

Virtual OpenCL (VCL)

Cluster Platform

Guide and Manuals

Revised for VCL-1.13

January 2012

Preface

This document presents the Virtual OpenCL (VCL) and SuperCL guide and manuals. VCL is a cluster platform that allows OpenCL applications to transparently utilize many OpenCL devices in a cluster.

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

Contents

Preface	iii
1 The Virtual OpenCL	3
1.1 Overview	3
1.2 Using VCL	3
2 SuperCL	5
2.1 Overview	5
2.2 Using SuperCL	5
3 Installation	7
3.1 Automatic installation	7
3.2 Manual installation	7
4 Manuals	9
4.1 For users	9
4.2 For programmers	9

Chapter 1

The Virtual OpenCL

1.1 Overview

Virtual OpenCL (VCL) is a cluster platform that allows OpenCL applications to transparently utilize many OpenCL devices (GPUs, CPUs, APUs) in a cluster, as if all the devices are on the local computer.

The main features of VCL are:

- Can run unmodified OpenCL 1.1 applications on a cluster.
- Transparent selection of devices.
- Applications can be started on any hosting-computer, including workstations without GPU devices.
- Applications can utilize cluster-wide GPU devices.
- Supports multiple applications on the same cluster.
- Runs on Linux clusters, with or without MOSIX.

1.2 Using VCL

The VCL cluster platform allows unmodified OpenCL applications to transparently utilize many OpenCL devices (CPUs and/or GPUs) in a cluster. This is accomplished by a runtime environment in which all the cluster devices are seen as if they are located in each 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 completely unmodified applications, we 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.

Remote nodes perform OpenCL functions on behalf of application(s) on the host(s). In order to support application concurrency on different devices, our implementation is fully

thread-safe (even though this is not a requirement of the OpenCL specifications). 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: we therefore attempt to minimize the number of network round-trips for existing applications, to the extent possible without compromising the compliance to the OpenCL-specifications.

Chapter 2

SuperCL

2.1 Overview

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

2.2 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 execution 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 to activate them all and only one reply is received.

Chapter 3

Installation

3.1 Automatic installation

The simplest way to install VCL is to run “./vcl.install”.

If you prefer, you may install VCL manually.

3.2 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
vlconf	/sbin/vlconf
opencl	/sbin/opencl
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 “**vlconf**” or edit the VCL configuration manually, according to the instructions in “**man vcl**”.

Chapter 4

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.

4.1 For users

VCL - Virtual OpenCL

4.2 For programmers

SuperCL - Optimize remote OpenCL operations

NAME

VCL — Virtual OpenCL for combining the power of many GPUs

INTRODUCTION

The OpenCL standard allows programs 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.

VCL is an OpenCL platform which extends access to GPUs (and other OpenCL devices) beyond the devices of the local computer.

Users of **VCL** run their programs on hosting-nodes (hosts), using the **VCL** library instead of a vendor-specific SDK (OpenCL Software-Development-Kit). The actual OpenCL devices (eg. GPUs) 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 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 implementation of the `CL_MEM_FILE_HOST_PTR` extension (see 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**.

RUNNING ON HOSTING NODES

To run a program 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=0`)

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 OpenCL value of `DEFAULT_DEVICE_TYPE`, 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 a program 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 program.

Alternately use:

```
vclrun --sig={signum}
```

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-directory 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.

CURRENT STATUS

VCL fully supports OpenCL version 1.1 (and 1.0).

Outstanding problems:

1. There is no way (yet) for programs to know on which back-end node a given device resides.
2. CL-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. Due to lack of support in the OpenCL standard, the library cannot return an error when a program 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, programs that call `clGetEventInfo()` may not be immediately aware when a kernel transits from the `CL_SUBMITTED` state into the `CL_RUNNING` state. A delay of up to one minute may be expected.

SEE ALSO

`supercl(3)`.

NAME

superCL - Optimize remote OpenCL operations by reducing network overheads

PURPOSE

The main obstacle in using OpenCL devices on remote computers (nodes) is network latency: 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 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.

While SuperCL is included and most useful in the VCL(7) distributed OpenCL platform, it may also be convenient for general OpenCL use.

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 similar in structure to standard clEnqueueXXX() 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 library and can be freely mixed with the other OpenCL functions: please see the OpenCL manual (version 1.1) for the description of queue, num_events_in_wait_list, event_wait_list and event.

Sequence is an array of instructions, constituting a mini-program that invokes OpenCL kernels and I/O operations. Each instruction is 128 bytes long, beginning with the cmd (command) field - The rest of the instruction depends on the specific command. The sequence ends 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 first encountered error.

Registers are available for program control (such as loops) and for other features as described below. Register values can be either 64-bit integers or 64-bit real numbers. All registers are initially set to integer-zero. There is no need to pre-define registers and there is no hard limit on the number of registers used, but a large number of registers (1000's and more) can slow down SuperCL significantly.

Register numbers are 31-bit integers and there are two types of registers - global and private. Global registers are shared among all threads, while private registers belong to a specific thread and/or its child-threads.

The instructions are:

SCL_KERNEL_ARG

Set a kernel argument to a fixed value.

Instruction parameters are in:

```

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 in either `kernel_arg.arg_int`, `kernel_arg.arg_long`, `kernel_arg.arg_float`, `kernel_arg.arg_double` or `kernel_arg.arg_value`.

In the rare case when an argument's size is more than `SCL_MAX_ARG_LEN` bytes, several consecutive `SCL_KERNEL_ARG` instructions are needed, all with the same `kernel_arg.kernel`, `kernel_arg.argno` and `kernel_arg.arg_len` (set to the total argument length).

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.

Instruction parameters are as above in `SCL_KERNEL_ARG`, except that the data is taken from the register `kernel_arg.regno`. If `kernel_arg.reg_is_private` is set, then `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.

Instruction parameters are in:

```

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: a value of `NULL` implies the same device as the device associated with the queue – otherwise, in multi-node platforms such as `VCL(7)`, the device must be on the same node and same platform as the device of the queue.

`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()`.

SCL_COPY_BUFFER

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

Instruction parameters are in:

```

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.

Instruction parameters are in:

```

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.

Instruction parameters are in:

```

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.

Instruction parameters are in:

```

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_LOAD_REGISTER_FROM_BUFFER

Load a register from an element in an OpenCL buffer.

Instruction parameters are in:

```
{
    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 containing 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

Everything as in `LOAD_REGISTER_FROM_BUFFER`, except that the value of the register is loaded to the buffer element.

SCL_COPY_BUFFER_TO_HOST

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

Instruction parameters are in:

```
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 will arrive at the host-application in the same order as completed (including by other threads).

SCL_COPY_IMAGE_TO_HOST

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

Instruction parameters are in:

```

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 will arrive at the host-application in the same order as completed (including by other threads).

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 will arrive at the host-application in the same order as completed (including by other threads).

SCL_ARITHMETIC

Perform various register operations.

Instruction parameters are in:

```
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:

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

If any of the operands 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:

<code>SCLA_INT</code>	<code>op1 = round_down_to_integer(op1)</code>
<code>SCLA_LOG</code>	<code>op1 = ln(op1)</code>
<code>SCLA_EXP</code>	<code>op1 = exp(op1)</code>
<code>SCLA_SQRT</code>	<code>op1 = sqrt(op1)</code>
<code>SCLA_SIN</code>	<code>op1 = sin(op1)</code>
<code>SCLA_COS</code>	<code>op1 = cos(op1)</code>
<code>SCLA_ASIN</code>	<code>op1 = asin(op1)</code>

`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_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.

Instruction parameter is:

long label;

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

SCL_FORK

Start a new thread.

Instruction parameters are in:

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

A new thread starts running from the label `fork.label`. By default, the label (`fork.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 `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 ancestors 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 sequence.

Instruction parameters are in:

```
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 occurred 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 occurs, 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 jump backwards.
4. Do not fork backwards.
5. Do not arrive at `SCL_JOIN`.

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

Argument length > `SCL_MAX_ARG_LEN`, but the following instruction(s) do not match the same `SCL_KERNEL_ARG` operation.

CL_INVALID_VALUE

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

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 dimensions (a buffer when expecting an image or an image when expecting a buffer).

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

Count when copying data to host, is negative.

CL_INVALID_PROGRAM_EXECUTABLE:

Deadlock - all threads stuck in `SCL_JOIN`.

IN CONJUNCTION WITH VCL

All memory-objects that are mentioned in the `sequence` are migrated to the `queue`'s remote 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 use the same memory-objects in different instances (or as arguments of single OpenCL kernels) that run on different nodes, as that would result in serialisation of the operations and an extreme loss of performance. No serialisation, however, is caused by the use of OpenCL memory-object read/write/copy functions.

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

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, asynchronously 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 asynchronously (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 (asynchronously). The application may then report (asynchronously) 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 in the initial Alpha-testing stage. No backward-compatibility of future releases with the current release should be assumed.

FUTURE PLANS

1. Allow direct data-transfer between memory-objects and the filesystem. In the case of VCL, this means direct access to remote file-systems, saving on having to send data through the network.
2. Copying data asynchronously from memory-objects that are part of a `clSuper ()` instance to memory-objects that are not (and therefore possibly reside on different nodes)
3. A tool to find OpenCL devices that are on the same node and platform as a given device.

SEE ALSO

`vcl (7)`.