The OpenCL Platform Layer

The OpenCL platform layer implements platform-specific features that allow applications to query OpenCL devices, device configuration information, and to create OpenCL contexts using one or more devices.

Querying Platform Info & Devices

- `clGetPlatformIDs()` returns a list of platform IDs.
- `clGetPlatformInfo()` provides detailed information about platforms.

**Example:**
```
cl_int num_platforms;
cl_platform_id *platforms;

cl_int err = clGetPlatformIDs(1, &num_platforms, &platforms);
if (err == CL_SUCCESS) {
    // Process the list of platforms...
}
```

Partitioning a Device

- `clCreateSubDevices()` partitions a device into subdevices.

**Example:**
```
cl_device_id device_id;

cl_device_id *sub_devices;

cl_int err = clCreateSubDevices(device_id, 2, &num_sub_devices, &sub_devices);
if (err == CL_SUCCESS) {
    // Process the subdevices...
}
```

Contexts

- `clCreateContext()` creates a context.

**Example:**
```
cl_context *context;

cl_int err = clCreateContext(&context, 1, &device_id, NULL, NULL, 0);
if (err == CL_SUCCESS) {
    // Process the context...
}
```

The OpenCL Runtime

API calls that manage OpenCL objects such as command-queues, memory objects, program objects, kernels, and kernel arguments.

Command Queues

- `clCreateCommandQueue()` creates a command queue.

**Example:**
```
cl_command_queue *command_queue;

cl_int err = clCreateCommandQueue(context, device_id, CL_QUEUE_PROFILING_ENABLE, &command_queue);
if (err == CL_SUCCESS) {
    // Process the command queue...
}
```

Buffer Objects

Elements are stored sequentially and accessed using a pointer by a kernel executing on a device.

Create Buffer Objects

- `clCreateBuffer()` creates a buffer.

**Example:**
```
cl_mem *buffer;

cl_int err = clCreateBuffer(context, CL_MEM_READ_WRITE, size, NULL, &buffer);
if (err == CL_SUCCESS) {
    // Process the buffer...
}
```

Get Command Queue Properties

- `clGetCommandQueueInfo()` retrieves properties of a command queue.

**Example:**
```
cl_int num_properties;
cl_queue_properties *property_list;

cl_int err = clGetCommandQueueInfo(command_queue, CL_QUEUE_PROPERTIES, 0, NULL, &num_properties, &property_list);
if (err == CL_SUCCESS) {
    // Process the properties...
}
```

OpenCL API

- `clGetKernelInfo()` retrieves information about a kernel.

**Example:**
```
cl_kernel kernel;

cl_int err = clGetKernelInfo(kernel, CL_KERNEL_EXTENSION_NAME, 0, NULL, &num_properties, &property_list);
if (err == CL_SUCCESS) {
    // Process the extension name...
}
```

Source Code

- `clGetProgramInfo()` retrieves information about a program.

**Example:**
```
cl_program program;

cl_int err = clGetProgramInfo(program, CL_PROGRAM_BINARY_COMMANDS_LENGTH, NULL, &num_properties, &property_list);
if (err == CL_SUCCESS) {
    // Process the binary commands length...
}
```

Command Execution

- `clEnqueueWriteCommandBuffer()` submits a command buffer.

**Example:**
```
cl_command_buffer command_buffer;

cl_int err = clEnqueueWriteCommandBuffer(command_queue, CL_COMMAND_WRITE_IMAGE, &command_buffer, NULL, 0, NULL, 0, 0, 0);
if (err == CL_SUCCESS) {
    // Process the command buffer...
}
```

Exception Handling

- `clGetEventInfo()` retrieves information about an event.

**Example:**
```
cl_event event;

cl_int err = clGetEventInfo(event, CL_EVENT_COMMAND_EXECUTION_STATUS, 0, NULL, &num_properties, &property_list);
if (err == CL_SUCCESS) {
    // Process the execution status...
}
```

Kernel Execution

- `clEnqueueKernelLaunch()` launches a kernel.

**Example:**
```
cl_kernel kernel;

cl_int err = clEnqueueKernelLaunch(command_queue, kernel, NULL, 0, NULL, 0, 0, 0);
if (err == CL_SUCCESS) {
    // Process the kernel launch...
}
```

Memory Objects

- `clCreateImage()` creates an image object.

**Example:**
```
cl_mem *image;

cl_int err = clCreateImage(context, CL_MEM_READ_WRITE, image_format, dim, array_size, NULL, &image);
if (err == CL_SUCCESS) {
    // Process the image...
}
```

Contexts

- `clCreateContext()` creates a context.

**Example:**
```
cl_context *context;

cl_int err = clCreateContext(&context, num_devices, NULL, NULL, NULL, 0);
if (err == CL_SUCCESS) {
    // Process the context...
}
```

Command Queues

- `clCreateCommandQueue()` creates a command queue.

**Example:**
```
cl_command_queue *command_queue;

cl_int err = clCreateCommandQueue(context, device_id, CL_QUEUE_PROFILING_ENABLE, &command_queue);
if (err == CL_SUCCESS) {
    // Process the command queue...
}
```
Buffer Objects (continued)

cl_int clEnqueueWriteBuffer (  
cl_command_queue command_queue,  
cl_mem buffer,  
cl_bool blocking_write,  
size_t offset,  
size_t size,  
const void *ptr,  
cl_uint num_events_in_wait_list,  
const cl_event *event_wait_list,  
cl_event *event);  

cl_int clEnqueueWriteBufferRect (  
cl_command_queue command_queue,  
cl_mem buffer,  
cl_bool blocking_write,  
const size_t *buffer_origin,  
const size_t *buffer_row_pitch,  
const size_t *buffer_slice_pitch,  
const size_t *host_row_pitch,  
const size_t *host_slice_pitch,  
const void *ptr,  
cl_uint num_events_in_wait_list,  
const cl_event *event_wait_list,  
cl_event *event);  

cl_int clEnqueueFillBuffer (  
cl_command_queue command_queue,  
cl_mem buffer,  
const void *data,  
size_t offset,  
size_t size,  
cl_uint num_events_in_wait_list,  
const cl_event *event_wait_list,  
cl_event *event);  

cl_int clEnqueueCopyBuffer (  
cl_command_queue command_queue,  
cl_mem src_buffer,  
cl_mem dst_buffer,  
const size_t *src_origin,  
const size_t *src_row_pitch,  
const size_t *src_slice_pitch,  
const size_t *dst_origin,  
const size_t *dst_row_pitch,  
const size_t *dst_slice_pitch,  
cl_mem_migration_flags flags,  
const void *user_data,  
size_t param_value_size,  
const cl_event *event_wait_list,  
cl_event *event);  

cl_int clEnqueueCopyBufferRect (  
cl_command_queue command_queue,  
cl_mem src_buffer,  
cl_mem dst_buffer,  
const size_t *src_origin,  
const size_t *src_row_pitch,  
const size_t *src_slice_pitch,  
const size_t *dst_origin,  
const size_t *dst_row_pitch,  
const size_t *dst_slice pitch,  
cl_mem_migration_flags flags,  
const void *user_data,  
size_t param_value_size,  
const cl_event *event_wait_list,  
cl_event *event);  

cl_int clEnqueueUnmapMemObject (  
cl_command_queue command_queue,  
cl_mem memobj,  
const size_t *mem_offset,  
const size_t *mem_size,  
cl_uint num_events_in_wait_list,  
const cl_event *event_wait_list,  
cl_event *event);  

Memory Objects
A memory object is a handle to a reference counted region of global memory.  
Includes Buffer Objects, Image Objects, and Pipe Objects.  
Items in blue apply when the appropriate extension is supported.

Conversions and Type Casting Examples [5.2.4]

T a = (T b);  
// Scalar to scalar,  
// or array to vector  
T a = convert_ _T ( b );  
T a = convert_ _ T R ( b );  
T a = convert_ _R _T ( b );  
T a = as_ _T ( b );  
T a = convert_ _T s _R _T ( b );  
R: one of the following rounding modes:

Pipe Object Queries [5.4.2]

cl_int clGetPipeInfo (  
cl_mem pipe,  
const cl_pipe_info *param_name,  
s_size_t *param_value_size,  
void *param_value,  
size_t *param_value_size_ret);  

cl_int clCreatePipe (  
cl_context context,  
const cl_pipe_properties *flags,  
cl_uint pipe_max_packets,  
cl_mem_info *memobj);  

cl_mem clCreatePipe (  
cl_context context,  
cl_mem_flags flags,  
cl_uint pipe_max_packets,  
const cl_mem_properties *properties,  
cl_int errcode_ret);  

Pipe flags:
0 or CL_PIPE_READ_WRITE, CL_PIPE_READ_WRITE_ONLY,  
CL_MEM_HOST_NO_ACCESS

Shared Virtual Memory
Shared Virtual Memory (SVM) allows the host and kernels executing on devices to directly  
share complex, pointer-containing data structures such as trees and linked lists.

SVM Sharing Granularity [5.5.1]

void *dsVMAlloc (  
cl_context context,  
cl_mem_flags flags,  
s_size_t size,  
unsigned int alignment)  

flags:  
cl_write, cl_read, cl_map (CL_MAP_WRITE, CL_MAP_READ)  

cl_int clSVMFree (  
cl_mem *memobj,  
void *svm_pointer);  

Enqueuing SVM Operations [5.5.2]

cl_int clEnqueueSVMFree (  
cl_command_queue command_queue,  
cl_uint num_svm_pointers,  
void *svm_pointers[]);  

cl_int clEnqueueSVMMap (  
cl_command_queue command_queue,  
cl_uint num_svm_pointers,  
const size_t *svm_pointers[],  
void *user_data,  
const size_t *param_value_size,  
void *param_value,  
const cl_event *event_wait_list,  
cl_event *event);  

(Continued on next page >)
OpenCL 2.0 Reference Guide

Shared Virtual Memory (continued)

cl_int clEnqueueSVMMemcpy (cl_command_queue command_queue, cl_bool blocking_copy, void *dst_ptr, size_t size, cl_uint num_events_in_wait_list, const cl_event *event_wait_list, cl_event *event)

cl_int clEnqueueSVMMemFill (cl_command_queue command_queue, void *sym_ptr, const void *pattern, size_t pattern_size, size_t size, cl_uint num_events_in_wait_list, const cl_event *event_wait_list, cl_event *event)

cl_int clEnqueueSVMMap (cl_command_queue command_queue, cl_bool blocking_map, cl_map_flags map_flags, void *sym_ptr, size_t size, cl_uint num_events_in_wait_list, const cl_event *event_wait_list, cl_event *event)

cl_int clEnqueueSVMUnmap (cl_command_queue command_queue, void *sym_ptr, cl_int num_events_in_wait_list, const cl_event *event_wait_list, cl_event *event)

Kernel Objects

A kernel is a function declared in a program, identified by the _kernel qualifier. A kernel object encapsulates the specific kernel function and the argument values to be used when executing it. Items in blue apply when the appropriate extension is supported.

Create Kernel Objects

[5.9.1, 5.9.2, 5.9.3]

cl_kernel clCreateKernel (cl_program program, const char *kernel_name, cl_int *errcode_ret)

cl_kernel clCreateKernelsInProgram (cl_program program, cl_uint num_kernels, cl_kernel *kernels, cl_uint *num_kernels_ret)

cl_int clRetainKernel (cl_kernel kernel)

cl_int clReleaseKernel (cl_kernel kernel)

Kernel Arguments and Queries

[5.9.2, 5.9.3]

cl_int clGetKernelArg (cl_kernel kernel, cl_uint arg_index, size_t *arg_size, void *arg_value)

cl_int clGetKernelArgSVMPointer (cl_kernel kernel, cl_uint arg_index, cl_ulong *arg_svm_ptr)

cl_int clGetKernelInfo (cl_kernel kernel, cl_kernel_info_info param_name, size_t *param_value_size, void *param_value, size_t *param_value_size_ret)

cl_int clGetKernelWorkGroupInfo (cl_kernel kernel, cl_device_id dev_id, cl_kernel_work_group_info param_name, size_t *param_value_size, void *param_value, size_t *param_value_size_ret)

cl_int clSetUserEventStatus (cl_event event, cl_int execution_status)

cl_int clWaitForEvents (cl_uint num_events, const cl_event *event_list)

cl_int clGetEventInfo (cl_event event, cl_event_info_info param_name, size_t *param_value_size, void *param_value, size_t *param_value_size_ret)

cl_int clGetKernelInfo (cl_kernel kernel, cl_kernel_info_info param_name, size_t *param_value_size, void *param_value, size_t *param_value_size_ret)

cl_int clGetKernelArg (cl_kernel kernel, cl_uint arg_index, size_t *arg_size, void *arg_value)

cl_int clEnqueueNDRangeKernel (cl_command_queue command_queue, cl_kernel kernel, cl_uint work_dim, size_t *global_work_size, size_t *local_work_size, size_t *global_work_offset, size_t *local_work_offset, cl_uint num_events_in_wait_list, cl_event *event_wait_list, cl_event *event)

cl_int clEnqueueNativeKernel (cl_command_queue command_queue, void *clallback(pfn_notify), void *user_data, size_t *param_value_size, void *param_value, size_t *param_value_size_ret)

cl_int clCreateUserEvent (cl_context context, cl_int *errcode_ret)

cl_int clSetUserEventStatus (cl_event event, cl_int execution_status)

cl_int clWaitForEvents (cl_uint num_events, const cl_event *event_list)

cl_int clGetEventInfo (cl_event event, cl_event_info_info param_name, size_t *param_value_size, void *param_value, size_t *param_value_size_ret)

cl_int clEnqueueNDRangeKernel (cl_command_queue command_queue, cl_kernel kernel, cl_uint work_dim, size_t *global_work_size, size_t *local_work_size, size_t *global_work_offset, size_t *local_work_offset, cl_uint num_events_in_wait_list, cl_event *event_wait_list, cl_event *event)

cl_int clEnqueueNativeKernel (cl_command_queue command_queue, void *clallback(pfn_notify), void *user_data, size_t *param_value_size, void *param_value, size_t *param_value_size_ret)

cl_int clCreateUserEvent (cl_context context, cl_int *errcode_ret)

Flush and Finish

[5.15]

cl_int clFlush (cl_command_queue command_queue)

cl_int clFinish (cl_command_queue command_queue)

Program Objects

An OpenCL program consists of a set of kernels that are identified as functions, declared with the _kernel qualifier in the program source.

Create Program Objects

[5.8.1]

cl_program clCreateProgramWithSource (cl_context context, cl_uint count, const char **strings, cl_int *lengths, cl_int *errcode_ret)

cl_program clCreateProgramWithBinary (cl_context context, cl_uint num_devices, cl_device_id *device_list, const size_t *lengths, const char **strings, cl_int *errcode_ret)

cl_program clCreateProgramWithBuiltInKernels (cl_context context, cl_uint num_devices, cl_device_id *device_list, const char *kernel_names, cl_int *errcode_ret)

cl_int clRetainProgram (cl_program program)

cl_int clReleaseProgram (cl_program program)

Building Program Executables

[5.8.2]

cl_program clBuildProgram (cl_program program, cl_uint num_devices, const cl_device_id *device_list, const char *options, void *clallback(pfn_notify), void *user_data, size_t *param_value_size, void *param_value, size_t *param_value_size_ret)

Separate Compilation and Linking

[5.8.3]

cl_program clCompileProgram (cl_program program, cl_uint num_devices, const cl_device_id *device_list, const char *options, cl_int *input_headers, cl_int *output_headers, void *clallback(pfn_notify), void *user_data, size_t *param_value_size, void *param_value, size_t *param_value_size_ret)

Execute Kernels

[5.10]

cl_event clEnqueueNDRangeKernel (cl_command_queue command_queue, cl_kernel kernel, cl_uint work_dim, size_t *global_work_size, size_t *local_work_size, size_t *global_work_offset, size_t *local_work_offset, cl_uint num_events_in_wait_list, cl_event *event_wait_list, cl_event *event)

cl_event clEnqueueNativeKernel (cl_command_queue command_queue, cl_kernel kernel, void *clallback(pfn_notify), void *user_data, size_t *param_value_size, void *param_value, size_t *param_value_size_ret)

Event Objects

Event objects can be used to refer to a kernel execution command, and read, write, map and copy commands on memory objects or user events.

Event Objects

[5.11]

cl_event clCreateUserEvent (cl_context context, cl_int *errcode_ret)
markers, barriers, waiting for events [5.12]

cl_int clEnqueueMarkerWithWaitList (  
    cl_command_queue command_queue,  
    cl_uint num_events_in_wait_list,  
    const cl_event *event_wait_list, cl_event *event)

cl_int clEnqueueBarrierWithWaitList (  
    cl_command_queue command_queue,  
    cl_uint num_events_in_wait_list,  
    const cl_event *event_wait_list, cl_event *event)

profiling operations [5.14]

cl_int clGetEventProfilingInfo (cl_event event,  
    cl_profiling_info param_name,  
    size_t *param_value_size, void *param_value,  
    size_t *param_value_size_ret)  

param_name: [table 5.23]

cl_PROFILING_COMMAND_COMPLETE,  
cl_PROFILING_COMMAND_SUBMIT, START, END

operators and qualifiers [6.3]

these operators behave similarly as in c99 except
operators may include vector types when possible:

++ == != & ^ ~ & | & ||  
+ - * % / " " 
> > < <= <= <> > < >> << 

operators include vector types:

float2 v,  
float3 v,  
float4 v,  
float8 v,  
float16 v,  

vector addressing equivalences:

<table>
<thead>
<tr>
<th>v.o</th>
<th>v.h</th>
<th>v.o</th>
<th>v.e</th>
</tr>
</thead>
<tbody>
<tr>
<td>v.x</td>
<td>v.y</td>
<td>v.z</td>
<td>v.s</td>
</tr>
<tr>
<td>v.x</td>
<td>v.y</td>
<td>v.z</td>
<td>v.s</td>
</tr>
<tr>
<td>v.x</td>
<td>v.y</td>
<td>v.z</td>
<td>v.s</td>
</tr>
<tr>
<td>v.x</td>
<td>v.y</td>
<td>v.z</td>
<td>v.s</td>
</tr>
<tr>
<td>v.x</td>
<td>v.y</td>
<td>v.z</td>
<td>v.s</td>
</tr>
<tr>
<td>v.x</td>
<td>v.y</td>
<td>v.z</td>
<td>v.s</td>
</tr>
<tr>
<td>v.x</td>
<td>v.y</td>
<td>v.z</td>
<td>v.s</td>
</tr>
<tr>
<td>v.x</td>
<td>v.y</td>
<td>v.z</td>
<td>v.s</td>
</tr>
<tr>
<td>v.x</td>
<td>v.y</td>
<td>v.z</td>
<td>v.s</td>
</tr>
</tbody>
</table>

address space qualifiers [6.5]

global, local, const, private

function qualifiers [6.7]

_kernel, kernel  
_attribute_1(type)  
//type defaults to int  
//type defaults to int

work-item built-in functions [6.13.1]

query the number of dimensions, global and local work size
specified to clEnqueueNDRangeKernel, and global and local
identifier of each work-item when this kernel is executed on a
device. sub-groups require the cl_khr_subgroups extension.

uint get_work_dim ()  

size_t get_global_size (uint dimid)  
size_t get_global_id (uint dimid)  
size_t get_local_size (uint dimid)
## Math Built-in Functions

### [6.13.2] [9.4.2]

These built-in functions are available for use in OpenCL expressions and are grouped into several categories:

- **Basic Mathematical Functions**
- **Logarithmic and Exponential Functions**
- **Power Functions**
- **Trigonometric Functions**
- **Rounding Functions**
- **Special Functions**

### Math Constants

The values of the following symbolic constants are single-precision floats:

- MAXFLOAT: Value of maximum non-infinite single-precision floating-point number.
- HUGE_VALF: Positive float expression, evaluates to +infinity.
- HUGE_VAL: Positive double expression, evals. to ‘infinity’ or overflows.
- INFINITY: Constant float expression, positive or unsigned infinity.
- NAN: Constant float expression, quiet NaN.

When double precision is supported, macros ending in _F are available in type double by removing _F from the macro name, and in type half when the cl_khr_fp16 extension is enabled by replacing _F with _H.

### Integer Built-in Functions

#### [6.13.3]

- **abs**: Absolute value function.
- **abs**: Maximum of two values.
- **max** and **min**: Return the maximum or minimum of two or more values.
- **clamp**: Clamps a value to a minimum and maximum value.
- **round**: Round a value to the nearest integer.
- **trunc**: Truncates a value to the nearest integer.

### Attribute Qualifiers

- **__attribute__((aligned))**: Align a memory location to a specified alignment.
- **__attribute__((aligned(alignment)))**: Align a memory location to a specified alignment.
- **__attribute__((aligned(alignment), nosvm))**: Align a memory location to a specified alignment without using the SVM.

### Common Built-in Functions

These functions operate component-wise and use round to nearest even rounding mode. They return the vector form of T, and Tn.

### OpenCL C Language

This section provides a detailed reference guide for the OpenCL C programming language, including syntax, semantics, and usage examples. It covers a wide range of topics from basic language constructs to advanced features like memory management and parallel programming.
### Common Functions (continued)

- `vec_t mix (vec_t a, vec_t b, float f)`
  - Description: Linear blend of `a` and `b`
  - Parameters: `a`, `b`, `f`
  - Returns: `vec_t`

- `vec_t cross (vec_t a, vec_t b, vec_t c)`
  - Description: Cross product of `a`, `b`, and `c`
  - Parameters: `a`, `b`, `c`
  - Returns: `vec_t`

- `vec_t normalize (vec_t v)`
  - Description: Normalizes the vector `v`
  - Parameters: `v`
  - Returns: `vec_t`

### Geometric Built-in Functions [6.13.5] [9.4.4]

- `T is_vec (vec_t v)`
  - Description: Test if `v` is a vector
  - Parameters: `v`
  - Returns: `bool`

- `T is_vec (vec4_t v)`
  - Description: Test if `v` is a vector
  - Parameters: `v`
  - Returns: `bool`

- `T is_type (vec_t v)`
  - Description: Test if `v` is a vector of a specific type
  - Parameters: `v`
  - Returns: `bool`

- `T is_type (vec4_t v)`
  - Description: Test if `v` is a vector of a specific type
  - Parameters: `v`
  - Returns: `bool`

### Relational Built-in Functions [6.13.6]

- `T is_equal (vec_t a, vec_t b)`
  - Description: Test if `a` and `b` are equal
  - Parameters: `a`, `b`
  - Returns: `bool`

- `T is_unequal (vec_t a, vec_t b)`
  - Description: Test if `a` and `b` are unequal
  - Parameters: `a`, `b`
  - Returns: `bool`

### Vector Data Load/Store [6.13.7] [9.4.6]

- `T load (T *p)`
  - Description: Load data from memory
  - Parameters: `T *p`
  - Returns: `T`

- `T load (T *p)`
  - Description: Load data from memory
  - Parameters: `T *p`
  - Returns: `T`

- `T load (T *p)`
  - Description: Load data from memory
  - Parameters: `T *p`
  - Returns: `T`

- `T load (T *p)`
  - Description: Load data from memory
  - Parameters: `T *p`
  - Returns: `T`

### Async Copies and Prefetch [6.13.10] [9.4.7]

- `T event_t async_work_group_copy (global T *src, size_t num_gentypes)`
  - Description: Copy work group to memory
  - Parameters: `T *src`, `size_t num_gentypes`
  - Returns: `event_t`

- `T event_t async_work_group_copy (global T *src, size_t num_gentypes)`
  - Description: Copy work group to memory
  - Parameters: `T *src`, `size_t num_gentypes`
  - Returns: `event_t`

- `T event_t async_work_group_copy (global T *src, size_t num_gentypes)`
  - Description: Copy work group to memory
  - Parameters: `T *src`, `size_t num_gentypes`
  - Returns: `event_t`

- `T event_t async_work_group_copy (global T *src, size_t num_gentypes)`
  - Description: Copy work group to memory
  - Parameters: `T *src`, `size_t num_gentypes`
  - Returns: `event_t`
Atomic Functions [6.13.11]

OpenCL C implements a subset of the C11 atomics (see 7.17 of the C11 spec.) and synchronization operations.

Atomic Functions

In the following definitions, A refers to one of the atomic_*, M refers to the type of the other argument for arithmetic operations. For atomic integer types, M is C. For atomic pointer types, M is ptrdiff_t. The type atomic_t is a 32-bit integer. atomic_long and atomic_ulong require extension cl_khr_int64_base_atomics or cl_khr_int64_extended_atomics. The atomic_double type requires double precision support. The default scope is work_group for local atomics and all_svm_devices for global atomics.

See the table under Atomic Types and Enum Constants for information about parameter types memory_order, memory_scope, and memory_flag.

Atomic Functions

void atomic_init(volatile A *obj, C value)
Initializes the atomic object pointed to by obj to the value value.

void atomic_work_item_fence(cl_mem_fence_flags, memory_order order, memory_scope scope)
Effects based on value of order. flags must be CLK_GLOBAL, LOCAL, IMAGE_MEM_FENCE or a combination of these.

void atomic_store(volatile A *object, C desired)
Atomicity replaces the value pointed to by object with the value of desired. Memory is affected according to the value of order.

C atomic_load(volatile A *object)
C atomic_load_explicit(volatile A *object, memory_order order, memory_scope scope)
Atomically returns the value pointed to by object. Memory is affected according to the value of order.

C atomic_exchange(volatile A *object, C desired)
C atomic_exchange_explicit(volatile A *object, C desired, memory_order order, memory_scope scope)
Atomically replaces the value pointed to by object with desired. Memory is affected according to the value of order.

bool atomic_compare_exchange_strong(volatile A *object, C expected, C desired)
bool atomic_compare_exchange_strong_explicit(volatile A *object, C expected, C desired, memory_order order, memory_scope scope)
Atomically compares the value pointed to by object for equality with that in expected, and if true, replaces the value pointed to by object with desired and, if false, updates the value in expected with the value pointed to by object. Further, if the comparison is true, memory is affected according to the value of success, and if the comparison is false, memory is affected according to the value of failure. These operations are atomic read-modify-write operations.

bool atomic_compare_exchange_weak(volatile A *object, C expected, C desired)
bool atomic_compare_exchange_weak_explicit(volatile A *object, C expected, C desired, memory_order order, memory_scope scope)
Atomically sets the value pointed to by object false. The order expected% is to be defined.

C atomic_fetch_key(volatile A *object, M operand)
C atomic_fetch_key_explicit(volatile A *object, M operand, memory_order order, memory_scope scope)
Atomically replaces the value pointed to by object with the result of the computation applied to the value pointed to by object and the given operand. Memory is affected according to the value of order. key% is to be defined.

bool atomic_flag_test_and_set(volatile atomic_flag *object)
bool atomic_flag_test_and_set_explicit(volatile atomic_flag *object, memory_order order, memory_scope scope)
Atomically sets the value pointed to by object to true. Memory is affected according to the value of order. Returns atomically, the value of the object immediately before the effects.

void atomic_flag_clear(volatile atomic_flag *object)
void atomic_flag_clear_explicit(volatile atomic_flag *object, memory_order order, memory_scope scope)
Atomically sets the value pointed to by object to false. The order argument shall not be memory_order_acquire nor memory_order_acq_rel. Memory is affected according to the value of order.

Atomic Types and Enum Constants

Values for key for atomic_fetch* functions

<table>
<thead>
<tr>
<th>key</th>
<th>op</th>
<th>computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>subtraction</td>
</tr>
<tr>
<td></td>
<td>^</td>
<td>bitwise exclusive or</td>
</tr>
</tbody>
</table>

Atomic Types

<table>
<thead>
<tr>
<th>Parameter Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory_order</td>
<td>memory_order_relaxed</td>
<td>Enum which identifies memory ordering constraints.</td>
</tr>
<tr>
<td></td>
<td>memory_order_release</td>
<td></td>
</tr>
<tr>
<td></td>
<td>memory_order.acquire</td>
<td></td>
</tr>
<tr>
<td></td>
<td>memory_order_acq_rel</td>
<td></td>
</tr>
<tr>
<td>memory_scope</td>
<td>memory_scope.work.item</td>
<td>Enum which identifies scope of memory ordering constraints. memory_scope_sub_group requires the cl_khr_subgroups extension.</td>
</tr>
<tr>
<td></td>
<td>memory_scope.work.group</td>
<td></td>
</tr>
<tr>
<td></td>
<td>memory_scope.sub_group</td>
<td></td>
</tr>
<tr>
<td></td>
<td>memory_scope.device</td>
<td></td>
</tr>
</tbody>
</table>

Atomic integer and floating-point types

† indicates types supported by a limited subset of atomic operations
‡ indicates size depends on whether implemented on 64-bit or 32-bit architecture
§ indicates types supported only if both 64-bit extensions are supported

Atomic Macros

#define ATOMIC_VAR_INIT(value) \ns of a token sequence to initialize an atomic object of a type that is initialization-compatible with value.

#define ATOMIC_FLAG_INIT(\n) \ninitialize an atomic flag to the clear state.

64-bit Atomics [9.3]

The cl_khr_int64_base_atomics extension enables 64-bit versions of the following functions: atom_add, atom_sub, atom_inc, atom_dec, atom_xchg, atom_cmpxchg

The cl_khr_int64_extended_atomics extension enables 64-bit versions of the following functions: atom_min, atom_max, atom_and, atom_or, atom_xor

Address Space Qualifier Functions [6.13.9]

When the event associated with a particular kernel invocation completes, the output of applicable printf calls is flushed to the implementation-defined output stream.

printf format string

The format string follows C99 conventions and supports an optional vector specifier: %q|flags|width|precision|vector|length conversion

Examples:

The following examples show the use of the vector specifier in the printf format string.

• float4 f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
printf("f4 = %2.2v4f", f);

Output: f4 = 1.00,2.00,3.00,4.00

uchar4 uc = (uchar4)(0xFA, 0xFB, 0xFC, 0xFD);
printf("uc% = \%hux4", uc);

Output: uc% = 0xFAh,0xFBh,0xFCh,0xFDh

uint2 ui = (uint2)(0x12345678, 0x07654321);
printf("unsigned short value = \%hux2", ui);

Output: unsigned short value = 0x5678,0x4321
Pipe Built-in Functions [6.13.16.2-4]

T represents the built-in OpenCL C scalar or vector integer or floating-point data types or any user-defined type defined built from these scalar and vector data types. Half scalar and vector types require the cl_khr_subgroups extension. Sub-groups require Double or vector double types require double precision support. The macro CLK_NULL_RESERVE_ID refers to an invalid reservation ID.

```
int read_pipe (pipe T p, T *ptr)
Read packet from p into ptr.

int read_pipe (pipe T p, reserve_id_t reserve_id, index T index)
Read packet from reserved area of the pipe reserve_id and index into ptr.

void write_pipe (pipe T p, T *ptr)
Write packet by ptr to p.

uint write_pipe_max_packets (pipe T p)
Returns maximum number of packets specified when p was created.

bool is_valid_reserve_id (reserve_id_t reserve_id)
Return true if reserve_id is a valid reservation ID and false otherwise.
```

Enqueuing and Kernel Query Built-in Functions [6.13.17] [9.7.3.4]

A kernel may enqueue code represented by Block syntax, and control execution order with event dependencies including user events and markers. There are several advantages to using the Block syntax: it is more compact; it does not require a cl kernel object; and enqueuing can be done as a single semantic step. Sub-groups require the cl_khr_subgroups extension. The macro CLK_NULL_EVENT refers to an invalid device event. The macro CLK_NULL_QUEUE refers to an invalid device queue.

```
int enqueue_kernel (queue_t queue, kernel_t kernel, flags T flags, const ndrange_t n, int n_events_in_wait_list, const event_t *event_wait_list, clk_event_t clk_event_t *event_wait_list, clk_event_t clk_event_t *event_wait_list, void *block[void])
Allows a work-item to enqueue a block for execution to queue. Work-items can enqueue multiple blocks to a device queue(). Flags may be one of CLK_ENQUEUE_FLAGS
(NO_WAIT, WAIT_KERNEL, WAIT_WORK_GROUP)

int enqueue_kernel (queue_t queue, kernel_t kernel, flags T flags, const ndrange_t n, int n_events_in_wait_list, const event_t *event_wait_list, clk_event_t clk_event_t *event_wait_list, void *block[void])
Enqueue a kernel to a device queue. Flags may be one of CLK_ENQUEUE_FLAGS
(NO_WAIT, WAIT_KERNEL, WAIT_WORK_GROUP)

void get_kernel_work_group_size (void *block[void])
Get the current work group size.

void get_kernel_work_group_size (void *block[void])
Get the current work group size.

void get_kernel_work_group_size (void *block[void])
Get the current work group size.

int enqueue_kernel (queue_t queue, kernel_t kernel, flags T flags, const ndrange_t n, int n_events_in_wait_list, const event_t *event_wait_list, clk_event_t clk_event_t *event_wait_list, void *block[void])
Enqueue a kernel to a device queue. Flags may be one of CLK_ENQUEUE_FLAGS
(NO_WAIT, WAIT_KERNEL, WAIT_WORK_GROUP)

int enqueue_kernel (queue_t queue, kernel_t kernel, flags T flags, const ndrange_t n, int n_events_in_wait_list, const event_t *event_wait_list, clk_event_t clk_event_t *event_wait_list, void *block[void])
Enqueue a kernel to a device queue. Flags may be one of CLK_ENQUEUE_FLAGS
(NO_WAIT, WAIT_KERNEL, WAIT_WORK_GROUP)

void get_kernel_work_group_size (void *block[void])
Get the current work group size.
```

Miscellaneous Vector Functions [6.13.12]

Tm and Tn are type char, uchar, short, ushort, int, uint, long, ulong, float, optionally double, or half if the cl_khr_fp16 extension is supported, where n is 2, 4, 8, or 16 except in vec_step it may also be 3. Tm is uchar, ushort, uint, or ulong.

```
int vec_step (Tn a)
Takes a built-in scalar or vector data type argument. Returns 1 for scalar, 4 for 3-component vector, else number of elements in the specified type.

in shuffle (Tm x, Tn mask)
Takes a 2D shuffle mask and a Tm data item. Returns a Tn result with element value modified by the corresponding value in the shuffle mask.

in shuffle (Tm x, Tm y, Tn mask)
Takes a 2D shuffle mask and two Tm data items. Returns a Tn result with element value in the first value modified by the corresponding value in the first shuffle mask and the second value modified by the corresponding value in the second shuffle mask.
```

Event Built-in Functions [6.13.17.8]

T is type int, uint, long, ulong, or float, optionally double, or half if the cl_khr_fp16 extension is enabled.

```
void retain (clk_event_t event)
Increments event reference count.

void release_event (clk_event_t event)
Decrements event reference count.

void ret kevent (clk_event_t event)
Create a user event.

bool is_valid_event (clk_event_t event)
True for valid event.

void set_user_event_status (clk_event_t event, int status)
Sets the execution status of a user event. Status: CL_COMPLETE or a negative error value.

void capture_event_profiling_info (clk_event_t event, int* name, global void *value)
Captures profiling information for command associated with event in value.
```

Helper Built-in Functions [6.13.17.9]

```
queue_t get_default_queue (void)
Default queue or CLK_NULL_QUEUE

ndrange_t ndrange_t (size_t global_work_size)
Builds a 1D ND-range descriptor.

ndrange_t ndrange_t (size_t global_work_size)
Builds a 2D or ND-range descriptor. n may be 2 or 3.
```

OpenCL Image Processing Reference

A subset of the OpenCL API and C Language specifications pertaining to image processing and graphics.
Image Objects (continued)

Map and Unmap Image Objects

```c
void *cldImageMap(cl_command_queue command_queue,
   cl_mem image, cl_bool blocking_map,
   cl_map_flags map_flags, const size_t *origin,
   const size_t *region, size_t *image_rowpitch,
   size_t *image_slicepitch, cl_uint num_events_in_wait_list,
   const cl_event *event_wait_list, cl_event *event, cl_int *error_ret)
```

map_flags: CL_MAP_READ|WRITE, CL_MAP_WRITE_INVALIDATE_REGION

Query Image Objects

```c
cl_int cldGetImageInfo(cl_mem image, const char *param_name, void *param_value, size_t *param_value_size, void *param_value_size_ret)
```

param_name: [Table 5.9] (Table 5.10)

<table>
<thead>
<tr>
<th>Image 1D Buffer</th>
<th>Image 2D Array</th>
<th>Image 3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMAGE_3D</td>
<td>IMAGE_2D_ARRAY</td>
<td>IMAGE_3D</td>
</tr>
</tbody>
</table>

OpenCL Image Processing

Image Formats

Supported image formats: image_channel_order with image_channel_data_type.

- void write_imagef(image1d_t image, int coord, color)
- void write_imageh(image1d_t image, int coord, color)
- void write_imagei(image1d_t image, int coord, color)
- void write_imageui(image1d_t image, int coord, color)
- void write_imagef(image2d_t image, int coord, int4 color)
- void write_imageh(image2d_t image, int coord, int4 color)
- void write_imagei(image2d_t image, int coord, int4 color)
- void write_imageui(image2d_t image, int coord, int4 color)
- void write_imagef(image2d_array_t image, int coord, int4 color)
- void write_imageh(image2d_array_t image, int coord, int4 color)
- void write_imagei(image2d_array_t image, int coord, int4 color)
- void write_imageui(image2d_array_t image, int coord, int4 color)
- void write_imagef(image3d_t image, int coord, int4 color)
- void write_imageh(image3d_t image, int coord, int4 color)
- void write_imagei(image3d_t image, int coord, int4 color)
- void write_imageui(image3d_t image, int coord, int4 color)

Optional support: [Table 5.6]

- CL_R, CL_A: CL_HALF_FLOAT, CL_FLOAT, CL_UNIFORM_INT8(8,16), CL_SIGNED_INT8(16,32), CL_UNSIGNED_INT8(16,32), CL_SNORM_INT8(16,32)
- CL_INTENSITY: CL_HALF_FLOAT, CL_FLOAT, CL_UNIFORM_INT8(8,16), CL_SIGNED_INT8(16,32), CL_UNSIGNED_INT8(16,32), CL_SNORM_INT8(16,32)
- CL_DEPTH_STENCIL: Only used if extension cl_khr_gl_depth_images is enabled and channel data type = CL_UNORM_INT8 or CL_FLOAT
- CL_LUMINANCE: CL_UNIFORM_INT8(8,16), CL_HALF_FLOAT, CL_FLOAT, CL_SNORM_INT8(16,32)
- CL_RGB: CL_UNIFORM_SHORT(5,6,5), CL_UNIFORM_INT(10,10,10), CL_UNSIGNED_INT(8,8,8,8), CL_SIGNED_INT(8,8,8,8), CL_SNORM_INT(8,8,8,8)
- CL_RGBA: CL_UNIFORM_INT8(8,8,8,8), CL_SIGNED_INT8(8,8,8,8), CL_SNORM_INT8(8,8,8,8), CL_HALF_FLOAT, CL_FLOAT, CL_UNORM_INT(8,8,8,8), CL_HALF_FLOAT, CL_FLOAT, CL_UNORM_SHORT(5,6,5,6)
- CL_BGRA: CL_HALF_FLOAT, CL_FLOAT, CL_UNORM_INT24, CL_HALF_FLOAT, CL_FLOAT, CL_UNORM_HASHED_RGB

Read and write functions for 2D images (continued)

```c
void write_imagef(image2d_t image, int coord, int4 color)
void write_imageh(image2d_t image, int coord, int4 color)
void write_imagei(image2d_t image, int coord, int4 color)
void write_imageui(image2d_t image, int coord, int4 color)
```
Extended mipmap read and write functions [9.18.2.1] These functions require the cl_khr_mipmap_image and cl_khr_mipmap_write_image extensions.

```c
float read_imagef(image2d_t_depth_t_image, sampler_t sampler, float2 coord, float2 gradient_x, float2 gradient_y)
uint4 read_imageui(image2d_t_image, sampler_t sampler, float2 coord, float2 gradient_x, float2 gradient_y)
```

Extended multi-sample image read functions [9.12.3] The extension cl_khr_gl_msaas_sharing adds the following built-in functions:

```c
float read_imagef(image2d_array_t_image, int2 coord, int sample)
float read_imageui(image2d_array_depth_msaa_t_image, int2 coord, int sample)
```

Sampler Declaration Fields [6.13.14.1] The sampler can be passed as an argument to the kernel using cGetSamplerInfo or can be declared in the outermost scope of kernel functions, or it can be a constant variable of type sampler_t declared in the program source.

```c
core_sampler_t<normalization_mode> = <normalization_mode> | <address-mode> | <filter-mode>
```

Access Qualifiers [6.6] Apply to 2D and 3D image types to declare if the image memory object is being read or written by a kernel.

```c
__read_only, read only
__write_only, write only
```
OpenCL Extensions [9]
These functions require the cl_khr_gl_sharing or cl_apple_gl_sharing extension.

CL Context > GL Context, Sharegroup [9.5.5]
cl_int clGetGLContextInfoKHR
(const cl_context_properties *properties, const cl_context_info_info *info, size_t *param_value_size, void *param_value, size_t *param_value_size_ret)

cl_int clCreateFromGLContextGroupKHR
(cl_context context, cl_mem flags, cl_int *errcode_ret)
flags: See clCreateFromGLContextGroup

CL Buffer Objects > GL Buffer Objects [9.6.2]
cl_mem clCreateFromGLBuffer (cl_context context, cl_mem_flags flags, GLuint bufobj, cl_int *errcode_ret)
flags: See clCreateFromGLBuffer

CL Image Objects > GL Textures [9.6.3]
cl_int clCreateFromGLTexture (cl_context context, cl_mem_flags flags, GLenum texture_target, GLint miplevel, GLuint texture, cl_int *errcode_ret)
flags: See clCreateFromGLTexture

d3d9 Media Surface Sharing [9.9]
These functions require the extension cl_khr_d9_media_sharing. The associated header file is cl_d9_media_sharing.h.

c_int clGetDeviceFromD3DXMediaAdapterKHR (cl_context context, cl_mem_flags flags, cl_d3d9_media_adapter_set khr_media_adapter_set, cl_uint num_entries, cl_device_id *devices, cl_int *num_devices)
media_adapter_type: CL_ADAPTER_ID3D9, ID3DX9, DXGIKHR_media_adapter_set: CL_ALL, PREFERRED_DEVICES - FOR_D3DX9_MEDIA_ADAPTER_KHR

c_mem clCreateFromD3DXMediaSurfaceKHR (cl_context context, cl_mem_flags flags, cl_d3d9_media_adapter_type_khr media_adapter_type, void *media_adapters, size_t *param_value_size, void *param_value, size_t *param_value_size_ret)

cl_int clEnqueueQueryAcquire, ReleaseDX9MediaSurfacesKHR (cl_command_queue command_queue, cl_uint num_objects, cl_mem *mem_objects, cl_event *event_wait_list, cl_event *event)

Direct3D 11 Sharing [9.10.7.3 - 9.10.7.6]
These functions require the cl_khr_d3d11 Sharing extension. Associated header file is cl_d3d11.h.

c_int clGetDeviceFromD3D11KHR (cl_platform_id platform, cl_d3d11_device_share_source khr_device_share_source, void *device, void *adapters, size_t *param_value_size, void *param_value, size_t *param_value_size_ret)

cl_mem clCreateFromD3D11BufferKHR (cl_context context, cl_mem_flags flags, ID3D11Buffer *buffer, cl_event *errcode_ret)
flags: See clCreateFromD3D11Buffer

c_int clEnqueueCreateFromD3D11ObjectsKHR (cl_context context, cl_mem_flags flags, ID3D11Texture2D *resource, UINT subresource, cl_int *errcode_ret)
flags: See clCreateFromD3D11Buffer

c_mem clCreateFromD3D11Texture2DKHR (cl_context context, cl_mem_flags flags, ID3D11Texture2D *resource, UINT subresource, cl_int *errcode_ret)
flags: See clCreateFromD3D11Buffer

c_mem clCreateFromD3D11Objects2DKHR (cl_context context, cl_mem_flags flags, ID3D11Texture2D *resource, UINT subresource, cl_int *errcode_ret)
flags: See clCreateFromD3D11Buffer

c_mem clCreateFromD3D11Objects2DKHR (cl_context context, cl_mem_flags flags, ID3D11Texture2D *resource, UINT subresource, cl_int *errcode_ret)
flags: See clCreateFromD3D11Buffer

Create CL Event Objects from EGL [9.20]
This function requires the extension cl_khr_egl_event.

c_int clEnqueueCreateFromD3D11ObjectKHR (cl_context context, cl_mem_flags flags, ID3D11ObjectKHR *object, cl_int *errcode_ret)
flags: See clCreateFromD3D11Buffer

EGL Interoperability [9.19, 9.20]
Create CL Image Objects from EGL [9.19]
These functions require the extension cl_khr_egl_image.

c_mem clCreateFromEGLImageKHR (cl_context context, CeglDisplayKHR display, CeglImageKHR image, cl_mem_flags flags, const egl_image_propertiesKHR *properties, cl_int *errcode_ret)

Direct3D 10 Sharing [9.8.7]
These functions require the cl_khr_d3d10 Sharing extension. The associated header file is cl_d3d10.h.

c_int clGetDeviceFromD3D10KHR (cl_platform_id platform, cl_d3d10_device_share_source khr_device_share_source, void *device, cl_int *errcode_ret)
flags: See clCreateFromD3D10Buffer

c_mem clCreateFromD3D10BufferKHR (cl_context context, cl_mem_flags flags, ID3D10Buffer *resource, cl_int *errcode_ret)
flags: See clCreateFromD3D10Buffer

c_mem clCreateFromD3D10Texture2DKHR (cl_context context, cl_mem_flags flags, ID3D10Texture2D *resource, UINT subresource, cl_int *errcode_ret)
flags: See clCreateFromD3D10Buffer

c_mem clCreateFromD3D10Objects2DKHR (cl_context context, cl_mem_flags flags, ID3D10Objects2D *resource, UINT subresource, cl_int *errcode_ret)
flags: See clCreateFromD3D10Buffer

Create CL Event Objects from EGL [9.20]
This function requires the extension cl_khr_egl_event.

c_int clEnqueueCreateFromEGLSyncKHR (cl_context context, CeglDisplayKHR display, cl_int *errcode_ret)
flags: See clCreateFromD3D10Buffer

OpenCL Extensions Reference

Using OpenCL Extensions [9]
The following extensions extend the OpenCL API. Extensions shown in italics provide core features.
To control an extension: #pragma OPENCL EXTENSION extension_name : (enable | disable)
To test if an extension is supported: clGetPlatformInfo() or clGetDeviceInfo()
To get the address of the extension function: clGetExtensionFunctionAddressForPlatform()
OpenCL Reference Card Index

The following index shows each item included on this card along with the page on which it is described. The color of the row in the table below is the color of the box to which you should refer.

<table>
<thead>
<tr>
<th>Page</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Access Qualifiers</td>
<td>4</td>
<td>Buffer Objects 1-2</td>
</tr>
<tr>
<td>7</td>
<td>Address Space Qualifier Functions</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Aligned attribute qualifiers</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Async Copies and Prefetch</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Atomic Functions</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Attribute Qualifiers</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Barriers</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Blocks</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

The Khronos Group is an industry consortium creating open standards for the authoring and acceleration of parallel computing, graphics and dynamic media on a wide variety of platforms and devices. See www.khronos.org to learn more about the Khronos Group.

Kronos and the Khronos Group logo are registered trademarks of the Khronos Group, and the Khronos OpenCL logo is a trademark of Apple Inc. and is used under license by Khronos.