SnuCL User Manual

Release 1.2 Beta Version June 2012

Center for Manycore Programming

School of Computer Science and Engineering

Seoul National University, Seoul 151-744, Korea

http://aces.snu.ac.kr

1. Introduction

SnuCL is an OpenCL framework and freely available, open-source software developed at Seoul National University. It naturally extends the original OpenCL semantics to the heterogeneous cluster environment. The target cluster consists of a single host node and multiple compute nodes. They are connected by an interconnection network, such as Gigabit and InfiniBand switches. The host node contains multiple CPU cores and each compute node consists of multiple CPU cores and multiple GPUs. For such clusters, SnuCL provides a system image running a single operating system instance to the user. A GPU or a set of CPU cores becomes an OpenCL compute device. SnuCL allows the application to utilize compute devices in a compute node as if they were in the host node. SnuCL achieves both high performance and ease of programming in a heterogeneous cluster environment.

2. Installation

2.1. Supported Platforms

SnuCL builds on 32-bit, 64-bit flavors of Linux. SnuCL runs on a single heterogeneous CPU/GPU system (i.e. a single node), or a heterogeneous CPU/GPU clusters consisting of a single host node and multiple compute nodes. The following processors are supported by SnuCL and become SnuCL compute devices.

- x86 CPUs
- ARM CPUs
- PowerPC CPUs
- NVIDIA GPUs

To install SnuCL on a single node, see section 2.2. To install SnuCL on a cluster, see section 2.3.

2.2. Installing SnuCL on a single node

Prerequisite. To install SnuCL on a single node, you must install the following:

- The vendor-provided driver and OpenCL runtime if you want to use a GPU device (e.g. CUDA Toolkit).

Installing. Download the SnuCL framework source code from http://snucl.snu.ac.kr. The package includes the SnuCL runtime, source-to-source translators, and all libraries required by the framework.

Put the gzipped tarball in your work directory. Then, untar it and configure shell environment variables for SnuCL.

```
user@computer:~/$ tar zxvf snucl.1.2.tar.gz
user@computer:~/$ export SNUCLROOT=$HOME/snucl
user@computer:~/$ export PATH=$PATH:$SNUCLROOT/bin
```

```
user@computer:~/$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$SNUCLROOT/lib
```

Build the SnuCL distribution using a script named *install.sh*. You should specify devices in the target system as follows:

- To use a CPU device (multi-core CPUs):

```
user@computer:~/$ cd snucl/build
user@computer:~/snucl/build$ ./install.sh X86
```

- To use both a CPU device and a GPU device:

```
user@computer:~/$ cd snucl/build
user@computer:~/snucl/build$ ./install.sh X86 LEGACY
```

An example run. You should now test running a SnuCL application and check it runs correctly. The *sample* example is started on the target system by entering the following commands:

```
user@computer:~/$ cd snucl/apps/sample
user@computer:~/ snucl /apps/sample$ make
user@computer:~/ snucl /apps/sample$ ./bin/sample
[ 0] 100
[ 1] 110
[ 2] 120
[ 3] 130
[ 4] 140
[ 5] 150
[ 6] 160
[ 7] 170
[ 8] 180
[ 9] 190
[10] 200
[11] 210
[12] 220
[13] 230
[14] 240
[15] 250
[16] 260
[17] 270
[18] 280
[19] 290
```

```
[20] 300
[21] 310
[22] 320
[23] 330
[24] 340
[25] 350
[26] 360
[27] 370
[28] 380
[29] 390
[30] 400
[31] 410
user@computer:~/snucl/apps/sample$
```

2.3. Installing SnuCL on a cluster

Prerequisite. To install SnuCL for a cluster, you must install the following in both the host node and the compute nodes:

- An MPI implementation (e.g., Open MPI).
- The vendor-provided driver and OpenCL runtime if you want to use GPU devices (e.g. CUDA Toolkit).

Additionally, you must have an account on all the nodes. You must be able to ssh between the host node and the compute nodes without using a password.

Installing. You must install SnuCL in all the nodes to run OpenCL applications on the cluster.

Download the SnuCL framework source code from http://snucl.snu.ac.kr. The package includes the SnuCL runtime, source-to-source translators, and all libraries required by the framework.

Put the gzipped tarball in your work directory and untar it.

```
user@computer:~/$ tar zxvf snucl.1.2.tar.gz
```

Then, configure shell environment variables for SnuCL. Add the following configuration in your shell startup scripts (e.g., .bashrc, .cshrc, .profile, etc.)

```
export SNUCLROOT=$HOME/snucl
```

```
export PATH=$PATH:$SNUCLROOT/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$SNUCLROOT/lib
```

Build the SnuCL distribution using a script named *install.sh*. You should specify devices in the compute nodes as follows:

- To use CPU devices (multi-core CPUs) in the compute nodes:

```
user@computer:~/$ cd snucl/build
user@computer:~/snucl/build$ ./install.sh X86 CLUSTER
```

- To use both CPU devices and GPU devices in the compute nodes:

```
user@computer:~/$ cd snucl/build
user@computer:~/snucl/build$ ./install.sh X86 LEGACY CLUSTER
```

An example run. You should now test running a SnuCL application and check it runs correctly. First, you should edit *snucl_nodes* in the directory $SNUCLROOT/bin. The file specifies the nodes' hostnames in the cluster. (See section 3.2)

The *sample* example is started on the target cluster by entering the following commands:

```
user@computer:~/$ cd snucl/apps/sample
user@computer:~/ snucl /apps/sample$ make cluster=1
user@computer:~/ snucl /apps/sample$ snuclrun 1 ./bin/sample
[ 0] 100
[ 1] 110
[ 2] 120
[ 3] 130
[ 4] 140
[ 5] 150
[ 6] 160
[ 7] 170
[ 8] 180
[ 9] 190
[10] 200
[11] 210
[12] 220
[13] 230
[14] 240
[15] 250
```

```
[16] 260
[17] 270
[18] 280
[19] 290
[20] 300
[21] 310
[22] 320
[23] 330
[24] 340
[25] 350
[26] 360
[27] 370
[28] 380
[29] 390
[30] 400
[31] 410
user@computer:~/snucl/apps/sample$
```

3. Using SnuCL

3.1. Building OpenCL applications using SnuCL

With SnuCL, 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 heterogeneous system to run on the cluster without any modification.

You only need to link your applications with SnuCL libraries to run your OpenCL applications in the cluster environment. You may use the Makefile template in the directory $SNUCLROOT/apps/sample. Set the variable *cluster* to 0 if SnuCL is installed on a single node or to 1 if SnuCL is installed on a cluster.

```
EXECUTABLE := <program name>
CCFILES := <source files>
cluster := <0 or 1>
include $(SNUCLROOT)/common.mk
```

3.2. Running OpenCL applications using SnuCL

To run an OpenCL application on a single node, just execute the application. To run an OpenCL application on a cluster, you can use a script named *snuclrun* as follows:

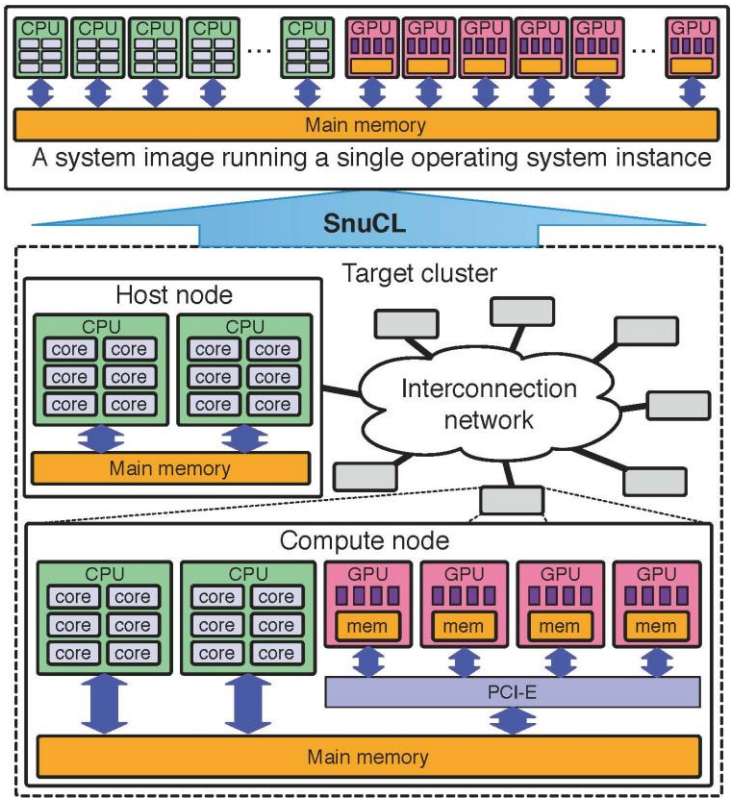
snuclrun <# of compute nodes> <program> [<program arguments>]

snuclrun uses a hostfile, $SNUCLROOT/bin/*snucl_nodes*, which specifies the nodes' hostnames in the cluster. *snucl_nodes* follows the hostfile format in the installed MPI implementation.

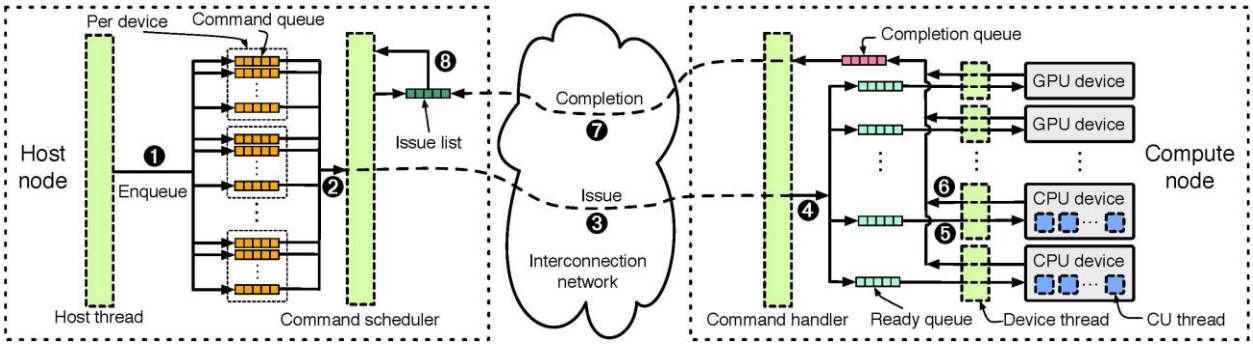
```
hostnode slots=1 max_slots=1
computenode1 slots=1 max_slots=1
computenode2 slots=1 max_slots=1
..
```

4. Understanding SnuCL

4.1. What SnuCL does with your OpenCL applications



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. 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.



This figure 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. The host thread enqueues commands to the command-queues (① in the figure). The command scheduler schedules the enqueued commands across compute devices in the cluster one by one (②).

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* (③) 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 (④). 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 (⑤). 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 (⑥). 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 (⑦).

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* (⑧).

5. Functions Supported

SnuCL follows the OpenCL core specification version 1.2 for CPU devices and 1.1 for GPU devices. This section summarizes the supported functions of the current SnuCL implementation.

5.1. Tested Platforms

SnuCL has been tested on two cluster systems (Cluster A and Cluster B). Cluster A is used to test CPU devices, and Cluster B is used to test GPU devices. Cluster A consists of the following nodes:

- Host node
  - AMD® Opteron® Processor 4184
  - Red Hat Enterprise Linux Server 6.1
  - gcc 4.4.5
  - Open MPI 1.4.3
- Compute node
  - AMD® Opteron Processor 6172
  - Red Hat Enterprise Linux Server 6.1
  - gcc 4.4.5
  - Open MPI 1.4.3

Cluster B consists of the following nodes:

- Host node
  - Intel® Xeon® Processor X5680
  - Red Hat Enterprise Linux Server 5.5
  - gcc 4.1.2

- Open MPI 1.4.1
- CUDA Toolkit 4.2

- Compute node
  - Intel® Xeon® Processor X5660
  - NVIDIA GeForce GTX 480
  - Red Hat Enterprise Linux Server 5.5
  - gcc 4.1.2
  - Open MPI 1.4.1
  - CUDA Toolkit 4.2

5.2. API Functions Supported

The following table shows all API functions in the OpenCL specification version 1.1 and version 1.2, and whether they are supported in the current SnuCL implementation. API functions marked with "(1.1)" or "(1.2)" are only in OpenCL 1.1 or OpenCL 1.2, respectively. All other functions are included in both OpenCL 1.1 and OpenCL 1.2. An "O" indicates that a function is fully supported. A "△" indicates that a function is partially supported. An empty cell indicates that a function is not supported yet.

API function	Single node		Cluster	
	CPU	GPU	CPU	GPU
Querying Platform Info				
clGetPlatformIDs	O	O	O	O
clGetPlatformInfo	O	O	O	O
Querying Devices				
clGetDeviceIDs	O	O	O	O
clGetDeviceInfo	O	O	O	O
Partitioning a Device				
clCreateSubDevices (1.2)	O			
clRetainDevice (1.2)	O		O	
clReleaseDevice (1.2)	O		O	

Contexts				
clCreateContext	O	O	O	O
clCreateContextFromType	O	O	O	O
clRetainContext	O	O	O	O
clReleaseContext	O	O	O	O
clGetContextInfo	O	O	O	O
Command Queues				
clCreateCommandQueue	O	O	O	O
clRetainCommandQueue	O	O	O	O
clReleaseCommandQueue	O	O	O	O
clGetCommandQueueInfo	O	O	O	O
Buffer Objects				
clCreateBuffer	O	O	O	O
clCreateSubBuffer	O	O		
clEnqueueReadBuffer	O	O	O	O
clEnqueueWriteBuffer	O	O	O	O
clEnqueueReadBufferRect	O	O	O	O
clEnqueueWriteBufferRect	O	O	O	O
clEnqueueCopyBuffer	O	O	O	O
clEnqueueCopyBufferRect	O	O	O	O
clEnqueueFillBuffer (1.2)	O			
clEnqueueMapBuffer	O	O		
Image Objects				
clCreateImage (1.2)	O		O	
clCreateImage2D (1.1)	O	O	O	O
clCreateImage3D (1.1)	O	O	O	O
clGetSupportedImageFormats	O	O	O	O
clEnqueueReadImage	O	O	O	O
clEnqueueWriteImage	O	O	O	O
clEnqueueCopyImage	O	O	O	O
clEnqueueFillImage (1.2)	O			
clEnqueueCopyImageToBuffer	O	O	O	O
clEnqueueCopyBufferToImage	O	O	O	O
clEnqueueMapImage	O	O		
clGetImageInfo	O	O	O	O
Querying, Unmapping, Migrating, Retaining, and Releasing Memory Objects				
clRetainMemObject	O	O	O	O
clReleaseMemObject	O	O	O	O
clSetMemObjectDestructorCallback	O	O	O	O
clEnqueueUnmapMemObject	O	O		
clEnqueueMigrateMemObjects (1.2)	O		O	
clGetMemObjectInfo	O	O	O	O

Sampler Objects				
clCreateSampler	O	O	O	O
clRetainSampler	O	O	O	O
clReleaseSampler	O	O	O	O
clGetSamplerInfo	O	O	O	O
Program Objects				
clCreateProgramWithSource	O	O	O	O
clCreateProgramWithBinary	O	O	O	O
clCreateProgramWithBuiltInKernels (1.2)	O		O	
clRetainProgram	O	O	O	O
clReleaseProgram	O	O	O	O
clBuildProgram	O	O	O	O
clCompileProgram (1.2)	O			
clLinkProgram (1.2)	O			
clUnloadPlatformCompiler (1.2)	O		O	
clUnloadCompiler (1.1)	O	O	O	O
clGetProgramInfo	O	O	O	O
clGetProgramBuildInfo	O	O	O	O
Kernel Objects				
clCreateKernel	O	O	O	O
clCreateKernelsInProgram	O	O	O	O
clRetainKernel	O	O	O	O
clReleaseKernel	O	O	O	O
clSetKernelArg	O	O	O	O
clGetKernelInfo	O	O	O	O
clGetKernelWorkGroupInfo	O	O	O	O
clGetKernelArgInfo (1.2)	O		O	
Executing Kernels				
clEnqueueNDRangeKernel	O	O	O	O
clEnqueueTask	O	O	O	O
clEnqueueNativeKernel	O		O	
Event Objects				
clCreateUserEvent	O	O	O	O
clSetUserEventStatus	O	O	O	O
clWaitForEvents	O	O	O	O
clGetEventInfo	O	O	O	O
clSetEventCallback	O	O	O	O
clRetainEvent	O	O	O	O
clReleaseEvent	O	O	O	O
Markers, Barriers and Waiting for Events				
clEnqueueMarkerWithWaitList (1.2)	O	O	O	O
clEnqueueBarrierWithWaitList (1.2)	O	O	O	O

clEnqueueMarker (1.1)	O	O	O	O
clEnqueueBarrier (1.1)	O	O	O	O
clEnqueueWaitForEvents (1.1)	O	O	O	O
Profiling Operations on Memory Objects and Kernels				
clGetEventProfilingInfo	O	O	O	O
Flush and Finish				
clFlush	O	O	O	O
clFinish	O	O	O	O

5.3. Built-in Functions Supported

The following table shows all built-in functions of the OpenCL C programming language, and whether they are supported in the current SnuCL implementation. API functions marked with "(1.1)" or "(1.2)" are only in OpenCL 1.1 or OpenCL 1.2, respectively. All other functions are included in both OpenCL 1.1 and OpenCL 1.2. An "O" indicates that a function is fully supported. A "Δ" indicates that a function is partially supported. An empty cell indicates that a function is not supported yet.

Built-in function	**CPU**	**GPU**
Work-Item Functions		
get_work_item	O	O
get_global_size	O	O
get_global_id	O	O
get_local_size	O	O
get_local_id	O	O
get_num_groups	O	O
get_group_id	O	O
get_global_offset	O	O
Math Functions		
acos	O	O
acosh	O	O
acospi	O	O
asin	O	O
asinh	O	O
asinpi	O	O
atan	O	O
atan2	O	O
atanh	O	O
atanpi	O	O

atan2pi	O	O
cbrt	O	O
ceil	O	O
copysign	O	O
cos	O	O
cosh	O	O
cospi	O	O
erfc	O	O
erf	O	O
exp	O	O
exp2	O	O
exp10	O	O
expm1	O	O
fabs	O	O
fdim	O	O
floor	O	O
fma	O	O
fmax	O	O
fmin	O	O
fmod	O	O
fract	O	O
frexp	O	O
hypot	O	O
ilogb	O	O
ldexp	O	O
lgamma	O	O
lgamma_r	O	O
log	O	O
log2	O	O
log10	O	O
log1p	O	O
logb	Δ	O
mad	O	O
maxmag	O	O
minmag	O	O
modf	O	O
nan	O	O
nextafter	O	O
pow	O	O
pown	O	O
powr	O	O
remainder	O	O

remquo	O	O
rint	O	O
rootn	O	O
round	O	O
rsqrt	O	O
sin	O	O
sincos	O	O
sinh	O	O
sinpi	O	O
sqrt	O	O
tan	O	O
tanh	O	O
tanpi	O	O
tgamma	O	O
trunc	O	O
half_cos	O	O
half_divide	O	O
half_exp	O	O
half_exp2	O	O
half_exp10	O	O
half_log	O	O
half_log2	O	O
half_log10	O	O
half_powr	O	O
half_recip	O	O
half_rsqrt	O	O
half_sin	O	O
half_sqrt	O	O
half_tan	O	O
native_cos	O	O
native_divide	O	O
native_exp	O	O
native_exp2	O	O
native_exp10	O	O
native_log	O	O
native_log2	O	O
native_log10	O	O
native_powr	O	O
native_recip	O	O
native_rsqrt	O	O
native_sin	O	O
native_sqrt	O	O

native_tan	o	o
Integer Functions		
abs	o	o
abs_diff	o	o
add_sat	o	o
hadd	o	o
rhadd	o	o
clamp	o	o
clz	o	o
mad_hi	o	o
mad_sat	o	o
max	o	o
min	o	o
mul_hi	o	o
rotate	o	o
sub_sat	o	o
upsample	o	o
popcount (1.2)	o	
mad24	o	o
mul24	o	o
Common Functions		
clamp	o	o
degrees	o	o
max	o	o
min	o	o
mix	o	o
radians	o	o
step	o	o
smoothstep	o	o
sign	o	o
Geometric Functions		
cross	o	o
dot	o	o
distance	o	o
length	o	o
normalize	o	∆
fast_distance	o	o
fast_length	o	o
fast_normalize	o	o
Relational Functions		
isequal	o	o
isnotequal	o	o

isgreater	O	O
isgreaterequal	O	O
isless	O	O
islessequal	O	O
islessgreater	O	O
isfinite	O	O
isinf	O	O
isnan	O	O
isnormal	O	O
isordered	O	O
isunordered	O	O
signbit	O	O
any	O	O
all	O	O
bitselect	O	O
select	O	O
Vector Data Load and Store Functions		
vload*n*	O	O
vstore*n*	O	O
vload_half	O	O
vload_half*n*	O	O
vstore_half	O	O
vstore_half_*rte*	O	O
vstore_half_*rtz*	O	O
vstore_half_*rtp*	O	O
vstore_half_*rtn*	O	O
vstore_half*n*_*rte*	O	O
vstore_half*n*_*rtz*	O	O
vstore_half*n*_*rtp*	O	O
vstore_half*n*_*rtn*	O	O
vloada_half*n*	O	O
vstorea_half*n*	O	O
vstorea_half*n*_*rte*	O	O
vstorea_half*n*_*rtz*	O	O
vstorea_half*n*_*rtp*	O	O
vstorea_half*n*_*rtn*	O	O
Synchronization Functions		
barrier	O	O
Explicit Memory Fence Functions		
mem_fence	O	O
read_mem_fence	O	O
write_mem_fence	O	O

Async Copies from Global to Local Memory, Local to Global Memory, and Prefetch		
async_work_group_copy	O	O
async_work_group_strided_copy	O	O
wait_group_events	O	O
prefetch	O	O
Atomic Functions		
atomic_add	O	O
atomic_sub	O	O
atomic_xchg	O	O
atomic_inc	O	O
atomic_dec	O	O
atomic_cmpxchg	O	O
atomic_min	O	O
atomic_max	O	O
atomic_and	O	O
atomic_or	O	O
atomic_xor	O	O
Miscellaneous Vector Functions		
vec_step	O	O
shuffle	O	O
shuffle2	O	O
Printf		
printf (1.2)	O	
Image Read and Write Functions		
read_imagef	O	O
read_imagei	O	O
read_imageui	O	O
write_imagef	O	O
write_imagei	O	O
write_imageui	O	O
get_image_width	O	O
get_image_height	O	O
get_image_depth	O	O
get_image_channel_data_type	O	O
get_image_channel_order	O	O
get_image_dim	O	O
get_image_array_size	O	O

5.4. Optional Features

The current SnuCL implementation supports the following optional features in OpenCL 1.2:

- The double scalar and vector types for CPU devices.
- Built-in functions for the double scalar and vector types.

6. Collective Communication Extensions

SnuCL provides collective communication operations for manipulating OpenCL buffer objects. These extensions to OpenCL are similar to MPI collective communication operations. The following table lists each collective communication operation and its MPI equivalent.

OpenCL	**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

The function

cl_int **clEnqueueBroadcastBuffer**(cl_command_queue * *cmd_queue_list*,
cl_mem *src_buffer*,
cl_uint *num_dst_buffers*,
cl_mem * *dst_buffer_list*,
size_t *src_offset*,
size_t * *dst_offset_list*,
size_t *cb*,
cl_uint *num_events_in_wait_list*,
const cl_event * *event_wait_list*,
cl_event * *event*)

enqueues commands to broadcast a buffer object identified by *src_buffer* to all buffer objects in the list identified by *dst_buffer_list*.

cmd_queue_list refers to the command-queues that are associated with the compute devices where the destination buffers identified by *dst_buffer_list* are located.

num_dst_buffers refers to the number of destination buffers identified by *dst_buffer_list*.

src_offset refers to the offset where to begin copying data from *src_buffer*.

dst_offset_list refers to the offsets where to begin copying data into *dst_buffer_list*.

cb refers to the size in bytes to copy.

event_wait_list and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed.

event returns an event object that identifies this particular broadcast command and can be used to query or queue a wait for this particular command to complete.

The function

cl_int **clEnqueueScatterBuffer**(cl_command_queue * *cmd_queue_list*,
cl_mem *src_buffer*,
cl_uint *num_dst_buffers*,
cl_mem * *dst_buffer_list*,
size_t *src_offset*,
size_t * *dst_offset_list*,
size_t *cb*,
cl_uint *num_events_in_wait_list*,
const cl_event * *event_wait_list*,
cl_event * *event*)

enqueues commands to distribute a buffer object identified by *src_buffer* to each buffer object in the list identified by *dst_buffer_list*.

cmd_queue_list refers to the command-queues that are associated with the compute devices where the destination buffers identified by *dst_buffer_list* are located.

num_dst_buffers refers to the number of destination buffers identified by *dst_buffer_list*.

src_offset refers to the offset where to begin copying data from *src_buffer*.

dst_offset_list refers to the offsets where to begin copying data into *dst_buffer_list*.

cb refers to the size in bytes to copy.

event_wait_list and *num_events_in_wait_list* specify events that need to complete before this

particular command can be executed.

event returns an event object that identifies this particular broadcast command and can be used to query or queue a wait for this particular command to complete.

The function

cl_int **clEnqueueGatherBuffer**(cl_command_queue *cmd_queue*,
cl_uint *num_src_buffers*,
cl_mem * *src_buffer_list*,
cl_mem *dst_buffer*,
size_t * *src_offset_list*,
size_t *dst_offset*,
size_t *cb*,
cl_uint *num_events_in_wait_list*,
const cl_event * *event_wait_list*,
cl_event * *event*)

enqueues commands to gather distinct buffer objects in the list identified by *dst_buffer_list* to a buffer object identified by *src_buffer*

cmd_queue refers to the command-queue that is associated with the compute device where the destination buffer identified by *dst_buffer* are located.

num_src_buffers refers to the number of source buffers identified by src*_buffer_list*.

src_offset_list refers to the offsets where to begin copying data from *dst_buffer_list*.

dst_offset refers to the offset where to begin copying data into dst*_buffer*.

cb refers to the size in bytes to copy.

event_wait_list and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed.

event returns an event object that identifies this particular broadcast command and can be used to query or queue a wait for this particular command to complete.

The function

cl_int **clEnqueueAllGatherBuffer**(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 *cb*,
cl_uint *num_events_in_wait_list*,
const cl_event * *event_wait_list*,
cl_event * *event*)

enqueues commands to gather data from all buffer objects in the list identified by src_*buffer_list* and distibute it to all buffer objects in the list identified by *dst_buffer_list.*

cmd_queue_list refers to the command-queues that are associated with the compute devices where the destination buffers identified by *dst_buffer_list* are located.

num_buffers refers to the number of buffers identified by src_*buffer_list* and *dst_buffer_list*.

src_offset_list refers to the offsets where to begin copying data from *src_buffer_list*.

dst_offset_list refers to the offsets where to begin copying data into *dst_buffer_list*.

cb refers to the size in bytes to copy.

event_wait_list and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed.

event returns an event object that identifies this particular broadcast command and can be used to query or queue a wait for this particular command to complete.

The function

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 *cb*,
cl_uint *num_events_in_wait_list*,
const cl_event * *event_wait_list*,
cl_event * *event*)

enqueues commands to distribute data from all buffer objects in the list identified by src_*buffer_list* to all buffer objects in the list identified by *dst_buffer_list* in order by index.

cmd_queue_list refers to the command-queues that are associated with the compute devices where the destination buffers identified by *dst_buffer_list* are located.

num_buffers refers to the number of buffers identified by src_*buffer_list* and *dst_buffer_list*.

src_offset_list refers to the offsets where to begin copying data from *src_buffer_list*.

dst_offset_list refers to the offsets where to begin copying data into *dst_buffer_list*.

cb refers to the size in bytes to copy.

event_wait_list and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed.

event returns an event object that identifies this particular broadcast command and can be used to query or queue a wait for this particular command to complete.

The function

cl_int **clEnqueueReduceBuffer**(cl_command_queue *cmd_queue*,
cl_uint *num_src_buffers*,
cl_mem * *src_buffer_list*,
cl_mem *dst_buffer*,
size_t * *src_offset_list*,
size_t *dst_offset*,
size_t *cb*,
cl_channel_type *datatype*,
cl_uint *num_events_in_wait_list*,
const cl_event * *event_wait_list*,
cl_event * *event*)

enqueues commands to reduce values on all buffer objects in the list identified by src_*buffer_list* and copy the value to the buffer object identified by *dst_buffer.*

cmd_queue refers to the command-queue that is associated with the compute device where the destination buffer identified by *dst_buffer* are located.

num_src_buffers refers to the number of source buffers identified by src_*buffer_list*.

src_offset_list refers to the offsets where to begin copying data from *dst_buffer_list*.

dst_offset refers to the offset where to begin copying data into dst_*buffer*.

cb refers to the size in bytes to copy.

datatype refers to the following built-in types: int, uint, float, and double.

event_wait_list and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed.

event returns an event object that identifies this particular broadcast command and can be used to query or queue a wait for this particular command to complete.

The function

cl_int **clEnqueueAllReduceBuffer**(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 *cb*,
cl_channel_type *datatype*,
cl_uint *num_events_in_wait_list*,
const cl_event * *event_wait_list*,
cl_event * *event*)

enqueues commands to reduce values on all buffer objects in the list identified by src_*buffer_list* and copy the value to all buffer objects in the list identified by *dst_buffer_list.*

cmd_queue_list refers to the command-queues that are associated with the compute devices where the destination buffers identified by *dst_buffer_list* are located.

num_buffers refers to the number of buffers identified by src_*buffer_list* and *dst_buffer_list*.

src_offset_list refers to the offsets where to begin copying data from *src_buffer_list*.

dst_offset_list refers to the offsets where to begin copying data into *dst_buffer_list*.

cb refers to the size in bytes to copy.

datatype refers to the following built-in types: int, uint, float, and double.

event_wait_list and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed.

event returns an event object that identifies this particular broadcast command and can be used to query or queue a wait for this particular command to complete.

The function

cl_int **clEnqueueReduceScatterBuffer**(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 *cb*,
cl_channel_type *datatype*,
cl_uint *num_events_in_wait_list*,
const cl_event * *event_wait_list*,
cl_event * *event*)

enqueues commands to combine values from all buffer objects in the list identified by src_*buffer_list* and scatter the results to all buffer objects in the list identified by *dst_buffer_list*.

cmd_queue_list refers to the command-queues that are associated with the compute devices where the destination buffers identified by *dst_buffer_list* are located.

num_buffers refers to the number of buffers identified by src_*buffer_list* and *dst_buffer_list*.

src_offset_list refers to the offsets where to begin copying data from *src_buffer_list*.

dst_offset_list refers to the offsets where to begin copying data into *dst_buffer_list*.

cb refers to the size in bytes to copy.

datatype refers to the following built-in types: int, uint, float, and double.

event_wait_list and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed.

event returns an event object that identifies this particular broadcast command and can be used to query or queue a wait for this particular command to complete.

The function

cl_int **clEnqueueScanBuffer**(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 *cb*,
cl_channel_type *datatype*,
cl_uint *num_events_in_wait_list*,
const cl_event * *event_wait_list*,
cl_event * *event*)

enqueues commands to perform an inclusive prefix reduction on data distributed across the buffer objects in the list identified by src*_buffer_list* and copy the results to all buffer objects in the list identified by *dst_buffer_list.*

cmd_queue_list refers to the command-queues that are associated with the compute devices where the destination buffers identified by *dst_buffer_list* are located.

num_buffers refers to the number of buffers identified by src*_buffer_list* and *dst_buffer_list*.

src_offset_list refers to the offsets where to begin copying data from *src_buffer_list*.

dst_offset_list refers to the offsets where to begin copying data into *dst_buffer_list*.

cb refers to the size in bytes to copy.

datatype refers to the following built-in types: int, uint, float, and double.

event_wait_list and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed.

event returns an event object that identifies this particular broadcast command and can be used to query or queue a wait for this particular command to complete.

7. Known Issues

The following is a list of known issues in the current SnuCL implementation.

- If SnuCL is installed on a single node, multiple GPU devices can't be used at the same time.
- Creating an OpenCL memory object (buffer or image) that has CL_MEM_USE_HOST_PTR flag is an invalid operation in the cluster environment.
- The logb built-in function has a precision problem in CPU devices.
- The following explicit conversions have precision problems in CPU devices:
  - uint_rte_float, uint_rtp_float.
  - int_rte_float, int_rtp_float, int_rtn_float.
  - float_rtp_double, float_rtn_double, float_rtz_double.
  - ulong_rte_float, ulong_rtp_float.
  - ulong_sat_rte_float, ulong_sat_rtp_float.
  - long_sat_rte_float, long_sat_rtp_float, long_sat_rtn_float.
- Variables in the __private and __local address spaces may not be aligned correctly if the version of gcc is older than 4.4.5.