/dev/world/2010

Your App Universe Down Under

28-29 September
Rydges Melbourne
OpenCL: Fundamentals

Derek Gerstmann
University of Western Australia
Open Computing Language (OpenCL)

The open standard for developing cross-platform, vendor agnostic, parallel programs that run on current and future multi-core processors within workstations, desktops, notebooks and embedded devices.
Course Outline

- Historical Background
- Anatomy of OpenCL
- OpenCL Architecture
- Examples of Using OpenCL
- Questions & Comments
Historical Background
Modern Architectures

Multi-Core CPUs

Massively Multi-Core GPUs
Wheel of Reincarnation

Timeline of Compute-Oriented Technology Milestones for Massively Multi-Core Processors
Need efficient use of all system resources to enable scalable high-performance applications
Motivation for OpenCL

Enable Heterogenous Processing
-- right now on current hardware for real applications
-- drive the future of software & hardware
Design Goals

Enable all compute resources in system
- CPUs, GPUs, and other processors
- Data- and task- parallel compute model

Efficient parallel programming model
- ANSI C99 based kernel language
Design Goals

Low-level abstraction for compute devices

- Avoid specifics of the hardware
- Enable high performance
- Support device independence
Design Goals

Consistent results on all platforms

- Well-defined precision requirements for all floating-point computations

Interoperability with Graphics APIs

- Dedicated support for OpenGL, OpenGL-ES and DirectX
Design Goals

Drive future hardware requirements

• For both consumer-level and high-performance computing applications
• Be prepared for the next spin on the wheel of reincarnation
Anatomy of OpenCL

Language Specification
Platform Specification
Runtime Specification
## Scalar Storage Types

Data Types for the OpenCL Framework

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void</code></td>
<td>incomplete type corresponding to empty set</td>
</tr>
<tr>
<td><code>cl_char</code></td>
<td>8bit signed two’s complement integer value</td>
</tr>
<tr>
<td><code>cl_uchar</code></td>
<td>8bit unsigned integer</td>
</tr>
<tr>
<td><code>cl_short</code></td>
<td>16bit signed two’s complement integer value</td>
</tr>
<tr>
<td><code>cl_ushort</code></td>
<td>16bit unsigned integer</td>
</tr>
<tr>
<td><code>cl_int</code></td>
<td>32bit signed two’s complement integer value</td>
</tr>
<tr>
<td><code>cl_uint</code></td>
<td>32bit unsigned integer</td>
</tr>
<tr>
<td><code>cl_long</code></td>
<td>64bit signed two’s complement integer value</td>
</tr>
<tr>
<td><code>cl_ulong</code></td>
<td>64bit unsigned integer</td>
</tr>
<tr>
<td><code>cl_half</code></td>
<td>half precision floating-point value (IEEE 754-2008)</td>
</tr>
<tr>
<td><code>cl_float</code></td>
<td>full precision floating-point value (IEEE 754)</td>
</tr>
<tr>
<td><code>cl_double</code></td>
<td>double precision floating-point value (IEEE 754) (opt)</td>
</tr>
</tbody>
</table>
Vector Storage Types
Data Types for the OpenCL Framework

N – Supported values of N are 2, 4, 8, 16.

- cl_char\(_N\) – 8bit signed two’s complement integer value
- cl_uchar\(_N\) – 8bit unsigned integer
- cl_short\(_N\) – 16bit signed two’s complement integer value
- cl_ushort\(_N\) – 16bit unsigned integer
- cl_int\(_N\) – 32bit signed two’s complement integer value
- cl_uint\(_N\) – 32bit unsigned integer
- cl_long\(_N\) – 64bit signed two’s complement integer value
- cl_ulong\(_N\) – 64bit unsigned integer value
- cl_half\(_N\) – half precision floating-point value (IEEE 754–2008)
- cl_float\(_N\) – full precision floating-point value (IEEE 754)
- cl_double\(_N\) – double precision floating-point value (IEEE 754)(opt)
Object Types
Data Types for the OpenCL Framework

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cl_platform_id</td>
<td>identifier for a specific platform</td>
</tr>
<tr>
<td>cl_device_id</td>
<td>identifier for a specific compute device</td>
</tr>
<tr>
<td>cl_context</td>
<td>handle for a compute context</td>
</tr>
<tr>
<td>cl_command_queue</td>
<td>handle for a command queue (for a specific compute device)</td>
</tr>
<tr>
<td>cl_mem</td>
<td>handle for a memory resource (managed by the context)</td>
</tr>
<tr>
<td>cl_program</td>
<td>handle for a program resource (library of kernels)</td>
</tr>
<tr>
<td>cl_kernel</td>
<td>handle for a compute kernel (compiled)</td>
</tr>
</tbody>
</table>

All object types are opaque handles
Enables cross-platform compatibility for complex data types

All objects are reference counted and garbage collected
When reference count reaches zero, object is deallocated
Enqueue Command Methods
Conventions for the OpenCL Framework

cl_int clEnqueueMethod(cl_command_queue, /* command queue */, ... /* method specific parameters */, cl_uint /* number of events in wait list */, const cl_event /* event wait list */, cl_event /* returned event */)
Create Object Methods
Conventions for the OpenCL Framework

```c
cl_object clCreateMethod(cl_context /* compute context */,
                        ...
                        /* method specific parameters */,
                        cl_int * /* returned error code */)
```

All object creation methods return an object handle
Method returns an invalid object if creation fails

All object creation methods require a compute context
Objects and resources are managed in a context

All object creation methods optionally return an error code
Returned value identifies any errors that were encountered
Value is set to CL_SUCCESS if object was created successfully
All property methods require an object and property
Enumerated property correspond to named properties (eg CL_DEVICE_NAME)
Method returns CL_SUCCESS if property was returned successfully

All property methods require a pointer and type size
Storage pointer must match property type and large enough for requested type size

All property methods optionally return a type size
Size may be less than requested depending on property request (eg list of values)
Management Methods
Conventions for the OpenCL Framework

All retain / release object methods return an error code
Method returns CL_SUCCESS if operation was successfully

All retain methods increment the object reference count
Reference count determines life span and is set to one upon creation of object

All release methods decrement the object reference count
When reference count reaches zero, object is deallocated

cl_int clRetainObject( cl_object /* handle of object to retain */ )

cl_int clReleaseObject( cl_object /* handle of object to release */ )
OpenCL Language

ANSI ISO C99-based kernel language

• Familiar to developers
• Some extensions & restrictions
• Includes a rich set of built-in functions
• Supports online & offline compilation
__kernel void square(
  __global float* input, __global float* output)
{
  size_t i = get_global_id(0);
  output[i] = input[i] * input[i];
}
OpenCL Language

Well-defined numerical precision

• All math functions have numerical accuracy requirements
• IEEE 754 floating-point rounding w/ limits on max error
OpenCL Platform

Hardware abstraction layer

• Supports a diverse set of resources
• Query, select and setup devices
• Create compute contexts
• Low-level device agnostic
OpenCL Runtime

Management interface for resources

- Query, select and setup devices
- Create and manage compute contexts
- Allocate, read and write memory objects
- Fill and manage command queues
- Synchronise devices and execute work
// Fill our data set with random float values
int count = 1024 * 1024;
for(i = 0; i < count; i++)
    data[i] = rand() / (float)RAND_MAX;

// Connect to a compute device, create a context and a command queue
cl_device_id device;
clGetDeviceIDs(CL_DEVICE_TYPE_GPU, 1, &device, NULL);
cl_context context = clCreateContext(0, 1, &device, NULL, NULL, NULL);
cl_command_queue queue = clCreateCommandQueue(context, device, 0, NULL);

// Create and build a program from our OpenCL-C source code
cl_program program = clCreateProgramWithSource(context, 1, (const char **) &src, NULL, NULL);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// Create a kernel from our program
cl_kernel kernel = clCreateKernel(program, "square", NULL);

// Allocate input and output buffers, and fill the input with data
cl_mem input = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * count, NULL, NULL);
cl_mem output = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * count, NULL, NULL);
clEnqueueWriteBuffer(queue, input, CL_TRUE, 0, sizeof(float) * count, data, 0, NULL, NULL);

// Get the maximum number of work items supported for this kernel on this device
size_t global = count; size_t local = 0;
clGetKernelWorkGroupInfo(kernel, device, CL_KERNEL_WORK_GROUP_SIZE, sizeof(int), &local, NULL);

// Set the arguments to our kernel, and enqueue it for execution
clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &output);
clSetKernelArg(kernel, 2, sizeof(unsigned int), &count);
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global, &local, 0, NULL, NULL);

// Force the command queue to get processed, and wait until all commands are complete
clFinish(queue);

// Read back the results
clEnqueueReadBuffer(queue, output, CL_TRUE, 0, sizeof(float) * count, results, 0, NULL, NULL);

// Validate our results
int correct = 0;
for(i = 0; i < count; i++)
    correct += (results[i] == data[i] * data[i]) ? 1 : 0;

// Print a brief summary detailing the results
printf("Computed '%d/%d' correct values!\n", correct, count);
OpenCL Architecture

- Platform Model
- Execution Model
- Memory Model
- Programming Model
Platform Model

Platform model encapsulates compute resources
Hierarchy of compute units logically grouped together based on locality
Platform Model

One host connected to one or more compute device(s)
Compute device could be a CPU, GPU, or other processor.
Each compute device composed of one or more units
A compute unit may be a core, array multi-processor, streaming multiprocessor, etc.
Each unit has one or more processing element(s)
Processing elements execute instructions together (eg. SIMD or SPMD)
Querying Platform Info and Devices [4.1, 4.2]

```
c_int clGetPlatformIDs (cl_uint num_entries,  
  cl_platform_id *platforms, cl_uint *num_platforms)

c_int clGetPlatformInfo (cl_platform_id platform,  
  cl_platform_info param_name, size_t param_value_size,  
  void *param_value, size_t *param_value_size_ret)
```

```
param_name: CL_PLATFORM {PROFILE, VERSION},  
CL_PLATFORM {NAME, VENDOR, EXTENSIONS}
```

```
c_int clGetDeviceIDs (cl_platform_id platform,  
  cl_device_type device_type, cl_uint num_entries,  
  cl_device_id *devices, cl_uint *num_devices)
```

```
device_type: CL_DEVICE_TYPE {CPU, GPU},  
CL_DEVICE_TYPE {ACCELERATOR, DEFAULT, ALL}
```
void *param_value, size_t *param_value_size_ret)

param_name: CL_DEVICE_TYPE,
  CL_DEVICE_VENDOR_ID,
  CL_DEVICE_MAX_COMPUTE_UNITS,
  CL_DEVICE_MAX_WORK_ITEM {DIMENSIONS, SIZES},
  CL_DEVICE_MAX_WORK_GROUP_SIZE,
  CL_DEVICE {NATIVE, PREFERRED} VECTOR_WIDTH_CHAR,
  CL_DEVICE {NATIVE, PREFERRED} VECTOR_WIDTH_SHORT,
  CL_DEVICE {NATIVE, PREFERRED} VECTOR_WIDTH_INT,
  CL_DEVICE {NATIVE, PREFERRED} VECTOR_WIDTH_LONG,
  CL_DEVICE {NATIVE, PREFERRED} VECTOR_WIDTH_FLOAT,
  CL_DEVICE {NATIVE, PREFERRED} VECTOR_WIDTH_DOUBLE,
  CL_DEVICE_MAX_CLOCK_FREQUENCY,
  CL_DEVICE_ADDRESS_BITS,
  CL_DEVICE_MAX_MEM_ALLOC_SIZE,
  CL_DEVICE_IMAGE_SUPPORT,
  CL_DEVICE_MAX {READ, WRITE} IMAGE_ARGS,
  CL_DEVICE_IMAGE2D_MAX {WIDTH, HEIGHT},
  CL_DEVICE_IMAGE3D_MAX {WIDTH, HEIGHT, DEPTH},
  CL_DEVICE_MAX_SAMPLERS,
  CL_DEVICE_MAX_PARAMETER_SIZE,
  CL_DEVICE_MEM_BASE_ADDR_ALIGN,
  CL_DEVICE_MIN_DATA_TYPE_ALIGN_SIZE,
  CL_DEVICE_SINGLE_FP_CONFIG,
  CL_DEVICE_GLOBAL_MEM_CACHE {TYPE, SIZE},
  CL_DEVICE_GLOBAL_MEM_CACHeline_SIZE,
  CL_DEVICE_GLOBAL_MEM_SIZE,
  CL_DEVICE_MAX_CONSTANT {BUFFER_SIZE, ARGS},
  CL_DEVICE_LOCAL_MEM {TYPE, SIZE},
  CL_DEVICE_ERROR_CORRECTION_SUPPORT,
  CL_DEVICE_PROFILING_TIMER_RESOLUTION,
  CL_DEVICE_ENDIAN_LITTLE,
  CL_DEVICE_AVAILABLE,
  CL_DEVICE_COMPILER_AVAILABLE,
  CL_DEVICE_EXECUTION_CAPABILITIES,
  CL_DEVICE_QUEUE_PROPERTIES,
  CL_DEVICE {NAME, VENDOR, PROFILE, EXTENSIONS},
  CL_DEVICE_HOST_UNIFIED_MEMORY,
  CL_DEVICE_OPENCL_C_VERSION,
  CL_DEVICE_VERSION,
  CL_DRIVER_VERSION, CL_DEVICE_PLATFORM
Execution Model

Application needs access to compute resources in system
Execution model defines the interaction between the host application and a device
Context encapsulates resources into a logical grouping

Think of this as a container of compute resources for your application
Execution Model

Memory, programs, and queues are owned by the context
Provides a container that localises allocations into a central store
Also, enables transparent sharing with OpenGL / OpenGL-ES / DirectX
The OpenCL Platform Layer

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

**Contexts** [4.3]

```c
cl_context clCreateContext (const cl_context_properties *properties, cl_uint num_devices,
    const cl_device_id *devices, void (CL_CALLBACK* pfn_notify)
    (const char *errinfo, const void *private_info,
     size_t cb, void *user_data),
    void *user_data, cl_int *errcode_ret)
```

**Properties:** CL_CONTEXT_PLATFORM, CL_GL_CONTEXT_KHR,
CL_CGL_SHAREGROUP_KHR, CL_{EGL, GLX}_DISPLAY_KHR,
CL_WGL_HDC_KHR

```c
cl_context clCreateContextFromType (const cl_context_properties *properties,
    cl_device_type device_type, void (CL_CALLBACK* pfn_notify)
    (const char *errinfo, const void *private_info, size_t cb,
     void *user_data),
    void *user_data, cl_int *errcode_ret)
```

**Properties:** See clCreateContext

```c
cl_int clRetainContext (cl_context context)
```

```c
cl_int clReleaseContext (cl_context context)
```

```c
cl_int clGetContextInfo (cl_context context,
    cl_context_info param_name, size_t param_value_size,
    void *param_value, size_t *param_value_size_ret)
```

**Param_name:** CL_CONTEXT_REFERENCE_COUNT,
CL_CONTEXT_DEVICES, CL_CONTEXT_PROPERTIES, CL_CONTEXT_NUM_DEVICES

---

©2010 Khronos Group - Rev. 0610

OpenCL API 1.1 Quick Reference Card - Page 1
Memory allocations can be raw bytes or formatted images

Buffers are basically the same as a malloc in C99 -- contiguous allocation of bytes

Images provide a direct path to texture hardware and support sampling, filtering, etc.
Execution Model

Memory is accessible for all devices in context
However, synchronisation is required to get expected results!
**Buffer Objects**

Elements of a buffer object can be a scalar or vector data type or a user-defined structure. Elements are stored sequentially and are accessed using a pointer by a kernel executing on a device. Data is stored in the same format as it is accessed by the kernel.

**Create Buffer Objects** [5.2.1]

```c
cl_mem clCreateBuffer (cl_context context,
    cl_mem_flags flags, size_t size, void *host_ptr,
    cl_int *errcode_ret)
```

```c
cl_mem clCreateSubBuffer (cl_mem buffer,
    cl_mem_flags flags,
    cl_buffer_create_type buffer_create_type,
    const void *buffer_create_info, cl_int *errcode_ret)
```

**flags** for clCreateBuffer and clCreateSubBuffer:
- CL_MEM_READ_WRITE
- CL_MEM_WRITE_ONLY
- CL_MEM_USE_ALLOC_COPY

**Read, Write, Copy Buffer Objects** [5.2.2]

```c
cl_int clEnqueueReadBuffer (cl_command_queue command_queue, cl_mem buffer,
    cl_bool blocking_read, size_t offset, size_t cb,
    void *ptr, cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list, cl_event *event)
```

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

```c
cl_int clEnqueueCopyBuffer (cl_command_queue command_queue, cl_mem src_buffer, cl_mem dst_buffer, size_t src_offset, size_t dst_offset, size_t cb,
    cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list, cl_event *event)
```

```c
cl_int clEnqueueCopyBufferRect (cl_command_queue command_queue, cl_mem buffer,
    cl_bool blocking_read, const size_t buffer_origin[3],
    const size_t host_origin[3], const size_t region[3],
    size_t buffer_row_pitch, size_t buffer_slice_pitch,
    size_t host_row_pitch, size_t host_slice_pitch,
    void *ptr, cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list, cl_event *event)
```

```c
cl_int clEnqueueWriteBufferRect (cl_command_queue command_queue, cl_mem buffer,
    cl_bool blocking_write, const size_t buffer_origin[3],
    const size_t host_origin[3], const size_t region[3],
    size_t buffer_row_pitch, size_t buffer_slice_pitch,
    size_t host_row_pitch, size_t host_slice_pitch,
    void *ptr, cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list, cl_event *event)
```
Image Objects

Create Image Objects [5.3.1]

```
cl_int clCreateImage2D (cl_context context,
    cl_mem_flags flags, const cl_image_format *image_format,
    size_t image_width, size_t image_height,
    size_t image_row_pitch, void *host_ptr, cl_int *errcode_ret)
```

flags: See clCreateImage2D

Query List of Supported Image Formats [5.3.2]

```
cl_int clGetSupportedImageFormats (cl_context context,
    cl_mem_flags flags, cl_mem_object_type image_type,
    cl_uint num_entries, cl_image_format *image_formats,
    cl_uint *num_image_formats)
```

flags: See clCreateImage2D

Copy Between Image, Buffer Objects [5.3.4]

```
cl_int clEnqueueCopyImageToBuffer ( 
    cl_command_queue command_queue, cl_mem src_image,
    cl_mem dst_buffer, const size_t src-origin[3],
    const size_t region[3], size_t dst_offset,
    cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list, cl_event *event)
```

```
cl_int clEnqueueCopyBufferToImage ( 
    cl_command_queue command_queue, cl_mem src_buffer,
    cl_mem dst_image, size_t src_offset,
    const size_t dst-origin[3], const size_t region[3],
    cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list, cl_event *event)
```

Read, Write, Copy Image Objects [5.3.3]

```
cl_int clEnqueueReadImage ( 
    cl_command_queue command_queue, cl_mem image,
    cl_bool blocking_read, const size_t origin[3],
    const size_t region[3], size_t row_pitch,
    size_t slice_pitch, void *ptr,
    cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list, cl_event *event)
```

```
cl_int clEnqueueWriteImage ( 
    cl_command_queue command_queue,
    cl_mem image, cl_bool blocking_write,
    const size_t origin[3], const size_t region[3],
    size_t input_row_pitch, size_t input_slice_pitch,
    const void *ptr, cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list, cl_event *event)
```

```
cl_int clEnqueueCopyImage ( 
    cl_command_queue command_queue,
    cl_mem src_image, cl_mem dst_image,
    const size_t src-origin[3], const size_t dst-origin[3],
    const size_t region[3], cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list, cl_event *event)
```

Copy Image Objects [5.3.6]

```
cl_int clGetMemObjectInfo (cl_mem memobj,
    cl_mem_info param_name, size_t param_value_size,
    void *param_value, size_t *param_value_size_ret)
```

```
cl_int clGetImageInfo (cl_mem image,
    cl_image_info param_name, size_t param_value_size,
    void *param_value, size_t *param_value_size_ret)
```

Rydges Melbourne
Execution Model

Application uses context for moving data to/from host
All data transfers between host and device must be done explicitly!
Execution Model

Programs contain compiled kernel objects
Each program must be compiled for every type of device and/or instruction set
Program Objects

Create Program Objects [5.6.1]

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

cl_program cCreateProgramWithBinary (  
    cl_context context, cl_uint num_devices,
    const cl_device_id *device_list, const size_t *lengths,
    const unsigned char **binaries, cl_int *binary_status,
    cl_int *errcode_ret)

cl_int cRetainProgram (cl_program program)

cl_int cReleaseProgram (cl_program program)

Build Program Executable [5.6.2]

cl_int cBuildProgram (cl_program program,
    cl_uint num_devices, const cl_device_id *device_list,
    const char *options, void (CL_CALLBACK *pfn_notify) (cl_program program, void *user_data),
    void *user_data)

Build Options [5.6.3]

Preprocessor: (-D processed in order listed in clBuildProgram)

-D name   -D name-definition   -I dir

Optimization options:

-cl-opt-disable
-cl-mad-enabled
-cl-finite-math-only
-cl-unsafe-math-optimizations
-cl-strict-aliasing
-cl-no-signed-zeros
-cl-fast-relaxed-math
Kernel and Event Objects

Create Kernel Objects [5.7.1]

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

cl_int 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 Args. & Object Queries [5.7.2, 5.7.3]

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

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

param_name: CL_KERNEL_FUNCTION_NAME, 
CL_KERNEL_NUM_ARGS, CL_KERNEL_REFERENCE_COUNT, 
CL_KERNEL_CONTEXT, CL_KERNEL_PROGRAM

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

param_name: CL_KERNEL_WORK_GROUP_SIZE, 
CL_KERNEL_COMPILE_WORK_GROUP_SIZE, 
CL_KERNEL｛LOCAL, PRIVATE｝MEM_SIZE, 
CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE

Reserved Data Types [1.1.4]

Other Built-in Data Types [1.1.3]

Built-in Vector Data Types [1.2.3]

Built-in Scalar Data Types [1.1.3]

Support Data Types

OpenCL API 1.1 Quick Reference Card - Page 2

©2010 Khronos Group · Rev. 0610

www.khronos.org/opencl

AUC

/dev/world/2010

Rydges Melbourne
Command queues provide asynchronous execution

Enqueue commands onto a command queue to do work *(sometime later)*
Execution Model

Queues can be processed in-order or out-of-order

In-order queues are default -- commands run when preceding entries are complete
Out-of-order queues require explicit synchronisation via events and wait-lists
Queues are directly associated with a specific device
Submit commands to the associated queue to do work on a particular device
**Command Queues** [5.1]

```c
cl_command_queue clCreateCommandQueue ( 
    cl_context context, cl_device_id device, 
    cl_command_queue_properties properties, 
    cl_int *errcode_ret)
```

**properties:** CL_QUEUE_PROFILING_ENABLE, CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE

```c
cl_int clRetainCommandQueue ( 
    cl_command_queue command_queue)
```

```c
cl_int clReleaseCommandQueue ( 
    cl_command_queue command_queue)
```

```c
cl_int clGetCommandQueueInfo ( 
    cl_command_queue command_queue, 
    cl_command_queue_info param_name, 
    size_t param_value_size, void *param_value, 
    size_t *param_value_size_ret)
```

**param_name:** CL_QUEUE_CONTEXT, CL_QUEUE_DEVICE, CL_QUEUE_REFERENCE_COUNT, CL_QUEUE_PROPERTIES
Execution Model

```c
int main(...) {
  ...
}
```

clEnqueueWriteBuffer(CPU_QUEUE, ...);
int main(...) {
  ...
  }

clEnqueueWriteBuffer(CPU_QUEUE, ...,);
clEnqueueNDRangeKernel(GPU_QUEUE, ...,);
int main(...) {
  ...
}

clEnqueueWriteBuffer(CPU_QUEUE, ...);
clEnqueueNDRangeKernel(CPU_QUEUE, ...);
clEnqueueNDRangeKernel(CPU_QUEUE, ...);
Execution Model

```c
int main(...) {
  ...
  }
```

cEnqueueWriteBuffer(CPU_QUEUE, ...);
cEnqueueNDRangeKernel(CPU_QUEUE, ...);
cEnqueueNDRangeKernel(CPU_QUEUE, ...);
cFinish(GPU_QUEUE, ...);

CPU

GPU
Execution Model

int main(...) {
  ...
}

clEnqueueWriteBuffer(CPU_QUEUE, ...);
clEnqueueNDRangeKernel(CPU_QUEUE, ...);
clEnqueueNDRangeKernel(CPU_QUEUE, ...);
clFinish(GPU_QUEUE, ...);
clEnqueueWriteBuffer(CPU_QUEUE, ...);
__kernel void square(
    __global float* input, __global float* output)
{
    size_t i = get_global_id(0);
    output[i] = input[i] * input[i];
}
Compute Kernels

```c
__kernel void square(
    __global float* input, __global float* output)
{
    size_t i = get_global_id(0);
    output[i] = input[i] * input[i];
}
```

**Compute kernels are just like exported dynlib methods**

Required to have a `void` return type, and the `__kernel` qualifier

Compiled to native code for a specific device and stored in a program object
Host application invokes a kernel over an index space.

Index space is an N-dimensional range (where N is 1, 2, or 3).
Work Distribution

Global range is executed over local work-groups
Each work-group has a collection of work-items addressable via a globally unique id
Work Distribution

Work-items share resources in a work-group

Mitigates communication costs for synchronisation -- group-wise barriers are cheap!
Work Distribution

Kernels are executed across a domain of work-items
Global dimensions define range of computation

Work-items are logically organised into work-groups
Local dimensions define size of work-group

Work-groups are executed in parallel
Work-items in a work-group can communicate to each other
Must use synchronisation to coordinate memory access
Memory Model

Address space hierarchy
All address spaces are distinct and cannot be intermixed
**Memory Model**

**Private:**
private to a single work-item

**Local:**
shared within a work-group

**Global/Constant:**
globally accessible by any work-item

**Host:**
accessible from host application
Data Transfer
All data movement between address spaces must be done explicitly

Applications must move data to/from Host / Global / Local / Private
Data Consistency

Shared memory model uses relaxed consistency
State of memory visible to a work-item is not guaranteed to be consistent among all work-items at all times
If consistency is needed, synchronisation is required

Synchronisation of memory must be done explicitly across all levels of the memory hierarchy in order to get the same data to be visible at any given time.
Compute Kernels

```c
__kernel void square(
  __global float* input, __global float* output)
{
  size_t i = get_global_id(0);
  output[i] = input[i] * input[i];
}
```

Built-in methods provide access to index space addresses

Use the `get_global_id()` built-in for globally unique addresses
Use the `get_group_id()` for the logical group id spanning the ND-range
Use the `get_local_id()` for the local work-item address within a work-group
Work-Item Built-in Functions [6.11.1]  

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint get_work_dim()</code></td>
<td>Num. of dimensions in use</td>
</tr>
<tr>
<td><code>size_t get_global_size (uint D)</code></td>
<td>Num. of global work-items</td>
</tr>
<tr>
<td><code>size_t get_global_id (uint D)</code></td>
<td>Global work-item ID</td>
</tr>
<tr>
<td><code>size_t get_local_size (uint D)</code></td>
<td>Num. of local work-items</td>
</tr>
<tr>
<td><code>size_t get_local_id (uint D)</code></td>
<td>Local work-item ID</td>
</tr>
<tr>
<td><code>size_t get_num_groups (uint D)</code></td>
<td>Num. of work-groups</td>
</tr>
<tr>
<td><code>size_t get_group_id (uint D)</code></td>
<td>Returns the work-group ID</td>
</tr>
<tr>
<td><code>size_t get_global_offset (uint D)</code></td>
<td>Returns global offset</td>
</tr>
</tbody>
</table>

Math Built-in Functions [6.13.2]  

[Diagram of mathematical functions with annotations]
Common Built-in Functions [6.11.4]

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T clamp (T x, T min, T max)</td>
<td>Clamp x to range given by min, max</td>
</tr>
<tr>
<td>floatn clamp (floatn x, floatn min, floatn max)</td>
<td></td>
</tr>
<tr>
<td>doublen clamp (doublen x, doublen min, doublen max)</td>
<td></td>
</tr>
<tr>
<td>halfn clamp (halfn x, halfn min, halfn max)</td>
<td></td>
</tr>
<tr>
<td>T degrees (T radians)</td>
<td>radians to degrees</td>
</tr>
<tr>
<td>T max (T x, T y)</td>
<td>Max of x and y</td>
</tr>
<tr>
<td>floatn max (floatn x, floatn y)</td>
<td></td>
</tr>
<tr>
<td>doublen max (doublen x, doublen y)</td>
<td></td>
</tr>
<tr>
<td>halfn max (halfn x, halfn y)</td>
<td></td>
</tr>
<tr>
<td>T min (T x, T y)</td>
<td>Min of x and y</td>
</tr>
<tr>
<td>floatn min (floatn x, floatn y)</td>
<td></td>
</tr>
<tr>
<td>doublen min (doublen x, doublen y)</td>
<td></td>
</tr>
<tr>
<td>halfn min (halfn x, halfn y)</td>
<td></td>
</tr>
<tr>
<td>T mix (T x, T y, T a)</td>
<td>Linear blend of x and y</td>
</tr>
<tr>
<td>floatn mix (floatn x, floatn y, floatn a)</td>
<td></td>
</tr>
<tr>
<td>doublen mix (doublen x, doublen y, doublen a)</td>
<td></td>
</tr>
<tr>
<td>halfn mix (halfn x, halfn y, halfn a)</td>
<td></td>
</tr>
<tr>
<td>T radians (T degrees)</td>
<td>degrees to radians</td>
</tr>
<tr>
<td>T step (T edge, T x)</td>
<td>0.0 if x &lt; edge, else 1.0</td>
</tr>
<tr>
<td>floatn step (floatn edge, floatn x)</td>
<td></td>
</tr>
<tr>
<td>doublen step (doublen edge, doublen x)</td>
<td></td>
</tr>
<tr>
<td>halfn step (halfn edge, halfn x)</td>
<td></td>
</tr>
<tr>
<td>T smoothstep (T edge0, T edge1, T x)</td>
<td>Step and interpolate</td>
</tr>
<tr>
<td>floatn smoothstep (floatn edge0, floatn edge1, floatn x)</td>
<td></td>
</tr>
<tr>
<td>doublen smoothstep (doublen edge0, doublen edge1, doublen x)</td>
<td></td>
</tr>
<tr>
<td>halfn smoothstep (halfn edge0, halfn edge1, halfn x)</td>
<td></td>
</tr>
<tr>
<td>T sign (T x)</td>
<td>Sign of x</td>
</tr>
</tbody>
</table>
**Integer Built-in Functions [6.11.3]**

* T is type char, charn, uchar, ucharn, short, shornt, ushort, ushorts, int, intnn, uint, uintnn, long, longn, ulong, or ulogn.

* U is the unsigned version of T. S is the scalar version of T.

### Functions

- **T sub_sat (Tx, Ty)**
  
  - x - y and saturates the result

- **For upsample, scalar types are permitted for the vector types below.**

  - **shortn upsample (charn hi, ucharn lo)**
    
    - result[i] = ((shortn)hi[i] << 8) | lo[i]

  - **ushortn upsample (charn hi, ucharn lo)**
    
    - result[i] = ((ushortn)hi[i] << 8) | lo[i]

  - **intn upsample (shortn hi, ushorts n lo)**
    
    - result[i] = ((intn)hi[i] << 16) | lo[i]

  - **uintn upsample (ushortn hi, ushorts n lo)**
    
    - result[i] = ((uintn)hi[i] << 16) | lo[i]

  - **longn upsample (intn hi, uintnn lo)**
    
    - result[i] = ((longn)hi[i] << 32) | lo[i]

  - **ulongn upsample (uintn hi, uintnn lo)**
    
    - result[i] = ((ulongn)hi[i] << 32) | lo[i]

The following fast integer functions optimize the performance of kernels. In these functions, T is type int, int2, int3, int4, int8, int16, uint, uint2, uint4, uint8 or uint16.

- **T mad24 (T a, T b, T c)**
  
  - Multiply 24-bit int. values a, b, add 32-bit int. result to 32-bit int. c

- **T mul24 (T a, T b)**
  
  - Multiply 24-bit int. values a and b
### Math Built-in Functions [6.11.2]

is type float or double (or optionally double, double, or half). Qualified, unqualified, and unqualified must be scalar when T is scalar. Q is qualifier _global, _local, or _private. HN indicates that half and single variants are available by prepending "half_" or "native_" to function name. Prototypes shown in purple are half_ and native_ only. Optional extensions enable double, half, and half only.

<table>
<thead>
<tr>
<th>Function (T)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T acos (T)</td>
<td>Arc cosine</td>
</tr>
<tr>
<td>T acosh (T)</td>
<td>Inverse hyperbolic cosine</td>
</tr>
<tr>
<td>T acosi (T x)</td>
<td>acos (x) / π</td>
</tr>
<tr>
<td>T asin (T)</td>
<td>Arc sine</td>
</tr>
<tr>
<td>T asinh (T)</td>
<td>Inverse hyperbolic sine</td>
</tr>
<tr>
<td>T asinpi (T x)</td>
<td>asin (x) / π</td>
</tr>
<tr>
<td>T atan (T y, over x)</td>
<td>Arc tangent</td>
</tr>
<tr>
<td>T atan2 (T y, T x)</td>
<td>Arc tangent of y / x</td>
</tr>
<tr>
<td>T atanh (T x)</td>
<td>Hyperbolic arc tangent</td>
</tr>
<tr>
<td>T atanpii (T x)</td>
<td>atan (x) / π</td>
</tr>
<tr>
<td>T bct (T)</td>
<td>Cube root</td>
</tr>
<tr>
<td>T ceil (T x)</td>
<td>Round to integer toward + infinity</td>
</tr>
<tr>
<td>T cphase (T x, T y)</td>
<td>x with sign changed to sign of y</td>
</tr>
<tr>
<td>T cos (T)</td>
<td>Cosine</td>
</tr>
<tr>
<td>T cosh (T)</td>
<td>Hyperbolic cosine</td>
</tr>
<tr>
<td>T cospi (T x)</td>
<td>cos (π x)</td>
</tr>
<tr>
<td>T half_divide (T x, T y)</td>
<td>x / y (T may be float or double)</td>
</tr>
<tr>
<td>T native_divide (T x, T y)</td>
<td>T may be float or double)</td>
</tr>
<tr>
<td>T erfc (T)</td>
<td>Complementary error function</td>
</tr>
<tr>
<td>T erf (T)</td>
<td>Error function of T</td>
</tr>
<tr>
<td>T exp (T x)</td>
<td>Exponential base e</td>
</tr>
<tr>
<td>T exp2 (T)</td>
<td>Exponential base 2</td>
</tr>
<tr>
<td>T exp10 (T)</td>
<td>Exponential base 10</td>
</tr>
<tr>
<td>T expm1 (T x)</td>
<td>e^(x - 1.0)</td>
</tr>
<tr>
<td>T fabs (T)</td>
<td>Absolute value</td>
</tr>
<tr>
<td>T fdim (T x, T y)</td>
<td>&quot;Positive difference&quot; between x and y</td>
</tr>
<tr>
<td>T floor (T)</td>
<td>Round to integer toward - infinity</td>
</tr>
<tr>
<td>T fma (T a, T b, T c)</td>
<td>Multiply and add, then round</td>
</tr>
<tr>
<td>T fmax (T x, T y)</td>
<td>Return if x &lt; y, otherwise it returns x</td>
</tr>
<tr>
<td>T fmin (T x, T y)</td>
<td>Return if y &lt; x, otherwise it returns x</td>
</tr>
<tr>
<td>T fmod (T x, T y)</td>
<td>Modulus. Returns x - y * trunc (x / y)</td>
</tr>
<tr>
<td>T fract (T x, T Q iptr)</td>
<td>Fractional value in x</td>
</tr>
<tr>
<td>T frexp (T x, T Q int &amp; exp)</td>
<td>Extract mantissa and exponential</td>
</tr>
<tr>
<td>T hypot (T x, T y)</td>
<td>Square root of x^2 + y^2</td>
</tr>
<tr>
<td>T int (T)</td>
<td>Return exponential as an integer value</td>
</tr>
<tr>
<td>T idexp (T x, T int n)</td>
<td>x * 2^n</td>
</tr>
<tr>
<td>T ifgamma (T x)</td>
<td>Log gamma function</td>
</tr>
<tr>
<td>T igamma (T x, T Q int *sign)</td>
<td></td>
</tr>
<tr>
<td>T log (T)</td>
<td>Natural logarithm</td>
</tr>
<tr>
<td>T log2 (T)</td>
<td>Base 2 logarithm</td>
</tr>
<tr>
<td>T log10 (T)</td>
<td>Base 10 logarithm</td>
</tr>
<tr>
<td>T log1p (T x)</td>
<td>In (1.0 + x)</td>
</tr>
<tr>
<td>T logb (T x)</td>
<td>Exponent of x</td>
</tr>
<tr>
<td>T mad (T a, T b, T c)</td>
<td>Approximates a * b + c</td>
</tr>
<tr>
<td>T maxmag (T x, T y)</td>
<td>Maximum magnitude of x and y</td>
</tr>
<tr>
<td>T minmag (T x, T y)</td>
<td>Minimum magnitude of x and y</td>
</tr>
<tr>
<td>T mod (T x, T Q iptr)</td>
<td>Decompose a floating-point number</td>
</tr>
<tr>
<td>T pow (T x, T int y)</td>
<td>Compute x to the power of y (x^y)</td>
</tr>
<tr>
<td>T powr (T x, T int n)</td>
<td>Compute x^n, where n is an integer</td>
</tr>
<tr>
<td>T powr (T x, T y)</td>
<td>Next representable floating-point value following x in the direction of</td>
</tr>
<tr>
<td>T remquo (T x, T y)</td>
<td>Floating point remainder</td>
</tr>
<tr>
<td>T remquo (T x, T y, T Q int * quo)</td>
<td>Floating point remainder and quotient</td>
</tr>
<tr>
<td>T round (T x)</td>
<td>Integer value nearest to x rounding</td>
</tr>
<tr>
<td>T rsqrt (T)</td>
<td>Inverse square root</td>
</tr>
<tr>
<td>T sin (T)</td>
<td>Sine</td>
</tr>
<tr>
<td>T sincos (T x, T Q T *cosval)</td>
<td>Sine and cosine of x</td>
</tr>
<tr>
<td>T sinh (T)</td>
<td>Hyperbolic sine</td>
</tr>
<tr>
<td>T sinpi (T x)</td>
<td>sin (π x)</td>
</tr>
<tr>
<td>T sqrt (T)</td>
<td>Square root</td>
</tr>
<tr>
<td>T tan (T)</td>
<td>Tangent</td>
</tr>
<tr>
<td>T tanh (T)</td>
<td>Hyperbolic tangent</td>
</tr>
<tr>
<td>T tanpi (T x)</td>
<td>tan (π x)</td>
</tr>
<tr>
<td>T tgamma (T x)</td>
<td>Hyperbolic sine</td>
</tr>
<tr>
<td>T trunc (T)</td>
<td>Round to integer toward zero</td>
</tr>
</tbody>
</table>
### Relational Built-in Functions [6.11.6]

$T$ is type `float`, `floatn`, `char`, `charn`, `uchar`, `uchar`, `short`, `shortn`, `ushort`, `ushortn`, `int`, `intn`, `uint`, `uintn`, `long`, `longn`, `ulong`, or `ulongn` (and optionally double, `doublen`). $S$ is type `char`, `charn`, `short`, `shortn`, `int`, `intn`, `long`, or `longn`. $U$ is type `uchar`, `uchar`, `ushort`, `ushortn`, `uint`, `uintn`, `ulong`, or `ulongn`. Optional extensions enable double, `doublen`, and `halfn` types.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Test Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int isequal(float x, float y)</code></td>
<td>Compare of $x == y$</td>
<td>Test for +ve or -ve infinity</td>
</tr>
<tr>
<td><code>intn isequal(floatn x, floatn y)</code></td>
<td>Compare of $x == y$</td>
<td>Test for a NaN</td>
</tr>
<tr>
<td><code>int isnotequal(float x, float y)</code></td>
<td>Compare of $x != y$</td>
<td>Test for a normal value</td>
</tr>
<tr>
<td><code>intn isnotequal(floatn x, floatn y)</code></td>
<td>Compare of $x != y$</td>
<td>Test if arguments are ordered</td>
</tr>
<tr>
<td><code>int isgreater(float x, float y)</code></td>
<td>Compare of $x &gt; y$</td>
<td>Test if arguments are unordered</td>
</tr>
<tr>
<td><code>intn isgreater(floatn x, floatn y)</code></td>
<td>Compare of $x &gt; y$</td>
<td>Test if arguments are unordered</td>
</tr>
<tr>
<td><code>int isgreaterequal(float x, float y)</code></td>
<td>Compare of $x &gt;= y$</td>
<td>Test if arguments are unordered</td>
</tr>
<tr>
<td><code>intn isgreaterequal(floatn x, floatn y)</code></td>
<td>Compare of $x &gt;= y$</td>
<td>Test if arguments are unordered</td>
</tr>
<tr>
<td><code>int isless(float x, float y)</code></td>
<td>Compare of $x &lt; y$</td>
<td>Test for sign bit</td>
</tr>
<tr>
<td><code>intn isless(floatn x, floatn y)</code></td>
<td>Compare of $x &lt; y$</td>
<td>Test for sign bit</td>
</tr>
<tr>
<td><code>int islesss(float x, double y)</code></td>
<td>Compare of $x &lt; y$</td>
<td>Test for sign bit</td>
</tr>
<tr>
<td><code>intn islesss(floatn x, doublen y)</code></td>
<td>Compare of $x &lt; y$</td>
<td>Test for sign bit</td>
</tr>
<tr>
<td><code>int isless(floatn x, floatn y)</code></td>
<td>Compare of $x &lt; y$</td>
<td>Test for sign bit</td>
</tr>
</tbody>
</table>
Vector Data Load/Store Functions [6.11.7]

Q is an Address Space Qualifier listed in 6.5 unless otherwise noted. R defaults to the current rounding mode, or is one of the Rounding Modes listed in 6.2.3.2. T is type char, uchar, short, ushort, int, uint, long, ulong, half, or float (or optionally double). Tn refers to the vector form of type T. Optional extensions enable the double, doublen, half, and halfn types.

\[ Tn\ vloadn(\text{size\_t\ offset},\text{const}\ Q\ T*p) \]
Read vector data from memory

\[ \text{void\ vstore\_half}(Tn\ data,\text{size\_t\ offset},Q\ \text{half\ *p}) \]
Write vector data to memory

\[ \text{float\ vload\_half}(\text{size\_t\ offset},\text{const}\ Q\ \text{half\ *p}) \]
Read a half from memory

\[ \text{floatn\ vload\_halfn}(\text{size\_t\ offset},\text{const}\ Q\ \text{half\ *p}) \]
Read multiple halves from memory

\[ \text{void\ vstore\_half}(\text{float\ data},\text{size\_t\ offset},Q\ \text{half\ *p}) \]
Write a half to memory

\[ \text{void\ vstore\_half\_R}(\text{float\ data},\text{size\_t\ offset},Q\ \text{half\ *p}) \]
Write a half to memory

\[ \text{void\ vstore\_half\_R}(\text{double\ data},\text{size\_t\ offset},Q\ \text{half\ *p}) \]
Write a half to memory

\[ \text{void\ vstore\_halfn}(\text{floatn\ data},\text{size\_t\ offset},Q\ \text{half\ *p}) \]
Write a half vector to memory

\[ \text{floatn\ vload\_halfn}(\text{size\_t\ offset},\text{const}\ Q\ \text{half\ *p}) \]
Read multiple halves from memory

\[ \text{sizeof}\ (\text{floatn})\ \text{bytes} \]
size of data read from location \((p + (\text{offset} \times \text{n}))\)
Atomic Functions [6.11.11, 9.4]

\( T \) is type int or unsigned int. \( T \) may also be type float for atomic\_xchg, and type long or ulong for extended 64-bit atomic functions. \( Q \) is volatile \_global or volatile \_local, except \( Q \) must be \_volatile \_global for atomic\_xchg when \( T \) is float.

The built-in atomic functions for 32-bit values begin with atomic\_ while the extended 64-bit atomic functions begin with atom\_. For example:

- **Built-in atomic function**
  - atomic\_add()
- **Extended atomic function**
  - atom\_add()

Extended 64-bit atomic functions are enabled by the following pragma; extension\_name is one of cl\_khr\_int64\{base, extended\}_atomics:

```
#pragma OPENCL EXTENSION extension\_name : enable
```

### Atomic Functions

- **atomic\_add**\((Q T *p, T val)\): Read, add, and store
- **atomic\_sub**\((Q T *p, T val)\): Read, subtract, and store
- **atomic\_xchg**\((Q T *p, T val)\): Read, swap, and store
- **atomic\_inc**\((Q T *p)\): Read, increment, and store
- **atomic\_dec**\((Q T *p)\): Read, decrement, and store
- **atomic\_cmpxchg**\((Q T *p, T cmp, T val)\): Read and store (\(*p == cmp\) ? \(*p\) : \(*p\) )
- **atomic\_min**\((Q T *p, T val)\): Read, store min\((\(*p\), \(*val\)\))
- **atomic\_max**\((Q T *p, T val)\): Read, store max\((\(*p\), \(*val\)\))
- **atomic\_and**\((Q T *p, T val)\): Read, store \((\(*p\) & \(*val\))\)
- **atomic\_or**\((Q T *p, T val)\): Read, store \((\(*p\) | \(*val\))\)
- **atomic\_xor**\((Q T *p, T val)\): Read, store \((\(*p\) ^ \(*val\))\)
Async Copies and Prefetch Functions [6.11.10]

`T` is type `char`, `charn`, `uchar`, `ucharin`, `short`, `shortn`, `ushort`, `ushortn`, `int`, `intn`, `uint`, `uintn`, `long`, `longn`, `ulong`, `ulongn`, `float`, `floatn`, and optionally halfn double, doublen. Optional extensions enable the halfn, double, and doublen types.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>event_t async_work_group_copy(__local T *dst, const __global T *src, size_t num_gentypes, event_t event)</code></td>
<td>Copies <code>num_gentypes T</code> elements from <code>src</code> to <code>dst</code></td>
</tr>
<tr>
<td><code>event_t async_work_group_strided_copy(__local T *dst, const __global T *src, size_t num_gentypes, size_t src_stride, event_t event)</code></td>
<td>Copies <code>num_gentypes T</code> elements from <code>src</code> to <code>dst</code></td>
</tr>
<tr>
<td><code>void wait_group_events(int num_events, event_t *event_list)</code></td>
<td>Wait for events that identify the <code>async_work_group_copy</code> operations to complete</td>
</tr>
<tr>
<td><code>void prefetch(const __global T *p, size_t num_gentypes)</code></td>
<td>Prefetch <code>num_gentypes * sizeof(T)</code> bytes into the global cache</td>
</tr>
</tbody>
</table>
Image Read and Write Built-in Functions [6.11.13, 9.5.6.8]
The built-in functions defined in this section can only be used with image memory objects created with clCreateImage2D or clCreateImage3D. sampler specifies the addressing and filtering mode to use. H = To enable read_imageh and write_imageh, enable extension cl_khr_fp16. 3D = To enable type image3d_t in write_imagef, i, uil, enable extension cl_khr_3d_image_writes.

float4 read_imagef (image2d_t image, sampler_t sampler, int2 coord) float4 read_imagef (image2d_t image, sampler_t sampler, float2 coord)
int4 read_imagei (image2d_t image, sampler_t sampler, int2 coord) int4 read_imagei (image2d_t image, sampler_t sampler, float2 coord)
uint4 read_imageui (image2d_t image, sampler_t sampler, int2 coord) uint4 read_imageui (image2d_t image, sampler_t sampler, float2 coord)

half4 read_imageh (image2d_t image, sampler_t sampler, int2 coord) half4 read_imageh (image2d_t image, sampler_t sampler, float2 coord)

void write_imagef (image2d_t image, int2 coord, float4 color) void write_imagef (image2d_t image, int2 coord, float4 color)
void write_imagei (image2d_t image, int2 coord, int4 color) void write_imagei (image2d_t image, int2 coord, uint4 color)
void write_imageh (image2d_t image, int2 coord, half4 color)

float4 read_imagef (image3d_t image, sampler_t sampler, int4 coord) float4 read_imagef (image3d_t image, sampler_t sampler, float4 coord)
int4 read_imagei (image3d_t image, sampler_t sampler, int4 coord) int4 read_imagei (image3d_t image, sampler_t sampler, float4 coord)
uint4 read_imageui (image3d_t image, sampler_t sampler, int4 coord) uint4 read_imageui (image3d_t image, sampler_t sampler, float4 coord)

Read an element from a 2D image

Read an element from a 3D image

Write color value to (x, y) location specified by coord in the 2D image

Write color value to (x, y) location specified by coord in the 3D image

Use this pragma to enable type image3d_t in write_imagef, i, uil: 
#pragma OPENCL EXTENSION cl_khr_3d_image_writes : enable
void write_imagef (image3d_t image, int4 coord, float4 color) 3D
void write_imagei (image3d_t image, int4 coord, int4 color) 3D
void write_imageui (image3d_t image, int4 coord, uint4 color) 3D

Sampler Objects [6.5]
c_int clCreateImageSampler | cl_context context, c_int normalized_coords, c_int addressing_mode, cl_int filter_mode, c_int interp_mode, c_int *enclave, ret
cl_int clGetImageSamplerInfo (cl_mem image, cl_image_info parameter_type, cl_int parameter_value, c_int *parameter_value_ret)

Map and Unmap Image Objects [6.3.3]
void clEnqueueMapImage ( | cl_command_queue command_queue, cl_mem image, cl_command_type cmd, cl_map_flag flags, float map_region, size_t offset, size_t region, size_t size, size_t image_info, cl_int num_subregions, cl_int subregion_id, cl_event *event, cl_int *error_code, ret
cl_int clUnmapImage ( | cl_mem image, cl_event *event, cl_int *error_code, ret

Access Qualifiers [6.4]
apply to image2d_t, image3d_t and image1d_t types to declare if the image memory object is being read or written by a kernel. The default qualifier is __read_only. __read_only, only
__write_only, only

Sampler Objects [6.5]
c_int clCreateImageSampler | cl_context context, cl_int normalized_coords, c_int addressing_mode, cl_int filter_mode, cl_int interp_mode, c_int *enclave, ret
cl_int clGetImageSamplerInfo (cl_mem image, cl_image_info parameter_type, cl_int parameter_value, c_int *parameter_value_ret)

Map and Unmap Image Objects [6.3.3]
void clEnqueueMapImage ( | cl_command_queue command_queue, cl_mem image, cl_command_type cmd, cl_map_flag flags, float map_region, size_t offset, size_t region, size_t size, size_t image_info, cl_int num_subregions, cl_int subregion_id, cl_event *event, cl_int *error_code, ret
cl_int clUnmapImage ( | cl_mem image, cl_event *event, cl_int *error_code, ret

Access Qualifiers [6.4]
apply to image2d_t, image3d_t and image1d_t types to declare if the image memory object is being read or written by a kernel. The default qualifier is __read_only. __read_only, only
__write_only, only

Sampler Objects [6.5]
c_int clCreateImageSampler | cl_context context, cl_int normalized_coords, c_int addressing_mode, cl_int filter_mode, cl_int interp_mode, c_int *enclave, ret
cl_int clGetImageSamplerInfo (cl_mem image, cl_image_info parameter_type, cl_int parameter_value, c_int *parameter_value_ret)

Map and Unmap Image Objects [6.3.3]
void clEnqueueMapImage ( | cl_command_queue command_queue, cl_mem image, cl_command_type cmd, cl_map_flag flags, float map_region, size_t offset, size_t region, size_t size, size_t image_info, cl_int num_subregions, cl_int subregion_id, cl_event *event, cl_int *error_code, ret
cl_int clUnmapImage ( | cl_mem image, cl_event *event, cl_int *error_code, ret

Access Qualifiers [6.4]
apply to image2d_t, image3d_t and image1d_t types to declare if the image memory object is being read or written by a kernel. The default qualifier is __read_only. __read_only, only
__write_only, only

Sampler Objects [6.5]
c_int clCreateImageSampler | cl_context context, cl_int normalized_coords, c_int addressing_mode, cl_int filter_mode, cl_int interp_mode, c_int *enclave, ret
cl_int clGetImageSamplerInfo (cl_mem image, cl_image_info parameter_type, cl_int parameter_value, c_int *parameter_value_ret)

Map and Unmap Image Objects [6.3.3]
void clEnqueueMapImage ( | cl_command_queue command_queue, cl_mem image, cl_command_type cmd, cl_map_flag flags, float map_region, size_t offset, size_t region, size_t size, size_t image_info, cl_int num_subregions, cl_int subregion_id, cl_event *event, cl_int *error_code, ret
cl_int clUnmapImage ( | cl_mem image, cl_event *event, cl_int *error_code, ret

Access Qualifiers [6.4]
apply to image2d_t, image3d_t and image1d_t types to declare if the image memory object is being read or written by a kernel. The default qualifier is __read_only. __read_only, only
__write_only, only
Questions?

Derek Gerstmann
University of Western Australia

w: http://local.wasp.uwa.edu.au/~derek
e: <derek.gerstmann [at] uwa.edu.au>