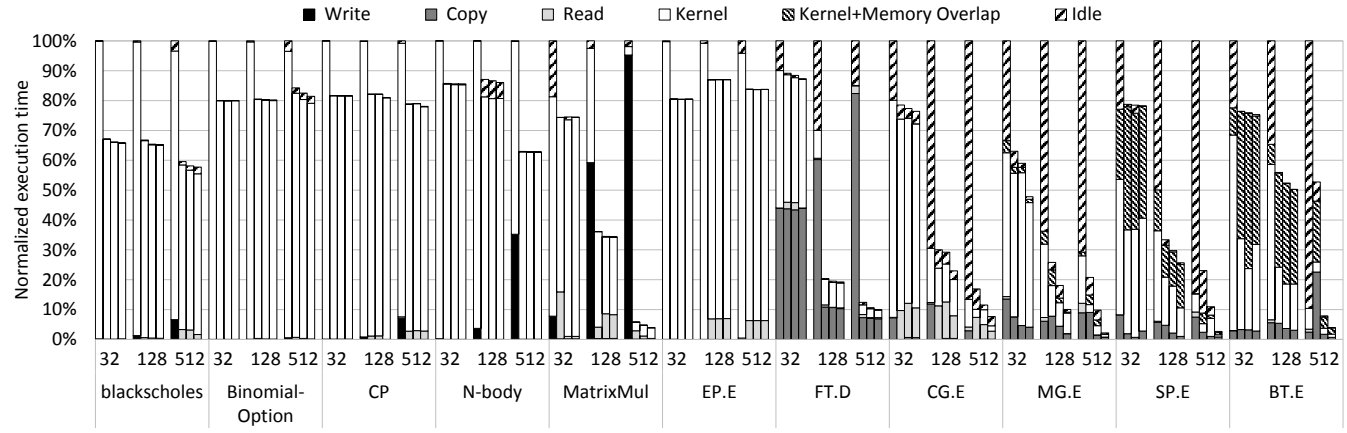

Figure 9. Execution time breakdown of each application on the large-scale CPU cluster. For each cluster configuration, there are four bars that correspond to SnuCL, SnuCL-D (decentralization only), SnuCL-D (decentralization + clAttachBufferToDevice), and SnuCL-D from left to right in turn. The bars are normalized to SnuCL.

decentralization technique and `clAttachBufferToDevice` calls.

Figure 9 shows the execution time breakdown for different OpenCL commands on the large-scale CPU cluster. We measure the accumulated execution time of each type of commands executed on the device with its ID equal to 0 for 32-, 128-, and 512-node configurations. We insert instrumentation code around `clEnqueue`$\cdots$ API calls, and the execution time includes the command-queueing, command-scheduling, command-delivery, and host-data-transfer overheads. The commands include device-memory-write (Write), device-memory-copy (Copy), device-memory-read (Read), and kernel-execution (Kernel) commands. Kernel+Memory Overlap is the case where a kernel command and device-memory (write/copy/read) commands are executed simultaneously on the device. Idle is the idle time of the device. For each cluster configuration, there are four bars that correspond to SnuCL, SnuCL-D (decentralization only), SnuCL-D (decentralization + clAttachBufferToDevice), and SnuCL-D from left to right in turn. The bars are normalized to SnuCL.

Comparison with SnuCL. The execution time of `blackscholes`, `BinomialOption`, and `CP` is dominated by the execution time of their kernels (Figure 9). The performance gap in Figure 8 between SnuCL and SnuCL-D is due to the command delivery overhead. Since the host node always delivers commands to compute devices in SnuCL, the delivery overhead increases as the number of nodes increases. Thus, SnuCL-D scales better than SnuCL.

For `N-body`, SnuCL with 512 nodes is noticeably slow compared with SnuCL-D. The gap for `MatrixMul` is dramatic for 128 and 512 nodes. For these applications, the portion of the execution time taken by write commands in SnuCL becomes bigger as the number of nodes increases. For a write command, both control messages and data should be deliv-

# of nodes (# of cores)	Device	black-scholes	Binomial-Option	CP	N-body	Matrix-Mul	EP.E	FT.D	CG.E	MG.E	SP.E	BT.E
32 (256)	All	6,848	6,528	2,688	256	256	320	17,728	2,353,153	2,001,680	4,623,168	1,625,856
	V	6,634	6,324	2,604	248	248	310	17,174	2,279,616	1,939,408	4,478,694	1,575,048
	VQO	0	0	0	0	0	0	0	5,252	18,918	48,108	24,108
128 (1024)	All	27,392	26,112	10,752	1,024	1,024	1,280	70,912	9,412,609	7,519,568	36,977,920	12,672,000
	V	27,178	25,908	10,668	1,016	1,016	1,270	70,358	9,339,072	7,462,396	36,689,030	12,573,000
	VQO	0	0	0	0	0	0	0	5,252	17,082	96,204	48,204
512 (4096)	All	109,568	104,448	43,008	4,096	4,096	5,120	283,648	37,650,433	28,381,808	295,793,664	100,036,608
	V	109,354	104,244	42,924	4,088	4,088	5,110	283,094	37,576,896	28,329,685	295,215,942	99,841,224
	VQO	0	0	0	0	0	0	0	5,252	15,348	192,396	96,396

Table 6. Number of Commands Enqueued.

ered to the device in SnuCL. Since array `Pos` in `N-body` and matrix `B` in `MatrixMul` (it computes $C = A \times B$) need to be transferred to every node from the host node, the host-data-transfer overhead increases as the number of nodes increases in SnuCL. On the other hand, every node in SnuCL-D already has the data. As a result, the performance gap between SnuCL and SnuCL-D becomes bigger as the number of nodes increases.

`EP` is an embarrassingly parallel application, and it spends a large portion of its execution time on kernel computation. The reason for the performance gap between SnuCL and SnuCL-D is similar to the case of `blackscholes`, `BinomialOption`, and `CP`.

For `FT`, the execution time of device-memory-copy commands becomes dominant as the number of nodes increases. Since `clEnqueueAlltoAllBuffer` commands in `FT` make the host deliver a point-to-point communication message to each compute node in SnuCL, the command delivery overhead becomes much more severe as the number of nodes increases. This is the reason why the performance gap between SnuCL and SnuCL-D becomes bigger as the number of nodes increases.

Table 6 shows the total number of commands enqueued in each node (All). V stands for the number of virtual commands executed, and VQO stands for the number of virtual commands executed after queueing optimization.

As shown in Table 6, `blackscholes`, `BinomialOption`, `CP`, `N-body`, `MatrixMul`, `EP`, and `FT` execute a relatively small number of commands. This implies that the command queueing, scheduling, and delivery overheads described in Section 1.1 are not significant for these applications. Thus, SnuCL-D (decentralization + clAttachBufferToDevice) and SnuCL-D do not improve performance that much compared with SnuCL-D (decentralization only). However, all the decentralized versions of SnuCL-D are faster than SnuCL. The larger the number of nodes, the bigger the performance gap.

`CG`, `MG`, `SP`, and `BT` execute a relatively large number of commands (Table 6). In addition, for these applications, Idle in SnuCL in Figure 9 becomes dominant as the number of nodes increases. This implies that the four types of overheads are significant. They become more significant as the number of nodes increases in SnuCL. On the other hand, the overheads are less significant in SnuCL-D. As a result, SnuCL-D scales much better than SnuCL.

clAttachBufferToDevice and queueing optimization. SnuCL-D (decentralization + clAttachBufferToDevice) and SnuCL-D do improve performance significantly for `CG`, `MG`, `SP`, and `BT` compared with SnuCL-D (decentralization only). Using `clAttachBufferToDevice` calls (SnuCL-D (decentralization + clAttachBufferToDevice)) outperforms SnuCL-D (decentralization only) because the scheduling overhead is significantly reduced.

Queueing optimization makes SnuCL-D outperform SnuCL-D (decentralization + clAttachBufferToDevice) for `CG`, `MG`, `SP`, and `BT`. It reduces the queueing overhead and further reduces the scheduling overhead. We see that the performance improvement becomes more significant as the number of nodes increases.

Table 6 shows the total number of virtual commands executed after queueing optimization (VQO). We see that it is much smaller than that of the case without queueing optimization in each application. Note that the number of commands that do not need to be enqueued increases as the number of nodes increases because the number of virtual devices increases in each node.

Speedup of SnuCL-D over SnuCL. Table 7 summarizes the speedup of SnuCL-D over SnuCL. Basically, the larger the number of nodes, the bigger the performance gap between SnuCL-D and SnuCL. For `CG`, `MG`, `SP`, and `BT`, the gap is much bigger because they execute a large number of commands. This implies that the command-queueing, command-scheduling, command-delivery, and host-data-transfer overheads are relatively large in these applications. Consequently, inserting `clAttachBufferToDevice` and applying queueing optimization are more effective for them. For 512 nodes, SnuCL-D is more than 20 times faster than SnuCL. Especially, SnuCL-D is 45.31 times faster than SnuCL for `SP`.

Comparison with MPI-Fortran. As shown in Figure 8, the performance of SnuCL-D is comparable to that of MPI-Fortran up to 128 nodes for `FT`. However, MPI-Fortran performs significantly worse than SnuCL-D for 512 nodes for `FT`.

# of nodes (# of cores)	blackscholes	BinomialOption	CP	N-body	MatrixMul	EP.E	FT.D	CG.E	MG.E	SP.E	BT.E
32 (256)	1.23	1.12	1.05	1.20	1.33	1.29	1.24	1.32	1.93	1.34	2.00
128 (1024)	1.23	1.18	1.06	1.20	2.84	1.36	3.74	2.90	10.68	3.81	2.10
512 (4096)	1.32	1.20	1.27	1.68	14.87	1.43	9.03	24.15	28.93	45.31	32.85

Table 7. Speedup of SnuCL-D over SnuCL on the Large-scale CPU Cluster.

Since there are 4096 MPI-Fortran processes for 512 nodes, which is bigger than the number of elements in the z-axis, MPI-Fortran performs one more level of `MPI_Alltoall` communication than SnuCL-D.

For `CG`, `MG`, `SP`, and `BT`, SnuCL-D performs slightly better than MPI-Fortran up to 128 nodes. These applications execute a large number of `clEnqueueCopyBuffer`. Since the number of MPI-Fortran processes is four times larger than the number of OpenCL devices in SnuCL-D, the amount of communication in MPI-Fortran is bigger than that in SnuCL-D. This is the reason why SnuCL-D outperforms MPI-Fortran up to 128 nodes. For 512 nodes, MPI-Fortran performs better than SnuCL-D because the amount of work in the OpenCL kernel for 512 nodes is much smaller than that in the kernel for 32 or 128 nodes. In other words, the execution time of a kernel is not big enough to amortize the inherent overhead of the vendor-specific OpenCL runtime used in SnuCL-D. On closer inspection, we find that all kernels in `CG`, `MG`, `SP`, and `BT` have an execution time less than 100 milli-seconds for 512 nodes. Since the largest input class allowed for the NPB applications is class E, we cannot increase the input size further to increase the amount of work. We expect that the performance of SnuCL-D for 512 nodes is comparable to that of MPI-Fortran for an input size bigger than class E.

Weak scalability. Strong scalability is useful to estimate the runtime overhead on each processor because the overhead takes more portion as the number of processors increases when the input size of an application is fixed. To see weak scalability, the input size per processor needs to be fixed while the number of processors varies. It is useful to estimate the communication overhead between processors. As the number of processors increases, the communication overhead becomes dominant in the execution time. So far, we have seen the strong scalability of SnuCL-D and SnuCL.

Since it is almost impossible to modify the input sets of the NPB applications to make the amount of work proportional to the number of processors, we show only the experimental results of the non-NPB applications in Figure 10 to see their weak scalability. The x-axis shows the number of nodes for each application. The y-axis shows parallel efficiency E over one node for each of SnuCL and SnuCL-D. When S is the speedup on N nodes over one node, E is given by $E = S/N \times 100\%$. If an application is completely weakly scalable, E is 100%.

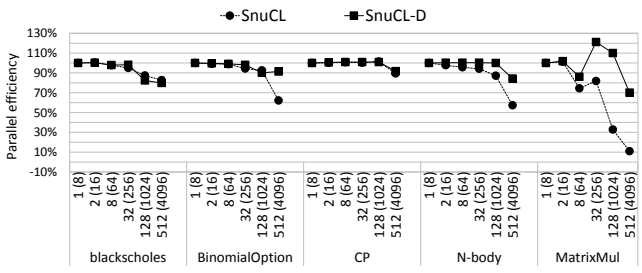

Figure 10. Weak scalability of non-NPB applications.

For `blackscholes` and `CP`, SnuCL-D is comparable to SnuCL because kernel executions dominate their execution time. For `BinomialOption`, `N-body`, and `MatrixMul`, SnuCL-D scales better than SnuCL. As the number of nodes grows, the efficiency gap between SnuCL and SnuCL-D grows larger. This implies that the communication overhead (the host-data-transfer overhead for `N-body` and `MatrixMul`) of SnuCL is more severe than that of SnuCL-D.

Since the time complexity of `MatrixMul` is O(N^3), the amount of computation grows faster than that of data (O(N^2)). That is, the amount of computation is not linearly proportional to the input data size in `MatrixMul`. Consequently, making the input data size be linearly proportional to the number of devices does not keep the per-device amount of work constant. For example, assume $N = 1536$ when we use just one device. To keep the per-device amount of work constant, say for 16 devices, N must be $\sqrt[3]{1536^3 * 16} = 3870$. This implies that the per-device data size becomes smaller when we increase the number of devices to see weak scalability. The effect of caches on the smaller data explains the superlinear speedup of SnuCL-D at 32 and 128 nodes. However, at 512 nodes, the communication overhead becomes bigger than the benefit of the smaller data size resulting in decreased efficiency.

4.5 Medium-scale GPU Cluster

Figure 11 shows the performance comparison of SnuCL and SnuCL-D on the medium-scale GPU cluster. The x-axis shows the number of nodes and the y-axis shows the speedup of SnuCL-D. Numbers in the parentheses represent numbers of GPU devices. Because of the memory requirement of each application, the speedup is obtained over SnuCL with 8 GPU devices on 2 nodes for all application except SP and BT. Note that SP and BT require the number of devices be a square number. The speedups of SP and BT are obtained over SnuCL with 16 GPU devices and 36 GPU devices, respectively.

The performance differences between SnuCL and SnuCL-D in `blackscholes`, `BinomialOption`, `CP`, `N-body`, `MatrixMul`, and `EP` are similar to those appeared in the experiment on the large-scale CPU cluster.

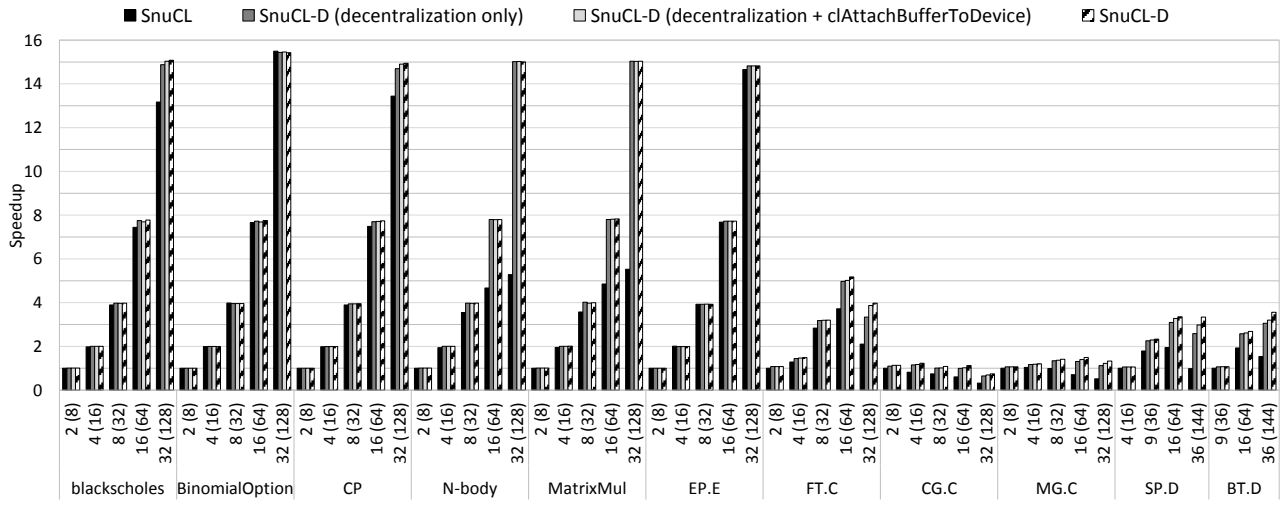

Figure 11. Comparison between SnuCL and SnuCL-D on the medium-scale GPU cluster.

Note that the host-data-transfer overhead of a GPU device is bigger than that of a CPU device in SnuCL (the host → a compute node → a GPU). Thus, the performance gap between SnuCL and SnuCL-D is observed even with a relatively small number of nodes compared to the large-scale CPU cluster.

Unlike the experimental result on the large-scale CPU cluster, SnuCL-D does not scale well for `FT`, `CG`, `MG`, and `SP`. The reason is the relatively small input size. The total amount of work in each kernel is too small to amortize the vendor-specific OpenCL runtime overhead. In addition, the computing power of a GPU is much bigger than that of a CPU. Since the total amount of device memory in a GPU is limited, we cannot increase the input size further.

SnuCL-D performs better than SnuCL especially for a large number of nodes. Since `CG`, `MG`, `SP`, and `BT` execute a large number of commands, the command-queueing and command-scheduling overheads in SnuCL are more significant than those of SnuCL-D.

5. Related Work

There have been many studies performed to enable OpenCL applications to run on a cluster as whole[3, 6, 14, 21, 23, 25, 47, 49]. In dOpenCL[21], the client executes the OpenCL host program while servers provide accesses to their devices over the network. *cl*OpenCL[3] provides wrapper functions for OpenCL host API functions, which call vendor-specific OpenCL host API functions. It uses Open-MX as a communication library. Hybrid OpenCL[6] is based on the FOXC OpenCL runtime and includes a network layer to support communication between nodes. There are several open-source projects: SocketCL[14], CLara[25], DistributedCL[47], CLuMPI[49], and SnuCL[23]. SnuCL proposes also collective communication extensions to OpenCL to boost performance.

rCUDA[16, 39, 40] proposes a CUDA framework that shares GPUs in a cluster to save energy consumption. DS-CUDA[37] is a GPU virtualization tool to use GPUs located in different nodes. DS-CUDA also supports redundant computations to increase reliability.

The aforementioned approaches all have a centralized host node that coordinates computations performed by other nodes in the cluster. Thus, the host node may become a performance bottleneck. All the previous approaches but SnuCL evaluate their frameworks with a small-scale cluster.

Some studies propose different interfaces to exploit accelerators in a cluster. libWater[18] proposes an SQL-like programming model to simplify the management of a large number of OpenCL devices available in a cluster. MGP[9] provides an OpenMP-like API layer that exploits the MOSIX virtual OpenCL layer to execute programs on a heterogeneous cluster. Charm++[1] is an object-based message-passing programming model. It is also extended to support heterogeneous clusters and to provide load balancing[27].

StarPU[7] is a task programming library for heterogeneous architectures. StarPU-MPI[8] extends StarPU with MPI. It exploits the task-based programming paradigm of StarPU for GPU clusters. OmpSs[17] is another task-based programming model based on OpenMP. It is also extended with MPI to exploit GPU clusters[13]. APC+[19] is yet another task-based programming model for GPU clusters with load-balancing support via work stealing.

StarPU and OmpSs are different from the OpenCL programming model in expressing the execution order of tasks. They ask programmers to specify tasks and data accessed by each task. Based on the information provided by the programmer, the runtime builds a task dependence graph and execute each task by checking its data dependences in the task dependence graph. On the other hand, the execution order between commands in OpenCL are enforced by event synchronizations

that are explicitly specified by the programmer if necessary. The OpenCL runtime does not build a data dependence graph between commands.

The redundant host computation used in SnuCL-D introduce non-determinacy in command scheduling across different nodes. SnuCL-D implements a solution to single-threaded OpenCL host programs. *Deterministic multithreading*[10, 11, 32, 38] is a solution to multithreaded OpenCL host programs to remove such non-determinacy.

Kendo[38] is a software-only deterministic multithreading technique for data-race-free programs. Grace[11] is a deterministic multithreading runtime system for fork-join based C/C++ programs. Both approaches have a limitation that arbitrary multithreaded programs cannot be executed deterministically. CoreDet[10] is a combination of a fully automatic compiler and a runtime system for deterministic execution of arbitrary C/C++ multithreaded programs. DTHREADS[32] is a deterministic replacement of the `pthreads` library for arbitrary C/C++ programs.

6. Conclusions

In this paper, we propose a scalable and distributed OpenCL framework called SnuCL-D for large-scale clusters. SnuCL-D overcomes the performance bottleneck incurred by four different types of overheads in previous centralized approaches, namely command-queueing, command-scheduling, command-delivery, and host-data-transfer overheads.

The experimental results show that SnuCL-D scales much better than SnuCL for applications that execute a large number of commands. SnuCL-D is more than 20 times faster than SnuCL on 512 nodes for these applications. They also indicates that SnuCL-D and MPI-Fortran are comparable in performance for most of the applications up to 128 nodes. For 512 nodes, MPI-Fortran outperforms SnuCL-D because the amount of work in an OpenCL kernel is too small to amortize the inherent overhead of the vendor-specific OpenCL runtime. We expect that SnuCL-D and MPI-Fortran are comparable in performance for more than 128 nodes with much bigger input sizes.

SnuCL-D efficiently and transparently extends the OpenCL programming model to clusters. Neither modifying the original code nor using any communication library is necessary to use SnuCL-D. An OpenCL application written for multiple devices can be executed efficiently on a large-scale cluster as a whole. The source code of SnuCL-D is publicly available at the URL `http://aces.snu.ac.kr`.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments and suggestions. We would also like to thank Prof. Adrian Sampson for shepherding the paper and providing many helpful feedbacks. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. 2013R1A3A2003664). ICT at Seoul National University provided research facilities for this study. When this work was done, Junghyun Kim was a Ph.D. student at Seoul National University. He is currently with Samsung Electronics Co., Ltd., Korea. His current email address is `jh0822.kim@samsung.com`. A part of this work has been done when Jungwon Kim was a Ph.D. student at Seoul National University. He is currently with Future Technologies Group at Oak Ridge National Laboratory, USA. His current email address is `kimj@ornl.gov`.

References

[1] Charm++. Website. `http://charm.cs.uiuc.edu/`.

[2] G. Aloisio and S. Fiore. Towards Exascale Distributed Data Management. *International Journal of High Performance Computing Applications*, 23(4):398–400, 2009.

[3] A. Alves, J. Rufino, A. Pina, and L. P. Santos. clOpenCL - Supporting Distributed Heterogeneous Computing in HPC Cluster. In *Euro-Par 2012: Parallel Processing Workshops, Revised Selected Papers*, pages 112–122. Springer-Verlag, Berlin, Heidelberg, 2013.

[4] AMD. AMD APP SDK. Website, January 2014. `http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/`.

[5] 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.

[6] R. Aoki, S. Oikawa, R. Tsuchiyama, and T. Nakamura. Hybrid OpenCL: Connecting Different OpenCL Implementations over Network. In *Proceedings of the 10th IEEE International Conference on Computer and Information Technology*, pages 2729–2735, Washington, DC, USA, 2010. IEEE Computer Society.

[7] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: a Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[8] C. Augonnet, O. Aumage, N. Furmento, S. Thibault, and R. Namyst. StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators. Research Report RR-8538, May 2014.

[9] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh. A Package for OpenCL based Heterogeneous Computing on Clusters with Many GPU Devices. In *Proceedings of 2010 IEEE International Conference on Cluster Computing Workshops and Posters*, pages 1–7, September 2010.

[10] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 53–64, 2010.

[11] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe Multithreaded Programming for C/C++. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 81–96. ACM, 2009.

[12] 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. ACM, 2008.

[13] J. Bueno, J. Planas, A. Duran, R. Badia, X. Martorell, E. Ayguade, and J. Labarta. Productive Programming of GPU Clusters with OmpSs. In *Proceedings of the 26th IEEE International Parallel Distributed Processing Symposium*, IPDPS '12, pages 557–568, May 2012.

[14] Casten. SocketCL. Website. `http://sourceforge.net/projects/socketcl`.

[15] F. Darema, D. A. George, N. V. A., and G. F. Pfister. A Single-program-multiple-data Computational Model for EPEX/FORTRAN. *Parallel Computing*, 7(1):11–24, April 1988.

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

[17] A. Duran, E. Ayguade, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: A Proposal for Programming Heterogeneous Multi-core Architectures. *Parallel Processing Letters*, 21(02): 173–193, 2011.

[18] I. Grasso, S. Pellegrini, B. Cosenza, and T. Fahringer. LibWater: Heterogeneous Distributed Computing Made Easy. In *Proceedings of the 27th ACM International Conference on Supercomputing*, ICS '13, pages 161–172, 2013.

[19] T. D. Hartley, E. Saule, and Ümit V. Çatalyürek. Improving Performance of Adaptive Component-based Dataflow Middleware. *Parallel Computing*, 38(6–7):289–309, 2012.

[20] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-performance Code Generation for Stencil Computations on GPU Architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 311–320, 2012.

[21] P. Kegel, M. Steuwer, and S. Gorlatch. dOpenCL: Towards a Uniform Programming Approach for Distributed Heterogeneous Multi-/Many-Core Systems. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium Workshops PhD Forum*, pages 174–186, May 2012.

[22] Khronos Group. *OpenCL 1.2 Specification*. Khronos Group, November 2012. `http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/`.

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

[24] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. OpenCL as a Programming Model for GPU Clusters. In *Languages and Compilers for Parallel Computing: 24th International Workshop, LCPC 2011, Fort Collins, CO, USA, September 8-10, 2011. Revised Selected Papers*, LNCS 7146, pages 233–248. Springer-Verlag, Berlin, Heidelberg, 2013.

[25] B. König. CLara - OpenCL Across the Net. Website. `http://sourceforge.net/projects/clara`.

[26] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective Automatic Parallelization of Stencil Computations. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 235–244, 2007.

[27] D. M. Kunzman and L. V. Kalé. Programming Heterogeneous Clusters with Accelerators Using Object-Based Programming. *Scientific Programming*, 19(1):47–62, 2011.

[28] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[29] J. Lee and D. A. Padua. Hiding Relaxed Memory Consistency with Compilers. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '00, pages 111–122, October 2000.

[30] J. Lee, D. A. Padua, and S. P. Midkiff. Basic Compiler Algorithms for Parallel Programs. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '99, pages 1–12, 1999.

[31] J. Lee, J. Kim, C. Jang, S. Kim, B. Egger, K. Kim, and S. Han. FaCSim: A Fast and Cycle-Accurate Architecture Simulator for Embedded Systems. In *Proceedings of the ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '08, pages 89–99, June 2008.

[32] T. Liu, C. Curtsinger, and E. D. Berger. DTHREADS: Efficient Deterministic Multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, 2011.

[33] MPI Forum. MPI: A Message Passing Interface Standard. Version 3. Website, 2012. `http://www.mpi-forum.org`.

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

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

[36] NVIDIA. CUDA Zone. Website, January 2014. `http://www.nvidia.com/object/cuda_home_new.html`.

[37] M. Oikawa, A. Kawai, K. Nomura, K. Yasuoka, K. Yoshikawa, and T. Narumi. DS-CUDA: A Middleware to Use Many GPUs in the Cloud Environment. In *Proceedings of 2012 SC Companion: High Performance Computing, Networking, Storage and Analysis*, pages 1207–1214, Nov 2012.

[38] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 97–108, 2009.

[39] A. J. Peña, C. Reaño, F. Silla, R. Mayo, E. S. Quintana-Ortí, and J. Duato. A Complete and Efficient CUDA-sharing Solution for HPC Clusters. *Parallel Computing*, 40(10):574–588, 2014. ISSN 0167-8191.

[40] C. Reaño, A. Peña, F. Silla, J. Duato, R. Mayo, and E. Quintana-Ortí. CU2rCU: Towards the Complete rCUDA Remote GPU Virtualization and Sharing Solution. In *Proceedings of the 19th International Conference on High Performance Computing*, HiPC '12, pages 1–10, December 2012.

[41] D. A. Reed and J. Dongarra. Exascale Computing and Big Data. *Communications of the ACM*, 58(7):56–68, July 2015.

[42] S. Ryoo, C. I. Rodrigues, S. S. Stone, J. A. Stratton, S.-Z. Ueng, S. S. Baghsorkhi, and W.-M. W. Hwu. Program Optimization Carving for GPU Computing. *Journal of Parallel and Distributed Computing*, 68 (10):1389–1401, 2008.

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

[44] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs That Share Memory. *ACM Trans. Program. Lang. Syst.*, 10 (2):282–312, April 1988.

[45] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, and W.-M. W. Liu, Geng Danieland Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical report, University of Illinois at Urbana-Champaign, March 2012. http://impact.crhc.illinois.edu/Parboil/parboil.aspx.

[46] Z. Sura, X. Fang, C.-L. Wong, S. P. Midkiff, J. Lee, and D. Padua. Compiler Techniques for High Performance Sequentially Consistent Java Programs. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 2–13, 2005.

[47] A. Tupinamba. DistributedCL. Website. `https://github.com/andrelrt/distributedcl`.

[48] O. Wolfson, S. Jajodia, and Y. Huang. An Adaptive Data Replication Algorithm. *ACM Transactions on Database Systems*, 22(2):255–314, June 1997.

[49] A. Woodland. CLuMPI (OpenCL under MPI). Website. `http://sourceforge.net/projects/clumpi`.

[50] J. Zhang, J. Lee, and P. K. McKinley. Optimizing the Java Pipe I/O Stream Library for Performance. In *Languages and Compilers for Parallel Computing: 15th Workshop, LCPC 2002, College Park, MD, USA, July 2002, Revised Papers*, LNCS 2481, pages 233–248. Springer-Verlag, Berlin, Heidelberg, 2005.