VirtualCL (VCL)

Cluster Platform

Programmer and Administrator Guide and Manuals

Revised for VCL-1.24

Feb. 2016

Preface

This document presents the VirtualCL programmer and administrator guide and also the VirtualCL and SuperCL manuals.

Further information is available at *http* : *//www.mosix.org/txt_vcl.html*.

Contents

Preface **iii**

1 The VCL cluster platform **1**
1.1 Overview . . . 1
1.2 Using VCL . . . 1

2 VCL Programmer's Guide **3**
2.1 Large number of devices . . . 3
2.2 clCreateContextFromType() . . . 3
2.3 clCreateContext() - the "anonymous" method . . . 4
2.4 clCreateContext() - the "specific" method . . . 4
2.5 Presence of unwanted devices . . . 6
2.6 Other VCL issues . . . 6

3 VCL with SLURM and MPI **7**
3.1 Administrator installation guide - Using VCL with SLURM . . . 7
3.2 VCL support for MPI . . . 9

4 SuperCL **11**
4.1 Overview . . . 11
4.2 Using SuperCL . . . 11

5 Installation **13**
5.1 Automatic installation . . . 13
5.2 Manual installation . . . 13

6 Manuals **15**
6.1 For users . . . 15
6.2 For programmers . . . 15

Chapter 1

The VCL cluster platform

1.1 Overview

The VirtualCL (VCL) cluster platform is a wrapper for OpenCL™ that allows most unmodified applications to transparently utilize multiple OpenCL devices in a cluster as if all the devices are on the local computer.

The main features of VCL are:

- Works with OpenCL devices (CPUs, GPUs, Accelerators) from all vendors.
- Supports almost all OpenCL 1.1 (and 1.0) applications.
- Applications can use cluster-wide OpenCL devices
- Transparent selection of devices.
- Applications can be started on any hosting-computer, including computers without OpenCL devices.
- Supports multiple applications on the same cluster.
- Supports SuperCL, an extension of OpenCL that allows micro-programs to run efficiently on devices of remote nodes.
- Runs on Linux clusters, with or without MOSIX.

1.2 Using VCL

VCL creates a run-time environment in which cluster-wide OpenCL devices are seen as if they are located in the hosting-node. Applications need not to be aware which nodes and devices are available and where the devices are located. The resulting platform benefits OpenCL applications that can use multiple devices concurrently.

Applications may create OpenCL "contexts" that span devices on several nodes; or multiple contexts each with the devices of a different node; or a combination of the above. Another alternative is to split job(s) into several independent processes, each running on a different set of devices. Applications may be specific about which devices they include in their contexts, but for the benefit of most unmodified applications, VCL also allow environment-variables to

control the device allocation policies. By default, each context that is created includes all the devices of a single node.

In the environment created by VCL, remote nodes perform OpenCL functions on behalf of application(s) on the host(s). In order to support application concurrency on different devices, the implementation is fully thread-safe (even though this is not a requirement of OpenCL). VCL communicates between hosts and remote nodes using standard TCP/IP sockets. When running OpenCL on remote devices, network latency is the main limiting factor, so VCL attempts to minimize the number of network round-trips.

Chapter 2

VCL Programmer's Guide

Writing VCL applications is essentially like writing any other OpenCL applications. This section describes a few subtle, but important differences.

2.1 Large number of devices

OpenCL was designed for a single computer with few devices, so its device-allocation support, when a large number of devices are present, is minimal.

In a cluster environment, such as with VCL:

- The number of devices could be large.
- The number and identity of devices can change as nodes join and leave the cluster.
- There are often other tasks competing for the same devices.

Therefore selecting devices for a new OpenCL context is no longer trivial.

OpenCL has two methods of selection, each with its own issues, these are "`clCreateContextFromType()`" and "`clCreateContext()`", the later further diving into the "anonymous" method and the "specific" method.

Before discussing the peculiarities of each method, it is important to note that the OpenCL standard does not specify whether or not devices can be shared between several applications/tasks/users - this is left for each specific implementation (SDK) to decide:

VCL does not allow (including for security reasons), the simultaneous sharing of an OpenCL device between two tasks: devices are allocated exclusively per task for the lifetime of all contexts that use it (but note that multi-threaded applications are considered a single task).

2.2 clCreateContextFromType()

The OpenCL standard does not specify how many devices should belong to a new context. The SDK is free to assign just one device or all the devices. The only thing the application can specify is whether it wants only CPUs, only GPUs, only accelerators, or all three.

When there are many devices, the OpenCL library has a hard time guessing how many devices the application wants.

VCL by default will allocate all the available devices (of the specified types) on ONE node. To change that, the user must set the environment variable "`CONTEXT_POLICY`", as follows:

CONTEXT_POLICY=o selects just one device

CONTEXT_POLICY=n selects all the devices of one node (default)

CONTEXT_POLICY=m selects all devices in cluster

The same is achieved by using:

```
vclrun --policy={o|n|m}
```

If the application requires a number of devices, but not "all the devices in the cluster", the user should set the environment variable "`MAX_DEVS_PER_CONTEXT`" (the default is `MAX_DEVS_PER_CONTEXT=1024`).

The same is achieved by using:

```
vclrun --maxdevs=nn
```

In summary, if the application requires 5 GPUs, it should be run with:

```
vclrun --policy=m --maxdevs=5 application [args]...
```

2.3 clCreateContext() - the "anonymous" method

If an application wants to create a context of exactly N GPUs (or CPUs), but does not care which GPUs it gets, then it can simply list all GPUs, using "`clGetDeviceIDs(platform,CL_DEVICE_TYPE_GPU,...)`", then use "`clCreateContext(props, N, array_of_N_devices,...)`" and VCL will take care of the rest. The application doesn't even need to check the availability of the device (`CL_DEVICE_AVAILABLE`), because VCL will automatically choose N of the available devices (if there are)!

So long as the application does not use "`clGetDeviceInfo()`" BEFORE creating the context, there are no further complications (note that there is no problem with using `clGetDeviceInfo` AFTER the context is created).

2.4 clCreateContext() - the "specific" method

If an application queries devices using "`clGetDeviceInfo()`" before calling "`clCreateContext()`", then the following problem could arise:

Suppose the cluster has two or more GPU types - they could even be almost identical, but one is revision C12.86 and the other is revision C12.87 (perhaps purchased a few months later) and they slightly differ in one or more of the parameters obtainable through "`clGetDeviceInfo()`".

Once the application queried about some property of a device (such as its name, its global or local memory size, its driver-version, its 3D dimensions, etc.), that device is pinned down to a specific device type (e.g. revision C12.86) - now according to the OpenCL specifications, if later the application requests to use that device for creating a context and no devices of revision C12.86 are available, it is no longer legal for VCL to replace this device with a C12.87 device!

Now suppose other tasks (including other MPI tasks of the same job!) have all done the same, queried device #1, found that it is an C12.86 and asked VCL to create a context using

it: the user may believe that there are sufficient GPUs in the cluster to accommodate all her tasks, but is not aware that the GPUs are divided between C12.86 and C12.87 and the numbers of each alone are insufficient. As a result, in some of the tasks "`clCreateContext()`" will fail with "`CL_DEVICE_NOT_AVAILABLE`".

Possible solutions:

1. If you expect to have no competing tasks over the same devices, then there is no problem.

2. If you can use another method - "clCreateContextFromType()" or not calling "clGetDeviceInfo()" before "clCreateContext()", do so.

3. Be prepared for failures (CL_DEVICE_NOT_AVAILABLE), then if a failure occurs, try other devices.

   You may also use "clGetDeviceInfo(device_id, CL_DEVICE_AVAILABLE,...)" to minimize the number of failures, but note that races are possible where other tasks can grab the device between that and "clCreateContext()".

4. Coordinate with your other tasks exactly which type(s) of devices each is going to use.

5. Sometimes it is possible to use the "vclrun –ban=" option (see below) to force specific device-types for specific tasks.
   A combination of:

   `vclrun --ban=~gC12.86` (use only GPUs of version C12.86);
   and
   `vclrun --ban=~gC12.87` (use only GPUs of version C12.87)
   may do the trick.

6. If you know in advance how many devices are needed per task, then pre-allocate devices per task before your application starts running. This can be done by setting the environment variable `‘‘VCL_PRESELECT"` as follows:

   ```
   VCL_PRESELECT={c}:{g}:{a}
   ```

   Where c is the number of CPU devices, g is the number of GPU devices and a is the number of accelerator devices.

   The same is achieved by using:

   ```
   vclrun --preselect={c}:{g}:{a}
   ```

   For example, if your task should use 2 GPUs, run it with:

   ```
   vclrun --preselect=0:2:0 application [args]...
   ```

   This solves the problem because each task will only see its allocated devices - even "textttclGetDeviceIDs()" will only return the selected devices.

2.5 Presence of unwanted devices

In a cluster, which may have been set up for other users as well, there is a greater likelihood of finding devices that cannot or should not be used by the application. The application itself may not be programmed to deal with that very well, so if you want to avoid seeing certain devices, you can ban them using:

```
vclrun --ban=what-to-ban
```

For example:

`vclrun --ban=~xGTX` Ban all devices that are not of Nvidia's GTX series.

`vclrun --ban=cIntel` Ban Intel CPU devices.

`vclrun '--ban=<2.4GB'` Ban all devices with less than 2.4 Gigabytes of global memory.

`vclrun --ban=xAMD` Ban all AMD devices (where "AMD" appears either in the device-name or its vendor-name).

For a full description of options, read "man vcl".

2.6 Other VCL issues

1. Please stay away from writing kernels with conditional code that produces, depending on the device-type, either different kernel-names; a different number of kernel-arguments; or different types of kernel-arguments. If you do, then OpenCL programs that are built on multiple device-types, will not work, so if you need to use the same kernel-code with different device types, you need to create and build the appropriate kernels in two or more separate OpenCL programs.

2. VCL is unable to detect argument-size errors in "clSetKernelArg()": when an application supplies a kernel with an argument of a wrong size, this error will only be detected when attempting to run the kernel, causing "clEnqueueNDRangeKernel()" to fail with CL_INVALID_ARG_SIZE.

3. The OpenCL standard does not specify whether kernels need to be explicitly "submitted" after being queued. In VCL, all queued kernels are automatically and immediately submitted - so profiling shows them as spending 0 time in the "CL_QUEUED" state and immediately enter the "CL_SUBMITTED" state. The "clFlush()" function is thus made a redundant null operation.

Chapter 3

VCL with SLURM and MPI

The chapter explains how to use VCL with the Simple Linux Utility for Resource Management (SLURM) cluster management and job scheduling system[1], especially when MPI is concerned, or any other multi-tasking environment.

VCL support for SLURM:

- Provide a per-job private ad-hoc VCL cluster, based on SLURM's allocation rather than having a fixed cluster. This includes the necessary SLURM prologs and epilogs to establish and destroy this private cluster.
- Inform Slurm when VCL detects insufficient OpenCL devices.
- Includes instructions for SLURM administrators and users on how to incorporate VCL into SLURM.

3.1 Administrator installation guide - Using VCL with SLURM

1. Create a directory that is accessible (through the use of some networked file-system) to the whole cluster as well as your Slurm front end and writeable only to "root" and SlurmUser. Any pathname will do, but the rest of this page assumes that you selected "usr/local/slurm/scripts".

   Copy the following files from here to that directory:

   `gvprolog, vprolog, tvprolog, vepilog, vcldown`

   Thus, as SlurmUser (usually "root"):

   ```
   mkdir /usr/local/slurm/scripts
   cp gvprolog vprolog tvprolog vepilog vcldown /usr/local/slurm/scripts
   chmod 755 /usr/local/slurm/scripts/*
   ```

2. Create another empty directory that is accessible (through the use of some networked file-system) to the whole cluster as well as your Slurm front-end and writeable only to "root" and SlurmUser. This page assume that you selected "usr/local/slurm/vprologs"

[1] https://computing.llnl.gov/linux/slurm/slurm.html

3. Copy the prototype SLURM-VCL interface configuration:

   As SlurmUser:
   `cp slurm_vcl.conf.proto /usr/local/slurm/scripts/slurm_vcl.conf`

   then edit that file ("usr/local/slurm/scripts/slurm_vcl.conf"), setting:

   SLURMBIN=
   The full path where the SLURM binaries live, depending on your SLURM installtion (often SLURMBIN=/usr/local/slurm/bin).

   COMMONDIR=
   The full path to the second, empty directory created in the previous step (so COMMONDIR=/usr/local/slurm/vprologs).

   FEATURE=
   Select a SLURM feature (e.g. FEATURE=vcl) to describe the nodes in your cluster where VCL is to be installed and used as a back-end (this means the nodes with GPUs or other OpenCL devices).

   BROKER=
   Full pathname to the VCL "broker" program. (usually BROKER=/sbin/broker).

   BACKEND_TIMEOUT=
   Time in seconds to wait, when a job starts, for VCL to make contact with all back-end nodes. Since the prolog-time is limited by BatchStartTimeout and MessageTimeout (both default to 10 seconds), the recommended value is min(BatchStartTimeout,MessageTimeout) minus one
   (e.g. BACKEND_TIMEOUT=9).

   If there are many compute-nodes in the cluster and the network between some of them is slow or unreliable, then you may consider increasing BatchStartTimeout and MessageTimeout in "slurm.conf".

4. Configure "slurm.conf" to be aware of VCL:

   PrologSlurmctld=/usr/local/slurm/scripts/gvprolog
   Prolog=/usr/local/slurm/scripts/vprolog
   Epliog=/usr/local/slurm/scripts/vepilog
   TaskProlog=/usr/local/slurm/scripts/tvprolog

   Then add to all lines that describe a VCL back-end node or a range of VCL back-end nodes the following:

   Feature=vcl

   (or whatever other name you selected for "FEATURE" in step #3).

5. Run "scontrol reconfigure".

6. Install VCL, version 1.19 or higher, on all the nodes in your cluster. When configuring VCL on your cluster, configure those nodes with OpenCL devices that you want to use under Slurm as back-end nodes, but DO NOT configure any hosting-nodes.

7. If your SLURM users expect some or all of the VCL nodes to have a minimum number of operational OpenCL devices, then run "vclconf" on the appropriate nodes, then select 'c' followed by:

   - The minimum expected number of CPU devices (CL_DEVICE_TYPE_CPU).
   - The minimum expected number of GPU devices (CL_DEVICE_TYPE_GPU).
   - The minimum expected number of accelerator devices (CL_DEVICE_TYPE_ACCELERATOR).
   - "/usr/local/slurm/scripts/vcldown".

   If you use this option and a VCL back-end fails or finds less OpenCL devices than expected, then its "vcl" feature will be removed. Once you fix the problem and restart VCL on that node (or reboot it), you can (as SlurmUser) restore its "vcl" feature, using "scontrol reconfig" or "scontrol update Nodename=xx Features=vcl".

8. Instruct users, when requiring NN VCL back-end nodes for a job to run:

   ```
   srun -Cvcl\*{NN} ...  vclrun [vclrun-parameters] {program} [args]...
   ```

   Notes:

   (a) The same applies for "sbatch" and "salloc".

   (b) If the name for "FEATURE=" chosen in step #3 is other than "vcl", then "-Cvcl*{NN}" must be changed accordingly to "-C{feature-name}*{NN}".

   (c) Instead, for the convenience of your users, you may prefer to write a small wrapper script which does the same.

3.2 VCL support for MPI

VCL support for MPI or any other multi-tasking environment includes:

- A pre-allocation option, to prevent improper competition for devices between ranks.
- An option to ban unwanted devices, making them invisible to the application.

An MPI example

Assumptions:

1. Some nodes in the cluster have 2 GPUs.
2. You want to run a 10-task MPI job, each task with 3 GPUs, i.e., you need 10*3/2=15 VCL back-end nodes.
3. Most nodes with GPUs are low-end computers, mainly just to hold the GPUs, while the application run on stronger, high-end CPUs.
4. The OpenCL application relies on "clCreateContextFromType()" rather than select its devices explicitly.

5. While the cluster has a mix of different OpenCL devices, your specific job must run only on Nvidia's GTX series GPUs.

One way to do this is:

```
salloc --mem-per-cpu=8192 -C '[vcl*15&highend*10]' -N15-25
    [other-options]...

{mpi-flavour} {mpi-parameters-requesting-10-tasks}
    vclrun --policy=m --maxdevs=3 --defdev=g --preselect=0:3:0 --ban=~xGTX
           --job=slurm {application} [application-arguments]...
```

Explanation of "vclrun" arguments:

`--policy=m` "clCreateContextFromType()" will create a context from multiple nodes.

`--maxdevs=3` "clCreateContextFromType()" will pick no more than 3 available devices per context.

`--defdev=g` CL_DEVICE_TYPE_DEFAULT is GPU.

`--preselect=0:3:0` Allocate 3 private GPUs for the task (recommended on clusters with mixed GPUs).

`--ban=~xGTX` Use only Nvidia's GTX series.

`--job=slurm` VCL will pick its devices from within its SLURM allocation.

Chapter 4

SuperCL

4.1 Overview

SuperCL is a micro-language to optimize remote OpenCL operations by reducing the network overheads.

4.2 Using SuperCL

When running OpenCL on remote devices, network latency is the main limiting factor. For example, in order to run a remote kernel or to perform a remote I/O operation, an instruction must be sent over the network, followed by a reply. This adds two network-latency delays to the kernel's run time.

SuperCL is an extension of OpenCL, which under VCL allows micro-programs of OpenCL operations to run efficiently on devices of remote computers (nodes). SuperCL minimizes the network delays by packing multiple remote kernel activations and/or I/O operations into a single call, so that only one networked instruction is needed for all kernel activations and only one reply is received.

Chapter 5

Installation

5.1 Automatic installation

To install VCL, run "**./vcl.install**".

VCL may also be installed manually.

5.2 Manual installation

As root, run: "**mkdir /usr/lib/vcl /etc/vcl**", then place the following files in the corresponding directories:

File	Directory
vcl	/etc/init.d/vcl
vclconf	/sbin/vclconf
opencld	/sbin/opencld
broker	/sbin/broker
libopenCL.so	/usr/lib/vcl/libOpenCL.so
vclrun	/usr/bin/vclrun
man/man7/vcl.7	/usr/share/man/man3/supercl.3
man/man3/supercl.3	/usr/share/man/man7/vcl.7
supercl.h	/usr/include/supercl.h

Then either run "**vclconf**" or edit the VCL configuration manually, according to the instructions in "**man vcl**".

Chapter 6

Manuals

The manuals in this chapter are provided for general information. Users are advised to rely on the manuals that are provided with their specific VCL distribution.

6.1 For users

VCL - VirtualCL

6.2 For programmers

SuperCL - Optimize remote OpenCL operations

NAME

VirtualCL (VCL) — - a wrapper for using cluster-wide OpenCL devices

INTRODUCTION

The OpenCL standard allows applications to accelerate computation by using various GPU and other devices in a generic way. However, the number of such accelerating devices is limited by hardware, with typically only 1-4 devices per computer.

VirtualCL (VCL) is a wrapper for OpenCL that extends access to OpenCL devices (such as CPUs, GPUs and accelerators) beyond the devices of the local computer.

Users of VCL run their applications on hosting-nodes (hosts), using the VCL library instead of a vendor-specific SDK (OpenCL Software-Development-Kit). The actual OpenCL devices reside in back-end nodes. Hosting-nodes and back-end nodes may overlap, so some computers may serve simultaneously as both a hosting-node and a back-end node.

REQUIREMENTS

1. All participating computers must run Linux with the x86_64 (64-bit) architecture.
2. Hosts must be connected to back-end nodes over a network that supports TCP/IP.
3. TCP/IP port 255 must be reserved for VCL (not used by other applications or blocked by a firewall).
4. Back-end nodes must have OpenCL version 1.1 (or higher) installed (but different nodes are not required to have the same hardware or SDK).

CONFIGURATION

To configure VCL interactively, simply run vclconf, which will guide you through the various configuration options.

Vclconf can be used in two ways:

1. In order to configure the local computer, respond to the first question by pressing <Enter>.
2. In order to configure other computers, respond to the first question by entering the path to a root-directory: this is typically done on an NFS server that stores an image of the root-partition for a cluster of hosts, back-end nodes, or both.

Below is a detailed description of the VCL configuration files, in case you prefer to edit them manually:

`/etc/vcl/is_back_end`
: The presence of this file indicates that the computer is a back-end node.

`/etc/vcl/may_read_files`
: The presence of this file allows reading files on this back-end node for the implementaiton of the CL_MEM_FILE_HOST_PTR extension (see below) and within SuperCL(3).

`/etc/vcl/may_write_files`
: The presence of this file allows writing files on this back-end node within SuperCL(3).

`/etc/vcl/amd-1.2`
: The presence of this file indicates that you wish to use AMD's experimental OpenCL-1.2 SDK on this back-end (see "CURRENT STATUS" below).

`/etc/vcl/is_host`
: The presence of this file indicates that the computer is a hosting-node.

`/etc/vcl/nodes`
: On hosting nodes, this file explicitly lists the potential back-end nodes, one per line, either as host-names or as IP addresses (see "STARTING VCL" below for debug options).

`/etc/vcl/passwd`
: This file contains a unique password that is agreed between the relevant hosts and back-end nodes. It MUST be owned by "root" and allow no read/write permissions to any other users. When present, this will prevent unauthorized hosting-nodes from attempting to contact `VCL`.

`/etc/vcl/dev_min`
: This file, if present on back-end nodes, contains one line comprised of 3 space-separated integers followed by a shell command. The integers (at least one of which must be non-zero), enumerate the minimum number of OpenCL devices of each type to expect, respectively CPUs; GPUs; and accelerators. If a fatal error occurs at the back-end, or once a minute if less OpenCL devices are detected, then the given shell command is run (as "root").

RUNNING ON HOSTING NODES

To run an application with VCL, either use the wrapper script,

```
vclrun [options] {program} [args]...
```

or make sure that the `VCL` library: `/usr/lib/vcl/libOpenCL.so` takes precedence over any vendor-specific OpenCL library. For example, by setting the following environment variable: `LD_LIBRARY_PATH=/usr/lib/vcl`, or `LD_PRELOAD=/usr/lib/vcl/libOpenCL.so`.

Certain aspects of VCL can be manipulated by environment variables. These variables can also be set using `vclrun` options:

CONTEXT_POLICY=
: Decides how devices are allocated to a context by `clCreateContextFromType()`:

  CONTEXT_POLICY=1 (or CONTEXT_POLICY=o)
  : selects just a single device.

  CONTEXT_POLICY=2 (or CONTEXT_POLICY=n)
  : selects as many devices as possible, all from the same back-end node.

  CONTEXT_POLICY=3 (or CONTEXT_POLICY=m)
  : selects as many available devices from all available back-end nodes.

  When unspecified, the default is "2".

  Alternately, use:

  ```
  vclrun --policy={1|2|3|o|n|m}.
  ```

MAX_DEVS_PER_CONTEXT=
: selects the maximum (integer) number of devices that can be allocated by `clCreateContextFromType()`. When unspecified, the default is 1024.

  Alternately, use:

  ```
  vclrun --maxdevs={n}.
  ```

DEFAULT_DEVICE_TYPE=
: assigns the default device types associated with OpenCL's `CL_DEVICE_TYPE_DEFAULT`, as used in `clGetDeviceIds()` and `clCreateContextFromType()`. `DEFAULT_DEVICE_TYPE=c` selects `CL_DEVICE_TYPE_CPU`, `DEFAULT_DEVICE_TYPE=g` selects `CL_DEVICE_TYPE_GPU` and `DEFAULT_DEVICE_TYPE=a` selects `CL_DEVICE_TYPE_ACCELERATOR`. Two-letter combination allow selecting a combination of 2 out of the above 3 device types. When unspecified, the default is all three.

Alternately, use:
`vclrun --defdev={c|g|a|cg|ca|ga}.`

OPENCL_EXTENSIONS=
When `clGetPlatformInfo()` is called with the `CL_PLATFORM_EXTENSIONS` parameter, it cannot report correctly which extensions are available because different back-end nodes may support different extensions. If an application requires certain extensions (and the user is confident that these extensions are indeed supported by all back-end nodes), then a comma-separated list of extensions can be given, which can then be returned by `clGetPlatformInfo()`.

`VCL` also adds its own extension, named "multi-node", which by default is the only extension returned.

(note that extensions that allow `OpenCL` to interact with `OpenGL` are not supported).

Alternately, use:
`vclrun --extensions={comma-separated-list-of-extensions}.`

SIG_FOR_VCL_USE=
The VCL library needs a signal for its internal use. The variable `SIG_FOR_VCL_USE={signum}` can be used to select a different signal-number in the range of 34 to 64, in case the default of 45 is already in use by the application.

Alternately use:
`vclrun --sig={signum}`

VCL_JOBID=
Declare the application to be part of a "job", which can be any positive number: to succeed, the system-administrator must first allocate separate resources for the given job (see `JOB SUPPORT` below).

Alternately, use:
`vclrun --job={job-number}`
or to obtain the job ID from a SLURM environment, use:
`vclrun --job=slurm.`

VCL_PRESELECT=
Cause VCL to allocate a fixed private set of devices for the application as it starts (when first calling "clGetDeviceIDs()" or "clCreateContextFromType()") and only use those rather than all the devices in the cluster. The value is three colon-separated integers, implying the number of CPUs, GPUs and accelerators respectively. If less devices are available, VCL will still attempt to work with the available devices.

Alternately use:
`vclrun --preselect={ncpus:ngpus:naccelerators}`

VCL_BANNED={what-to-ban}
Ban certain devices so the application will not see or use them in `clGetDeviceIDs()` or `clCreateContextFromType()`.

`{what-to-ban}` can be one of the following:

{c|g|a|x}[{string}]
The first letter selects the device type:
c CPU
g GPU

a Accelerator
x All device types

The optional case-insensitive string is a any sub-string that may appear in either a device's name or its vendor's name. If the string is missing, then all devices of the given type are banned.

<{value}[KB|MB|GB|TB]
Any devices with less global memory than the given real value (in bytes, kilobytes, megabytes, gigabytes or terabytes).

>{value}[KB|MB|GB|TB]
Any devices with more global memory than the given real value (in bytes, kilobytes, megabytes, gigabytes or terabytes).

Banned items can be preceded by the character '~' to reverse the condition. Several bans can be combined by placing the character ':' between them.

`Examples:`
VCL_BANNED=cIntel:<4.5GB
Ban all Intel-CPU devices and all devices with less than 4.5GB global memory.
VCL_BANNED=~gGTX:~cAMD:a
Ban all GPUs that are not of Nvidia's GTX series, all CPUs that are not AMD's and all accelerators.

Alternately use:
`vclrun --ban={what-to-ban}`
(multiple selections of `--ban=` are allowed)

STARTING VCL

To start `VCL`, run:
`/etc/init.d/vcl start`.

If you just configured a back-end node to also be a hosting-node, run:
`/etc/init.d vcl start_host`.

If you just configured a hosting node to also be a back-end node, run:
`/etc/init.d/vcl start_backend`.

VCL EXTENSIONS

A new memory-object creation flag was introduced in `VCL` to prevent the expensive overhead of sending kernel input over the network. The initial contents of an input memory-object may instead be read from a file on the first back-end node where the memory-object is used by a kernel.

When the
`CL_MEM_FILE_HOST_PTR` (0x100000)
flag is set, the `host_ptr` argument of `clCreateBuffer()`, `clCreateImage2D()` and `clCreateImage3D()` points to a structure that describes a file from which the memory-object is to be read. The structure contains:

```
{
        long long version;                      /* must be 1 for now */
        char *filename;                 /* the file from which to read */
        unsigned long long file_offset; /* where to start reading */
        unsigned long long count;       /* number of bytes to read */
}
```

If the filename does not start with a '/', then it is interpreted relative to the current-diretory on the hosting-node.

If count is greater than the object's size, then it is truncated to the object's size. If it is smaller than the object's size, then the rest of the memory-object is filled with 0's.

Unless filename is "/dev/null", a file named /etc/vcl/may_read_files must be present on the back-end node to permit access. filename is opened on the back-end host using the same user and group IDs as that of the calling application. If the file cannot be read on the back-end node (including due to lack of permission or the absence of the file /etc/vcl/may_read_files), then the kernel fails with the error CL_OUT_OF_RESOURCES.

The contents of a memory-object created with the CL_MEM_FILE_HOST_PTR flag are undefined until the first kernel uses that memory-object. Attempts to read/write/copy such a memory-object before its first use will fail with the error CL_INVALID_MEM_OBJECT.

The CL_MEM_FILE_HOST_PTR flag cannot be combined with CL_MEM_USE_HOST_PTR or CL_MEM_COPY_HOST_PTR.

Another major extension is SUPERCL, combining multiple back-end operations in a single call, thus saving on network delays - please read the SUPERCL(3) manual page.

JOB SUPPORT

A "job" is a set of application-instances that share access to a number of OpenCL devices on different nodes (not necessary the cluster defined by vclconf). Jobs are numbered with positive integers. To set up for running a job, the system-administrator (or a "root" script) should run:

```
/sbin/broker -j{jobID} [-u user] -w[seconds]
       -l{host1[:c1:g1:a1][,{host2}[:c2:g2:a2]]...
```

The option -u user argument limits the job to a specific user.

The -w argument ensures that the applications within this job cannot start before all the nodes with OpenCL devices respond - otherwise applications that start shortly after may detect less or even no OpenCL devices. Broker normally runs as a daemon in the background, but with the -w argument, it will remain in the foreground for up to the given number of seconds (default is 10) or until all the specificed nodes respond, whichever comes first: if not all nodes respond within the given time, then broker terminates with exit-code 15.

Each node within the comma-separated list of nodes that follows the -l argument may be followed by a specification of the maximum number of devices to use from that node. For example, -llocalhost:1:2:3 means to use up to one CPU device from the local node, up to 2 GPU devices and up to 3 accelerator devices.

When a job is ended, the system-administrator (or a "root" script) should stop its broker using:

```
/sbin/broker -k{jobID}.
```

CURRENT STATUS

VCL supports OpenCL version 1.1 (and 1.0) almost completely.

Outstanding problems:

1. There is no way (yet) for applications to know on which back-end node a given device resides.
2. OpenCL-programs that produce different routines, or routines with different parameters for different device-types, will only work properly if they are created as different "Programs" for different device-types.

3. VCL is unable to return an error when an application supplies a kernel with an argument of a wrong size. An error will therefore only occur when the kernel is eventually activated.

4. Unless your platform supports an OpenCL extension such as `cl_amd_event_callback`, then at times, applications that call `clGetEventInfo()` may not be immediately aware when a kernel transits from the `CL_SUBMITTED` state into the `CL_RUNNING` state. In some cases, a delay of up to one minute can be expected.

5. There is a bug in the experimental AMD SDK for OpenCL-1.2. If your platform version is "OpenCL 1.2 AMD-APP (938.1)" or "OpenCL 1.2 AMD-APP (938.2)", then you have this faulty SDK.

   VCL can overcome this bug, but it cannot simultaneously support on the same back-end both this faulty platform along with other platforms which do not support the OpenCL-1.2 standard (or higher). If you wish to use this faulty SDK, then the system-administrator must configure (using `vclconf`) whether to use it or your other SDK(s).

SEE ALSO

`supercl(3).`

NAME

SuperCL - Optimize remote OpenCL operations by reducing network overheads

PURPOSE

Network latency is expensive when using OpenCL devices on remote computers (nodes), as in the case of VCL(7). In order to run a remote kernel or perform other remote I/O operations, an instruction must be sent over the network, followed by a reply and this adds two network-latency delays to the kernel's execution time. SuperCL minimizes these delays by packing multiple remote kernel activations and/or I/O operations into a single call, so that only one networked instruction is needed to activate them all and only one reply is received.

AUDIENCE

This manual is intended for programmers who are familiar with the OpenCL library.

SYNOPSIS

```
#include <supercl.h>

cl_int clSuper(cl_command_queue queue,
                          struct super_sequence *sequence,
                          cl_uint num_events_in_wait_list,
                          const cl_event_wait_list *event_wait_list,
                          cl_event *event);
```

DESCRIPTION

clSuper() is functionally similar to standard command-enqueueing OpenCL functions, but instead of queuing a single operation, it queues a sequence of operations. clSuper shares its OpenCL context and other OpenCL objects with the rest of the OpenCL environment and can be freely mixed with other OpenCL functions.

num_events_in_wait_list, event_wait_list and event are used in the same manner as the last three arguments of all other OpenCL command-enqueueing functions (check the OpenCL documentation).

Sequence is an array of instructions, constituting a mini-program that invokes multiple OpenCL kernels and I/O operations on memory objects. Each instruction is 128 bytes long, beginning with the cmd (command) field, where the rest of the instruction depends on the specific command. The sequence must end with the command SCL_END_OF_SEQUENCE. clSuper executes the instructions of the sequence in order, unless an instruction calls for a jump, forks a new thread or terminates a thread. When the last thread reaches the SCL_END_OF_SEQUENCE without encountering any errors, the operation terminates and the event status is set to CL_COMPLETE. If an error is encountered, all threads terminate and the event status is set to the error-code.

SuperCL programs can use registers for program control (such as loops) and for certain parameterised operations as described below. Registers can contain either 64-bit integers or 64-bit real numbers. Register numbers are positive 31-bit integers, registers need not be declared in advance and are initialised to integer-zero on first use. While there is no hard limit on the number of registers, using a very large number of them (1000's and more) can significantly slow down SuperCL.

There are two types of registers - global and private: global registers are shared among all the program-threads, while private registers belong to specific thread-groups. Note that if a program does not use threads (see the SCL_FORK instruction below), then the global and private registers are one and the same.

All the memory-objects that are mentioned in the sequence are migrated to the queue's target node before the sequence commences and remain on that node at least for the duration of the clSuper()

instance. It is therefore best to not attempt using the same memory-objects in different SuperCL instances (or as arguments of unrelated OpenCL kernels) that run on different nodes, as that would result in serialisation of the operations and a serious loss of performance.

Note that the addition of the `CL_MEM_FILE_HOST_PTR` flag in `cl_mem_flags` (See `vcl(7)`) allows for the use of temporary memory-objects that can be used only within a `clSuper()` instance and whose data never needs to be transferred across the network from the host-application to the remote node.

The instructions are:

SCL_KERNEL_ARG
Set a kernel argument to a fixed value.

The parameters for this instruction are:

```
struct
{
        cl_kernel kernel;
        cl_int argno;
        long arg_len;
        union
        {
                int arg_int;
                long arg_long;
                float arg_float;
                double arg_double;
                char arg_value[SCL_MAX_ARG_LEN];
                cl_mem arg_mem;
                cl_sampler arg_sampler;
                struct
                {
                        int regno;
                        int reg_is_private;
                };
        };
} kernel_arg;
```

`kernel_arg.kernel` is the kernel for which to set the argument.

`kernel_arg.argno` is the index of the argument to set (starting from 0).

`kernel_arg.arglen` is the size of the argument.

Memory-object arguments are placed in `kernel_arg.arg_mem`. Sampler arguments are placed in `kernel_arg.arg_sampler`. Regular arguments are placed according to their type in either `kernel_arg.arg_int`, `kernel_arg.arg_long`, `kernel_arg.arg_float`, `kernel_arg.arg_double` or `kernel_arg.arg_value`. The size of local arguments is placed in `kernel_arg.arg_len`, with `kernel_arg.arg_long` of 0.

In the rare case when an argument's size is more than `SCL_MAX_ARG_LEN bytes`, a series of consecutive `SCL_KERNEL_ARG` instructions can be used, with the argument split between their `kernel_arg.arg_value`. If so, `kernel_arg.arg_len` must be set to the total argument size and `kernel_arg.kernel`, `kernel_arg.argno` and `kernel_arg.arg_len` must be the same throughout the series.

Note that all kernel arguments must be defined within SuperCL sequences before a kernel can run: arguments previously set outside the sequence (using `clSetKernelArg()`) `do not hold within clSuper()`.

SCL_KERNEL_ARG_FROM_REGISTER
Set a variable kernel argument from a given register.

The parameters for this instruction are as in `SCL_KERNEL_ARG` above, except that the data is taken from the register `kernel_arg.regno`. If `kernel_arg.reg_is_private` is set, then `kernel_arg.regno` designates a private register.

The argument's size must be 1, 2, 4 to 8 bytes and if the register contains a real value, then the argument can only be 4 bytes (float) or 8 bytes (double). It is the programmer's responsibility to make sure that the argument's type corresponds to the register's type (integer or real).

SCL_RUN_KERNEL
Run an OpenCL kernel.

The parameters for this instruction are:

```
struct
{
        cl_kernel kernel;
        cl_device_id device;
        int work_dim;
        size_t global_work_offset[3];
        size_t global_work_size[3];
        size_t local_work_size[3];
} run_kernel;
```

`run_kernel.kernel` is the kernel to run. `run_kernel.device` is the device to run the kernel on: the common value of NULL implies the same device as the device associated with the `queue`, but it can also be another device, so long as it is on the same node and of the same platform as the device of the `queue` (note that this deviates and expands on the normal functionality of OpenCL queues).

`run_kernel.work_dim`, `run_kernel.global_work_offset`, `run_kernel.global_work_size` and `run_kernel.local_work_size` correspond to the same arguments of `clEnqueueNDRangeKernel()`.

When any element of `run_kernel.global_work_offset` is negative, or when any element of `run_kernel.global_work_size` and/or `run_kernel.local_work_size` is less than 1, this means that the actual offset/size is in a private register whose number is minus the corresponding value.

SCL_COPY_BUFFER
Copy an OpenCL buffer, or a part thereof, to another OpenCL buffer.

The parameters for this instruction are:

```
struct
{
        cl_mem from;
        cl_mem to;
        off_t from_offset;
        off_t to_offset;
        size_t count;
```

```
        unsigned int flags;
} copy_buffer;
```

When no flags are set, `copy_buffer.count` bytes are copied from buffer `copy_buffer.from` at offset `copy_buffer.from_offset` to buffer `copy_buffer.to` at offset `copy_buffer.to_offset`.

When `copy_buffer.flags` include the flags: `SCLF_FROM_OFFSET_IS_A_REGISTER_NUMBER`, `SCLF_TO_OFFSET_IS_A_REGISTER_NUMBER` and/or `SCLF_COUNT_IS_A_REGISTER_NUMBER`, then the corresponding values in `copy_buffer.from_offset`, `copy_buffer.to_offset` and/or `copy_buffer.count` are taken to be register numbers containing the corresponding integer values. Further, if `copy_buffer.flags` also include the flags: `SCLF_FROM_OFFSET_REGISTER_IS_PRIVATE`, `SCLF_TO_OFFSET_REGISTER_IS_PRIVATE` and/or `SCLF_COUNT_REGISTER_IS_PRIVATE`, then the corresponding registers are taken to be private registers.

SCL_COPY_IMAGE
Copy an OpenCL image, or a region thereof, to another OpenCL image.

The parameters for this instruction are:

```
struct
{
        cl_mem from;
        cl_mem to;
        size_t src_origin[3];
        size_t dst_origin[3];
        size_t region[3];
} copy_image;
```

The given region (`copy_image.region`) is copied from the image `copy_image.from` at `copy_image.src_origin` to the image `copy_image.to` at `copy_image.dst_origin`.

SCL_COPY_IMAGE_TO_BUFFER
Copy an OpenCL image, or a region thereof, to an OpenCL buffer.

The parameters for this instruction are:

```
struct
{
        cl_mem image;
        cl_mem buffer;
        size_t image_origin[3];
        size_t region[3];
        off_t buffer_offset;
        unsigned int flags;
} copy_image_to_buffer;
```

When no flags are set, the region `copy_image_to_buffer.region` is copied from the image `copy_image_to_buffer.image` at `copy_image_to_buffer.image_origin` to the buffer `copy_image_to_buffer.buffer` at `copy_image_to_buffer.buffer_offset`.

When `copy_image_to_buffer.flags` includes the flag `SCLF_TO_OFFSET_IS_A_REGISTER_NUMBER`, then `copy_image_to_buffer.buffer_offset` is taken to contain the number of an integer-register that

contains the target buffer's offset. Further, if `copy_image_to_buffer.flags` also includes the flag `SCLF_TO_OFFSET_REGISTER_IS_PRIVATE`, then that register is taken to be a private register.

SCL_COPY_BUFFER_TO_IMAGE
Copy an OpenCL buffer, or a part thereof, to an OpenCL image.

The parameters for this instruction are:

```
struct
{
        cl_mem buffer;
        cl_mem image;
        off_t buffer_offset;
        size_t image_origin[3];
        size_t region[3];
        unsigned int flags;
} copy_buffer_to_image;
```

When no flags are set, a section of the buffer `copy_buffer_to_image.buffer`, beginning at `copy_buffer_to_image.buffer_offset`, is copied to the region `copy_buffer_to_image.region` of the image `copy_buffer_to_image.image` at `copy_buffer_to_image.image_offset`.

When `copy_buffer_to_image.flags` includes the flag `SCLF_FROM_OFFSET_IS_A_REGISTER_NUMBER`, then `copy_buffer_to_image.buffer_offset` is taken to contain the number of an integer-register that contains the source buffer's offset. Further, if `copy_buffer_to_image.flags` also includes the flag `SCLF_FROM_OFFSET_REGISTER_IS_PRIVATE`, then that register is taken to be a private register.

SCL_FILL_BUFFER
Fill a buffer (or part thereof) with a pattern.

The parameters for this instruction are:

```
struct
{
        cl_mem buffer;
        char pattern[16];
        int pattern_size;
        size_t offset;
        size_t size;
        unsigned int flags;
} fill_buffer;
```

A segment of the buffer `fill_buffer.buffer` of size `fill_buffer.size`, is filled with a pattern (`fill_buffer.pattern`) of size `fill_buffer.pattern_size` (up to 16 bytes), starting at offset `fill_buffer.offset`.

When `fill_buffer.flags` contains the bit `SCLF_OFFSET_IS_A_REGISTER_NUMBER`, then `fill_buffer.offset` points to a register that contains the actual offset and if the bit `SCLF_OFFSET_REGISTER_IS_PRIVATE` is also set, then that is a private register. Similarly, when `fill_buffer.flags` contains the bit `SCLF_COUNT_IS_A_REGISTER_NUMBER`, then `fill_buffer.size` points to a register that contains the actual size and when the bit `SCLF_COUNT_REGISTER_IS_PRIVATE` is also set, then that is a private register.

SCL_FILL_IMAGE
Fill an image (or part thereof) with a pattern.

The parameters for this instruction are:

```
{
struct
        cl_mem image;
        float fill_color[4];
        size_t origin[3];
        size_t region[3];
        unsigned int flags[3];
} fill_image;
```

A region of the image `fill_image.image` of dimensions `fill_image.region`, starting at `fill_image.origin`, is filled with the four-compenent RGBA colour `fill_image.fill_color`.

For each dimension (d), when the corresponding `fill_buffer.flags[d]` contains the bit `SCLF_OFFSET_IS_A_REGISTER_NUMBER`, then `fill_buffer.origin[d]` points to a register that contains the actual origin in that dimension and if the bit `SCLF_OFFSET_REGISTER_IS_PRIVATE` is also set, then that is a private register. Similarly, when `fill_buffer.flags[d]` contains the bit `SCLF_COUNT_IS_A_REGISTER_NUMBER`, then `fill_buffer.region[d]` points to a register that contains the actual extent in that dimension and when the bit `SCLF_COUNT_REGISTER_IS_PRIVATE` is also set, then that is a private register.

SCL_LOAD_REGISTER_FROM_BUFFER
Load a register from an element of an OpenCL buffer.

The parameters for this instruction are:

```
struct
{
        int regno;
        int flags;
        cl_mem buffer;
        size_t offset;
        enum
        {
                LOAD_CHAR, LOAD_UCHAR, LOAD_SHORT, LOAD_USHORT, LOAD_INT,
                LOAD_UINT, LOAD_LONG, LOAD_ULONG, LOAD_FLOAT, LOAD_DOUBLE
        } register_type;
} reg_buffer;
```

An element from the buffer `reg_buffer.buffer` at offset `reg_buffer.offset` is loaded to the register number `reg_buffer.regno`. The type of the element is determined by `reg_buffer.register_type` and can be: char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, float or double.

If `reg_buffer.flags` includes the flag `SCLF_REGISTER_IS_PRIVATE`, then the register to be loaded is a private register.
If `reg_buffer.flags` includes the flag `SCLF_REGISTER_IS_INDIRECT`, then `reg_buffer.regno` is taken to be a register that contains the (integer) number of the register into which to load the data.
If `reg_buffer.flags` also includes the flag `SCLF_INDIRECT_REGISTER_IS_PRIVATE` then the

register contaning the register-number is private.

If `reg_buffer.flags` includes the flag `SCLF_OFFSET_IS_A_REGISTER_NUMBER` then `reg_buffer.offset` is taken to be an (integer) register number of a register that contains the actual (integer) offset.
If `reg_buffer.flags` also include the flag `SCLF_OFFSET_REGISTER_IS_PRIVATE`, then that register is private.

SCL_LOAD_BUFFER_FROM_REGISTER
Store a register in an element of an OpenCL buffer.

The parameters for this instruction are exactly as in `LOAD_REGISTER_FROM_BUFFER`, except that the value of the register is copied to the buffer element.

SCL_COPY_BUFFER_TO_HOST
Copy an OpenCL buffer, or a part thereof, to the host (application) memory.

The parameters for this instruction are:

```
struct
{
        cl_mem buf;
        size_t offset;
        size_t count;
        void *to;
        size_t host_offset;
        unsigned int flags;
} copy_buffer_to_host;
```

When no flags are set, `copy_buffer_to_host.count` bytes of the buffer `copy_buffer_to_host.buf`, beginning at `copy_buffer_to_host.offset`, are copied to host (application) memory address `(copy_buffer_to_host.to + copy_buffer_to_host.offset)`

When `copy_buffer_to_host.flags` includes the flags `SCLF_FROM_OFFSET_IS_A_REGISTER_NUMBER`, `SCLF_TO_OFFSET_IS_A_REGISTER_NUMBER` and/or SCLF_COUNT_IS_A_REGISTER_NUMBER, then the respective buffer-offset, host-offset and/or count are taken to be register numbers where the corresponding registers contain the values of the buffer-offset, host-offset and/or the byte-count. Further, when `copy_buffer_to_host.flags` also contains the flags `SCLF_FROM_OFFSET_REGISTER_IS_PRIVATE`, `SCLF_TO_OFFSET_REGISTER_IS_PRIVATE` and/or `SCLF_COUNT_REGISTER_IS_PRIVATE`, then the corresponding registers are taken to be private registers.

This instruction may complete before the data actually arrives at the host-application. It is however guaranteed that data from all `SCL_COPY_BUFFER_TO_HOST` and `SCL_COPY_IMAGE_TO_HOST` instructions and signals from all `SCL_SIGNAL_HOST` instructions (even when issued by other threads) will arrive at the host-application in the same order as completed.

SCL_COPY_IMAGE_TO_HOST
Copy an OpenCL image, or a region thereof, to the host (application) memory.

The parameters for this instruction are:

```
struct
```

```
{
        cl_mem image;
        size_t origin[3];
        size_t region[3];
        void *to;
} copy_image_to_host;
```

The region `copy_image_to_host.region` of the image `copy_image_to_host.image`, beginning at `copy_image_to_host.origin`, is copied to the host (application) memory address `copy_image_to_host.to`

This instruction may complete before the data actually arrives at the host-application. It is however guaranteed that data from all `SCL_COPY_BUFFER_TO_HOST` and `SCL_COPY_IMAGE_TO_HOST` instructions and signals from all `SCL_SIGNAL_HOST` instructions (even when issued by other threads) will arrive at the host-application in the same order as completed.

SCL_SIGNAL_HOST
Send a signal to the host application.

Instruction parameter is:

```
int sig;   /* 1 <= sig <= 64 */
```

This instruction may complete before the signal actually arrives at the host-application. It is however guaranteed that data from all `SCL_COPY_BUFFER_TO_HOST` and `SCL_COPY_IMAGE_TO_HOST` instructions and signals from all `SCL_SIGNAL_HOST` instructions (even when issued by other threads) will arrive at the host-application in the same order as completed.

SCL_FILENAME
Set a file-name for later use by the `SCL_CHDIR, SCL_READFILE` and `SCL_WRITEFILE` instructions.

The parameters for this instruction are:

```
struct
{
        char cont;
        char numeric;
        char digits;
        char reg_is_private;
        int regno;
        char fn[SCL_MAX_FILENAME_LEN];
} filename;
```

Each thread carries one file-name (initially the NULL string). When forking, this file-name is inherited by the child-thread.

If `filename.cont` is 0, then a fresh file-name is started. If it is 1, then a string is appended to the existing file-name.

If `filename.numeric` is 0, then the string is taken from `filename.fn`. If it is 1, then a numeric string is built from the non-negative integer register, `filename.regno` (when `filename.reg_is_private` is set, then `filename.regno` refers to a private register). Further, if `digits` is positive (1-127), then the numeric string will contain at least that number of digits, 0-padded to the left as necessary.

SCL_CHDIR
Set a current-directory for later use by the `SCL_READFILE` and `SCL_WRITEFILE` instructions.

This instruction has no parameters.

Each thread may carry a current-directory (initially none). When forking, this current-directory is inherited by the child-thread.

The thread's current directory is set by this instruction to the current thread's file-name, which must therefore be previously set.

Unless the current thread's file-name starts with a '/', the directory-name is interpreted relative to the former current-directory, which must also be previously set.

Note that the current-directory refers to the directory itself, not to its name, so if that directory is later moved, file input/output that is relative to the current-directory will occur in the moved directory.

SCL_READFILE

SCL_WRITEFILE
Read data from a file to an OpenCL buffer or write data from an OpenCL buffer to a file.

The parameters for these instructions are:

```
struct
{
        cl_mem buffer;
        off_t file_offset;
        off_t buffer_offset;
        size_t count;
        unsigned int flags;
        int mode;
} file_rw;
```

The name of the file to read/write should be set in advance using the `SCL_FILENAME` instruction. Unless that file-name starts with a '/', it is interpreted relative to the thread's current-directory, which must be previously set as well.

`file_rw.buffer` is the buffer to read/write. `file_rw.file_offset` is the file-offset from where to read or where to write. `file_rw.buffer_offset` is the offset in the buffer where to read or write from. `file_rw.count` is the number of bytes to read/write. `file_rw.flags` may contain the following flags:

SCLF_FILE_OFFSET_IS_A_REGISTER_NUMBER
: `file_rw.file_offset` contains a register number which contains the actual file offset.

SCLF_FILE_OFFSET_REGISTER_IS_PRIVATE
: The register given with `SCLF_FILE_OFFSET_IS_A_REGISTER_NUMBER` is a private register.

SCLF_BUFFER_OFFSET_IS_A_REGISTER_NUMBER
: `file_rw.buffer_offset` contains a register number which contains the actual buffer offset.

SCLF_BUFFER_OFFSET_REGISTER_IS_PRIVATE
: The register given with `SCLF_BUFFER_OFFSET_IS_A_REGISTER_NUMBER` is a private register.

SCLF_COUNT_IS_A_REGISTER_NUMBER
: `file_rw.count` contains a register number which contains the actual count.

SCLF_COUNT_REGISTER_IS_PRIVATE
: The register given with `SCLF_COUNT_IS_A_REGISTER_NUMBER` is a private register.

SCLF_CREATE_FILE
(only with `SCL_WRITEFILE`) If the file to be written does not exist, create it with mode given in `file_rw.mode`.

SCL_ARITHMETIC
Perform various register operations or jumps.

The parameters for this instruction are:

```
struct
{
        union
        {
                long long i;
                double d;
        } val1, val2;
        enum supercl_arithmetic op;
        unsigned int flags;
        long label;
} arithmetic;
```

The operation to perform depends on `arithmetic.op`.

The first set of operations have two operands (`arithmetic.val1` and `arithmetic.val2`), where the first operand, a register, is affected by the second operand. These are:

SCLA_SET_VALUE	op1 = op2
SCLA_ADD	op1 += op2
SCLA_SUBTRACT	op1 -= op2
SCLA_MULTIPLY	op1 *= op2
SCLA_DIVIDE	op1 /= op2
SCLA_MODULUS	op1 %= op2
SCLA_POW	op1 ^= op2

If any of the operands is a is a real number, then the resulting value is a real number and if both operands are integers, then the resulting value is an integer. The exceptions are `SCLA_SET_VALUE`, where the resulting value is of the same type as the second operand and `SCLA_POW`, where the result is always a real number. The second operand of `SCLA_MODULUS` must be a positive integer.

The next operation is `SCLA_EXCHANGE`, where the contents of the first operand is exchanged with the contents of the second operand.

Next are single-operand operations:

SCLA_INT	op1 = round_down_to_integer(op1)
SCLA_LOG	op1 = ln(op1)
SCLA_EXP	op1 = exp(op1)
SCLA_SQRT	op1 = sqrt(op1)
SCLA_SIN	op1 = sin(op1)
SCLA_COS	op1 = cos(op1)
SCLA_ASIN	op1 = asin(op1)
SCLA_ACOS	op1 = acos(op1)

These result in real values, except for `SCLA_INT` that results in an integer value and `SCLA_SQRT` that retains the original type of `arithmetic.op1`.

Next are conditional jumps:

SCLA_JUMP_EQ	if(op1 == op2) goto arithmetic.label;
SCLA_JUMP_NE	if(op1 != op2) goto arithmetic.label;
SCLA_JUMP_LT	if(op1 < op2) goto arithmetic.label;
SCLA_JUMP_LE	if(op1 <= op2) goto arithmetic.label;
SCLA_JUMP_GT	if(op1 > op2) goto arithmetic.label;
SCLA_JUMP_GE	if(op1 >= op2) goto arithmetic.label;

By default, the label (`arithmetic.label`) is searched forward (wrapping back to the first instruction if the end-of-sequence is reached) - unless the flag `SCLF_JUMP_BACKWARD` is set in `arithmetic.flags`, causing a backward search.

Finally are conditional pauses - waiting until a condition is fulfilled by other threads:

SCLA_PAUSE_UNTIL_EQ	pause until(op1 == op2);
SCLA_PAUSE_UNTIL_NE	pause until(op1 != op2);
SCLA_PAUSE_UNTIL_LT	pause until(op1 < op2);
SCLA_PAUSE_UNTIL_LE	pause until(op1 <= op2);
SCLA_PAUSE_UNTIL_GT	pause until(op1 > op2);
SCLA_PAUSE_UNTIL_GE	pause until(op1 >= op2);

Operands are assumed by default to be integer global register-numbers unless the following flags are set in `arithmetic.flags`:

SCLF_OP1_INTEGER / SCLF_OP2_INTEGER
: The corresponding operand is a constant 64-bit integer (with value in `arithmetic.val1.i` or `arithmetic.val2.i`)

SCLF_OP1_REAL / SCLF_OP2_REAL
: The corresponding operand is a constant 64-bit real number (with value in `arithmetic.val1.d` or `arithmetic.val2.d`)

SCLF_OP1_SPECIAL_REGISTER / SCLF_OP2_SPECIAL_REGISTER
: The corresponding operand is a special internal register. At this time only two such registers are available, both are 64-bit integers and both are read-only:

1. `OPENCL_EVENT_COUNTER` increments every time an OpenCL operation completes on the device where the `clSuper()` is queued. This register cannot be relied upon to count the number of OpenCL operations on the device because it may also increment on other events - but as its value can only increase, it can be used to check whether any OpenCL function completed (including functions initiated by the current or another `SuperCL()` instance), for example in order to pause until the contents of a memory-object have changed.
2. `SUPERCL_NANOTIME` provides the time in nano-seconds since an arbitrary point in the past (which can be different for different devices).

SCLF_OP1_PRIVATE / SCLF_OP2_PRIVATE
: The corresponding register operand is private.

SCLF_OP1_INDIRECT / SCLF_OP2_INDIRECT
: The corresponding operand is a register containing the integer register-number on which to operate (this allows, for example, to create arrays of registers). The resulting register-number must be a non-negative integer.

SCLF_OP1_INDIRECT_PRIVATE / SCLF_OP2_INDIRECT_PRIVATE
: In combination with `SCLF_OP1_INDIRECT / SCLF_OP2_INDIRECT`, the corresponding register that contains the number of the register-operand is private.

SCL_LABEL
A place to jump to.

The parameter for this instruction is:

long label;

No operation is carried out: arithmetic jump operations can go here and new threads can start here.

SCL_FORK
Start a new thread.

The parameters for this instruction are:

```
struct
{
        long label;
        unsigned int flags;
} fork;
```

A new thread starts running from the label `fork.label`. By default, the label (`fork.label`) where the new thread starts is searched forward (wrapping back to the first instruction if the end-of-sequence is reached) - unless the flag `SCLF_JUMP_BACKWARD` is set in `fork.flags`, causing a backward search.

If `fork.flags` includes `SUPERCL_FORK_PRIVATE_REGISTERS`, then the new thread acquires its own set of private registers - otherwise the new thread shares its parent's private registers (if neither the parent thread nor any of its ancesstors was created using `SCL_FORK` with the `SUPERCL_FORK_PRIVATE_REGISTERS` set, then the "private" registers are the global registers).

SCL_JOIN
Join two threads.

This instruction has no parameters.

The first thread that reaches any specific `SCL_JOIN instruction waits there.` The second thread that arrives at that point exits and causes the first thread to continue.

If all remaining threads are waiting at an `SCL_JOIN` instruction, then `clSuper()` fails with the `CL_INVALID_PROGRAM_EXECUTABLE` error.

SCL_END_OF_SEQUENCE
End of the SuperCL sequence.

The parameters for this instruction are:

```
struct
{
        long version;
        int *faulty_instruction;
} end_of_sequence;
```

When all threads reach this instruction, the `clSuper()` instance is complete.

For the current release, `end_of_sequence.version` must be 0.

If `end_of_sequence.faulty_instruction` is not NULL and an error occurs, then the instruction number at which the error occured is stored in the integer pointed by `end_of_sequence.faulty_instruction` (at the time of calling `clSuper`). Instruction numbers start at 0. A few general errors that are not related to a specific instruction may instead store the total number of instructions. If no error occur, then the pointed integer is not modified.

In order to prevent races and allow threads to perform complex arithmetic operations in an atomic fashion, threads are guaranteed to continue to run uninterrupted by other threads so long as they:
1. Do not run OpenCL kernels.
2. Do not perform operations that involve OpenCL memory-objects.
3. Do not perform file I/O.
4. Do not jump backwards.
5. Do not fork backwards.
6. Do not arrive at `SCL_JOIN`.

C++

C++ programs can use SuperCL, but must prepend a "u." to all the instruction-parameters, for example: `u.kernel_args.kernel`.

ERROR CODES

Some errors are detected immediately when `clSuper()` is invoked, causing it to return an error. Among these errors:

CL_INVALID_VALUE
The `end_of_sequence.version` is not zero.

CL_INVALID_DEVICE
A device for running a kernel is either not on the same node as the node associated with `queue`; not of the same platform; or not in the same context.

CL_INVALID_KERNEL
A kernel mentioned in the `sequence` does not exist or does not belong to the same context.

CL_INVALID_PROGRAM
A kernel mentioned in the `sequence` is not built on any device of the node and platform associated with `queue`.

CL_INVALID_MEM_OBJECT
A memory-object mentioned in the `sequence` does not exist or does not belong to the same context.

CL_INVALID_VALUE
`sequence` is NULL.

CL_INVALID_VALUE
Kernel-argument length > SCL_MAX_ARG_LEN, but the following instruction(s) do not match the same `SCL_KERNEL_ARG` operation.

CL_INVALID_VALUE
Kernel-argument length is negative; zero for a regular argument; not sizeof(cl_mem) for a memory-object argument; or not sizeof(cl_sampler) for a sampler argument.

CL_INVALID_VALUE
Memory-object argument is not a memory-object

CL_INVALID_VALUE
Kernel-argument from register is not 1, 2, 4 or 8 bytes long.

CL_INVALID_VALUE
Inappropriate flags for an instruction.

CL_INVALID_VALUE
Negative or extremely large offset or region.

CL_INVALID_VALUE
Inappropriate signal number.

CL_INVALID_VALUE
Inappropriate register_type.

CL_INVALID_PROGRAM_EXECUTABLE
Jump/Fork to a non-existent label.

CL_INVALID_CONTEXT
Memory-object or sampler argument is not of the same context as `queue`.

CL_INVALID_ARG_INDEX
Invalid argument number.

CL_INVALID_SAMPLER
Sampler argument is not a sampler.

CL_INVALID_QUEUE
`queue` is not a valid command queue.

CL_INVALID_WORK_DIMENSION
Kernel's work-dimension is not 1-3.

CL_INVALID_WORK_GROUP_SIZE
Negative or extremely large work-group size.

CL_INVALID_MEM_OBJECT
A memory-object mentioned in a copy operation is of inappropriate dimenstions (a buffer when expecting an image or an image when expecting a buffer).

CL_INVALID_VALUE
Attempt to constract a file-name from a NULL-string; or using a negative number of digits.

Other errors are detected only once the `sequence` starts. In addition to the usual errors that are reported by OpenCL when functions fail, the following are also possible:

CL_INVALID_PROGRAM_EXECUTABLE
Lost connection with the remote node.

CL_INVALID_OPERATION
Arithmetic error: division by zero; square-root of a negative number; logarithmus of a non-positive number; modulus of a non-positive or non-integer value; raising a negative value to a non-integer power;

CL_INVALID_OPERATION
Negative or non-integer register number during an arithmetic operation.

CL_INVALID_VALUE
Attempt to load a 1 or 2 byte buffer element from a register that contains a real value.

CL_INVALID_VALUE
Register number derived from "offset" or "count" is negative or does not fit in a 31-bit integer.

CL_INVALID_VALUE
Indirect register in a copy operation contains a non-integer or a negative value.

CL_INVALID_VALUE
Bad; non-integer; or negative register used in constructing a file-name.

CL_INVALID_VALUE
File-name or former current-directory not defined when attempting to set current-directory.

CL_OUT_OF_RESOURCES
Failure to open the current-directory: possibly due to it's non-existence; file-name or former current-directory undefined; lack of access-permissions by user; or lack of administrative permission to perform I/O altogether on the node where `SuperCL` is running.

CL_INVALID_VALUE
Invalid; non-integer; or negative-valued register used as file-offset, buffer-offset or count in file-I/O operations.

CL_OUT_OF_RESOURCES
Failure to read/write file, possibly due to the file's non-existence; file-name or current-directory undefined; reading a file that is too short; lack of access-permissions by user; or lack of administrative permission to perform I/O altogether on the node where `SuperCL` is running.

CL_INVALID_VALUE
Count when copying data to host, is negative.

CL_INVALID_PROGRAM_EXECUTABLE:
Deadlock - all threads stuck in `SCL_JOIN`.

EXAMPLES OF USE

1. Run a number of kernels in a row.
2. Run a number of kernels N times in a loop.
3. Run a number of kernels N times in a loop - in between, assynchroneously report intermediate results to the application (possibly send it a signal to let it know that the data is ready).
4. Run an iterative kernel. The user of the interactive application may from time to time request to read a particular section of the data or to pause or terminate the SuperCL instance (this can be done by writing to a control buffer). If kernels run for a long time, then responding to user requests can be done in a different thread.
5. Run an iterative kernel on an image that is too large to fit on one node and must therefore be divided among several SuperCL instances on several nodes. While the next iteration is running, the edges from the previous iteration are sent assynchorneously (by a different thread), through the application, to other SuperCL instances on other nodes. Once edges arrive from other nodes, a different kernel can be used to integrate the edges with the main buffer/image.
6. Run a GPU kernel, then check the accuracy of the result using a CPU kernel: iterate until sufficient accuracy is achieved.
7. As above, but send intermediate results to the application (assynchroneously). The application may then report (assynchroneously) about the progress of other SuperCL instances on other nodes, which can affect whether to continue iterating and/or modify some parameters.

CURRENT STATUS

`SuperCL` is still evolving, so no binary backward-compatibility of future releases with the current release should be assumed.

SEE ALSO

`vcl(7).`