PyCL Documentation

Release 0.1a

Ken Watford

December 15, 2015

Contents

PyCL is yet another OpenCL wrapper for Python. Its primary goal is simple: wrap OpenCL in such a way that as many Python implementations can use it as feasible. It is currently tested on CPython 2.{5,6,7}, 3.2, and PyPy 1.5. It is known to largely not work on Jython, whose ctypes library is still immature.

To achieve this, we eschew extension modules and dependencies outside of the standard library. Ideally things like NumPy arrays and PIL images should Just Work, but they shouldn't be required.

If you're looking to get actual work done in OpenCL, this probably isn't the distribution for you... yet. Before considering using PyCL for anything, give PyOpenCL a look. Its API is stable, its wrapper layer is fast C++, and it has fairly reasonable dependencies.

If you're looking to contribute, or just get the latest development release, take a look at our repository.

CHAPTER 1

Installation

It's on PyPI, so installation should be as easy as:

```
pip install pycl
      -or-
easy_install pycl
```

But it's a single module and there's nothing to compile, so downloading it from PyPI or the repository and using it directly works too.

To actually use it, though, you'll need an OpenCL platform installed. If you're on Mac OS X 10.6 or later, you're already done. Otherwise, download and install an appropriate platform from AMD, Intel, or NVIDIA.

CHAPTER 2

Module API Reference

Brief usage example:

```
from pycl import *
from array import array
source = '''
kernel void mxplusb(float m, global float *x, float b, global float *out) {
    int i = get_global_id(0);
    out[i] = m*x[i]+b;
}
'''
ctx = clCreateContext()
queue = clCreateCommandQueue(ctx)
program = clCreateProgramWithSource(ctx, source).build()
kernel = program['mxplusb']
kernel.argtypes = (cl_float, cl_mem, cl_float, cl_mem)
x = array('f', range(10))
x_buf, in_evt = buffer_from_pyarray(queue, x, blocking=False)
y_buf = x_buf.empty_like_this()
run_evt = kernel(2, x_buf, 5, y_buf).on(queue, len(x), wait_for=in_evt)
y, evt = buffer_to_pyarray(queue, y_buf, wait_for=run_evt, like=x)
evt.wait()
print y
```

For Numpy users, see *buffer_from_ndarray()* and *buffer_to_ndarray()*.

Additionally, if run as a script, will print out a summary of your platforms and devices.

Most of the C typedefs are available as subclasses of Python ctypes datatypes. The spelling might be slightly different.

The various enumeration and bitfield types have attributes representing their defined constants (e.g. *CL_DEVICE_TYPE_GPU*). These constants are also available at the module level, in case you can't remember what type *CL_QUEUED* is supposed to be. They are all somewhat magical in that they'll make a reasonable effort to pretty-print themselves:

```
>>> cl_device_type.CL_DEVICE_TYPE_GPU | cl_device_type.CL_DEVICE_TYPE_CPU
CL_DEVICE_TYPE_CPU | CL_DEVICE_TYPE_GPU
>>> cl_mem_info(0x1100)
CL_MEM_TYPE
```

The types representing various object-like datastructures often have attributes so that you can view their infos without needing to call the appropriate `clGetThingInfo` function. They may have other methods and behaviors.

One last note about the datatypes: despite any appearance of magic and high-level function, these are just ctypes objects. It is entirely possible for you to assign things to the `value` attribute of the enum/bitfield constants or of object-

like items. Overwriting constants and clobbering pointers is generally a bad idea, though, so you should probably avoid it. (I tried vetoing assignment to .value, but PyPy didn't like that. So you're on your own.)

Wrapped OpenCL functions have their usual naming convention (`clDoSomething`). These are't the naked C function pointers - you will find that the argument lists, return types, and exception raising are more in line with Python. Check the docstrings. That said, you can refer to the function pointer itself with the wrapped function's `call` attribute, which is how the functions themselves do it. The function pointer itself has argument type, return type, and error checking added in the usual ctypes manner.

The list of wrapped functions is *very* incomplete. Feel free to contribute if you need a function that hasn't been wrapped yet.

There are currently no plans to provide wrappers for OpenCL extensions (like OpenGL interop). Maybe later.

class `pycl.`**`cl_device_type`**

Bitfield used by *`clCreateContextFromType()`* to create a context from one or more matching device types.

See also *`cl_device.type`* and *`clGetDeviceInfo()`*

`CL_DEVICE_TYPE_ACCELERATOR`

`CL_DEVICE_TYPE_CPU`

`CL_DEVICE_TYPE_ALL`

`CL_DEVICE_TYPE_GPU`

`CL_DEVICE_TYPE_DEFAULT`

class `pycl.`**`cl_errnum`**

A status code returned by most OpenCL functions. Exceptions exist for each error code and will be raised in the event that the code is flagged by any wrapper function. The exception names are formed by removing the 'CL', title-casing the words, removing the underscores, and appending 'Error' to the end. Some of these are a little redundant, like `BuildProgramFailureError`.

And no, there is no `SuccessError`.

`CL_PROFILING_INFO_NOT_AVAILABLE`

`CL_INVALID_KERNEL_DEFINITION`

`CL_INVALID_VALUE`

`CL_INVALID_PROGRAM_EXECUTABLE`

`CL_INVALID_IMAGE_SIZE`

`CL_INVALID_WORK_DIMENSION`

`CL_INVALID_EVENT`

`CL_BUILD_PROGRAM_FAILURE`

`CL_INVALID_WORK_GROUP_SIZE`

`CL_MEM_COPY_OVERLAP`

`CL_INVALID_PROGRAM`

`CL_INVALID_COMMAND_QUEUE`

`CL_INVALID_KERNEL_NAME`

`CL_INVALID_GL_SHAREGROUP_REFERENCE_KHR`

`CL_DEVICE_NOT_AVAILABLE`

CL_INVALID_QUEUE_PROPERTIES

CL_INVALID_CONTEXT

CL_INVALID_KERNEL_ARGS

CL_OUT_OF_HOST_MEMORY

CL_INVALID_BUILD_OPTIONS

CL_MAP_FAILURE

CL_INVALID_KERNEL

CL_INVALID_GLOBAL_WORK_SIZE

CL_IMAGE_FORMAT_MISMATCH

CL_KERNEL_ARG_INFO_NOT_AVAILABLE

CL_INVALID_GL_OBJECT

CL_MISALIGNED_SUB_BUFFER_OFFSET

CL_INVALID_OPERATION

CL_INVALID_PLATFORM

CL_INVALID_DEVICE_TYPE

CL_INVALID_PROPERTY

CL_INVALID_ARG_VALUE

CL_LINKER_NOT_AVAILABLE

CL_INVALID_EVENT_WAIT_LIST

CL_OUT_OF_RESOURCES

CL_INVALID_DEVICE

CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST

CL_INVALID_ARG_SIZE

CL_INVALID_BUFFER_SIZE

CL_DEVICE_NOT_FOUND

CL_INVALID_WORK_ITEM_SIZE

CL_INVALID_MIP_LEVEL

CL_INVALID_MEM_OBJECT

CL_SUCCESS

CL_INVALID_SAMPLER

CL_INVALID_HOST_PTR

CL_COMPILE_PROGRAM_FAILURE

CL_COMPILER_NOT_AVAILABLE

CL_INVALID_BINARY

CL_DEVICE_PARTITION_FAILED

CL_MEM_OBJECT_ALLOCATION_FAILURE

CL_INVALID_IMAGE_FORMAT_DESCRIPTOR

CL_LINK_PROGRAM_FAILURE

CL_IMAGE_FORMAT_NOT_SUPPORTED

CL_INVALID_ARG_INDEX

CL_INVALID_GLOBAL_OFFSET

class pycl.**cl_platform_info**

The set of possible parameter names used with the *clGetPlatformInfo()* function.

CL_PLATFORM_PROFILE

CL_PLATFORM_EXTENSIONS

CL_PLATFORM_VENDOR

CL_PLATFORM_VERSION

CL_PLATFORM_NAME

class pycl.**cl_device_info**

The set of possible parameter names used with the *clGetDeviceInfo()* function.

CL_DEVICE_IMAGE2D_MAX_HEIGHT

CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS

CL_DEVICE_IMAGE_SUPPORT

CL_DEVICE_MIN_DATA_TYPE_ALIGN_SIZE

CL_DEVICE_PARTITION_AFFINITY_DOMAIN

CL_DEVICE_MEM_BASE_ADDR_ALIGN

CL_DEVICE_ENDIAN_LITTLE

CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE

CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE

CL_DEVICE_PREFERRED_VECTOR_WIDTH_LONG

CL_DEVICE_PREFERRED_VECTOR_WIDTH_SHORT

CL_DEVICE_PREFERRED_VECTOR_WIDTH_INT

CL_DEVICE_NATIVE_VECTOR_WIDTH_LONG

CL_DEVICE_GLOBAL_MEM_SIZE

CL_DEVICE_MAX_COMPUTE_UNITS

CL_DEVICE_REFERENCE_COUNT

CL_DEVICE_VENDOR

CL_DEVICE_IMAGE_MAX_BUFFER_SIZE

CL_DEVICE_PREFERRED_VECTOR_WIDTH_HALF

CL_DRIVER_VERSION

CL_DEVICE_DOUBLE_FP_CONFIG

CL_DEVICE_NATIVE_VECTOR_WIDTH_HALF

CL_DEVICE_IMAGE_PITCH_ALIGNMENT

CL_DEVICE_PROFILING_TIMER_RESOLUTION

CL_DEVICE_MAX_READ_IMAGE_ARGS

CL_DEVICE_QUEUE_PROPERTIES

CL_DEVICE_TYPE

CL_DEVICE_MAX_WORK_GROUP_SIZE

CL_DEVICE_IMAGE3D_MAX_DEPTH

CL_DEVICE_IMAGE_MAX_ARRAY_SIZE

CL_DEVICE_MAX_CONSTANT_ARGS

CL_DEVICE_IMAGE3D_MAX_HEIGHT

CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE

CL_DEVICE_ADDRESS_BITS

CL_DEVICE_EXTENSIONS

CL_DEVICE_NATIVE_VECTOR_WIDTH_SHORT

CL_DEVICE_BUILT_IN_KERNELS

CL_DEVICE_GLOBAL_MEM_CACHE_TYPE

CL_DEVICE_NATIVE_VECTOR_WIDTH_DOUBLE

CL_DEVICE_PLATFORM

CL_DEVICE_PREFERRED_VECTOR_WIDTH_CHAR

CL_DEVICE_PARTITION_TYPE

CL_DEVICE_PARENT_DEVICE

CL_DEVICE_IMAGE_BASE_ADDRESS_ALIGNMENT

CL_DEVICE_LINKER_AVAILABLE

CL_DEVICE_COMPILER_AVAILABLE

CL_DEVICE_LOCAL_MEM_SIZE

CL_DEVICE_AVAILABLE

CL_DEVICE_NATIVE_VECTOR_WIDTH_FLOAT

CL_DEVICE_NATIVE_VECTOR_WIDTH_INT

CL_DEVICE_HOST_UNIFIED_MEMORY

CL_DEVICE_GLOBAL_MEM_CACHE_SIZE

CL_DEVICE_PRINTF_BUFFER_SIZE

CL_DEVICE_PARTITION_MAX_SUB_DEVICES

CL_DEVICE_IMAGE3D_MAX_WIDTH

CL_DEVICE_VERSION

CL_DEVICE_NATIVE_VECTOR_WIDTH_CHAR

CL_DEVICE_IMAGE2D_MAX_WIDTH

CL_DEVICE_SINGLE_FP_CONFIG

CL_DEVICE_PROFILE

CL_DEVICE_HALF_FP_CONFIG

CL_DEVICE_PREFERRED_INTEROP_USER_SYNC

CL_DEVICE_NAME

CL_DEVICE_OPENCL_C_VERSION

CL_DEVICE_ERROR_CORRECTION_SUPPORT

CL_DEVICE_MAX_MEM_ALLOC_SIZE

CL_DEVICE_MAX_CLOCK_FREQUENCY

CL_DEVICE_PARTITION_PROPERTIES

CL_DEVICE_MAX_WORK_ITEM_SIZES

CL_DEVICE_EXECUTION_CAPABILITIES

CL_DEVICE_VENDOR_ID

CL_DEVICE_MAX_WRITE_IMAGE_ARGS

CL_DEVICE_PREFERRED_VECTOR_WIDTH_FLOAT

CL_DEVICE_MAX_PARAMETER_SIZE

CL_DEVICE_MAX_SAMPLERS

CL_DEVICE_LOCAL_MEM_TYPE

class pycl.**cl_device_fp_config**
One of the possible return types from *clGetDeviceInfo()*. Bitfield identifying the floating point capabilities of the device.

CL_FP_ROUND_TO_ZERO

CL_FP_FMA

CL_FP_ROUND_TO_INF

CL_FP_DENORM

CL_FP_ROUND_TO_NEAREST

CL_FP_INF_NAN

CL_FP_SOFT_FLOAT

class pycl.**cl_device_mem_cache_type**
One of the possible return types from *clGetDeviceInfo()*. Describes the nature of the device's cache, if any.

CL_READ_WRITE_CACHE

CL_READ_ONLY_CACHE

CL_NONE

class pycl.**cl_device_local_mem_type**
One of the possible return types from *clGetDeviceInfo()*. Describes where 'local' memory lives in the device. Presumably, *CL_GLOBAL* means the device's local memory lives in the same address space as its global memory.

CL_LOCAL

CL_GLOBAL

class pycl.**cl_device_exec_capabilities**

One of the possible return types from *clGetDeviceInfo()*. Bitfield identifying what kind of kernels can be executed. All devices can execute OpenCL C kernels, but some have their own native kernel types as well.

CL_EXEC_KERNEL

CL_EXEC_NATIVE_KERNEL

class pycl.**cl_device_partition_property**

CL_DEVICE_PARTITION_BY_COUNTS

CL_DEVICE_PARTITION_EQUALLY

CL_DEVICE_PARTITION_BY_COUNTS_LIST_END

CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN

class pycl.**cl_device_affinity_domain**

CL_DEVICE_AFFINITY_DOMAIN_L2_CACHE

CL_DEVICE_AFFINITY_DOMAIN_NUMA

CL_DEVICE_AFFINITY_DOMAIN_L3_CACHE

CL_DEVICE_AFFINITY_DOMAIN_L1_CACHE

CL_DEVICE_AFFINITY_DOMAIN_L4_CACHE

CL_DEVICE_AFFINITY_DOMAIN_NEXT_PARTITIONABLE

class pycl.**cl_command_queue_properties**

Bitfield representing the properties of a command queue.

CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE

CL_QUEUE_PROFILING_ENABLE

class pycl.**cl_context_properties**

If you find yourself looking at an array of these and need to make any sense of them... good luck! It's a list of key-value pairs, null-terminated. The keys are unsigned ints representing enum constants. *CL_CONTEXT_PLATFORM* (0x1084) is the most common one you'll see. I believe the rest are parts of extensions, such as the OpenGL interop extension.

The meaning of the odd elements depends entirely on the enum that came just before it. In the case of *CL_CONTEXT_PLATFORM*, the value represents a pointer to a cl_platform object.

class pycl.**cl_context_info**

Parameter names understood by *clGetContextInfo()*.

Note that `cl_context_inf.CL_CONTEXT_PLATFORM` does not technically belong here, and the C-level code won't accept it. The wrapped version of *clGetContextInfo()* will, however, recognize it and extract the appropriate value from the context's properties list.

CL_GLX_DISPLAY_KHR

CL_CONTEXT_PROPERTIES

CL_GL_CONTEXT_KHR

CL_CONTEXT_REFERENCE_COUNT

CL_CONTEXT_DEVICES

CL_EGL_DISPLAY_KHR

CL_WGL_HDC_KHR

CL_CGL_SHAREGROUP_KHR

CL_CONTEXT_PROPERTY_USE_CGL_SHAREGROUP_APPLE

CL_CONTEXT_NUM_DEVICES

CL_CONTEXT_PLATFORM

class pycl.**cl_command_queue_info**
Parameter names understood by *clGetCommandQueueInfo()*

CL_QUEUE_DEVICE

CL_QUEUE_PROPERTIES

CL_QUEUE_CONTEXT

CL_QUEUE_REFERENCE_COUNT

class pycl.**cl_channel_order**
Indicates the meanings of vector fields in an image.

CL_RG

CL_RGBA

CL_R

CL_RGB

CL_ARGB

CL_INTENSITY

CL_RA

CL_RGx

CL_A

CL_LUMINANCE

CL_Rx

CL_RGBx

CL_BGRA

class pycl.**cl_channel_type**
Indicates the type and size of image channels.

CL_UNORM_SHORT_555

CL_SNORM_INT8

CL_SIGNED_INT8

CL_UNORM_SHORT_565

CL_SNORM_INT16

CL_UNORM_INT16

CL_SIGNED_INT32

CL_UNORM_INT_101010

CL_UNSIGNED_INT8

CL_HALF_FLOAT

CL_UNORM_INT8

CL_UNSIGNED_INT32

CL_FLOAT

CL_UNSIGNED_INT16

CL_SIGNED_INT16

class pycl.**cl_mem_flags**
Bitfield used when constructing a memory object. Indicates both the read/write status of the memory as well as how the memory interacts with whatever host pointer was provided. See the OpenCL docs for *clCreateBuffer()* for more information.

CL_MEM_USE_HOST_PTR

CL_MEM_WRITE_ONLY

CL_MEM_COPY_HOST_PTR

CL_MEM_READ_WRITE

CL_MEM_ALLOC_HOST_PTR

CL_MEM_READ_ONLY

class pycl.**cl_mem_object_type**
Possible return type for *clGetMemObjectInfo()*. Indicates the type of the memory object.

CL_MEM_OBJECT_IMAGE3D

CL_MEM_OBJECT_IMAGE2D

CL_MEM_OBJECT_BUFFER

class pycl.**cl_mem_info**
Parameter names accepted by *clGetMemObjectInfo()*

CL_MEM_REFERENCE_COUNT

CL_MEM_TYPE

CL_MEM_ASSOCIATED_MEMOBJECT

CL_MEM_CONTEXT

CL_MEM_OFFSET

CL_MEM_MAP_COUNT

CL_MEM_HOST_PTR

CL_MEM_SIZE

CL_MEM_FLAGS

class pycl.**cl_mem_migration_flags**
The set of possible parameter names used with the `clEnqueueMigrateMemObjects()` function.

CL_MIGRATE_MEM_OBJECT_CONTENT_UNDEFINED

CL_MIGRATE_MEM_OBJECT_HOST

class pycl.**cl_image_info**
Parameter names accepted by `clGetImageInfo()`

CL_IMAGE_HEIGHT

CL_IMAGE_SLICE_PITCH

CL_IMAGE_FORMAT

CL_IMAGE_ELEMENT_SIZE

CL_IMAGE_WIDTH

CL_IMAGE_DEPTH

CL_IMAGE_ROW_PITCH

class pycl.**cl_buffer_create_type**
Parameter type for `clCreateSubBuffer()` that indicates how the subbuffer will be described.

The only supported value is *CL_BUFFER_CREATE_TYPE_REGION*, which indicates the subbuffer will be a contiguous region as defined by a *cl_buffer_region* struct.

CL_BUFFER_CREATE_TYPE_REGION

class pycl.**cl_addressing_mode**
Addressing mode for sampler objects. Returned by `clGetSamplerInfo()`.

CL_ADDRESS_MIRRORED_REPEAT

CL_ADDRESS_CLAMP

CL_ADDRESS_NONE

CL_ADDRESS_CLAMP_TO_EDGE

CL_ADDRESS_REPEAT

class pycl.**cl_filter_mode**
Filter mode for sampler objects. Returned by `clGetSamplerInfo()`.

CL_FILTER_NEAREST

CL_FILTER_LINEAR

class pycl.**cl_sampler_info**
Parameter names for `clGetSamplerInfo()`.

CL_SAMPLER_REFERENCE_COUNT

CL_SAMPLER_ADDRESSING_MODE

CL_SAMPLER_FILTER_MODE

CL_SAMPLER_NORMALIZED_COORDS

CL_SAMPLER_CONTEXT

class pycl.**cl_map_flags**
Read/write flags used for applying memory mappings to memory objects. See `clEnqueueMapBuffer()` and `clEnqueueMapImage()`.

CL_MAP_WRITE

CL_MAP_READ

class pycl.**cl_program_info**
Parameter names for *clGetProgramInfo()*

CL_PROGRAM_REFERENCE_COUNT

CL_PROGRAM_DEVICES

CL_PROGRAM_NUM_DEVICES

CL_PROGRAM_CONTEXT

CL_PROGRAM_SOURCE

CL_PROGRAM_BINARIES

CL_PROGRAM_BINARY_SIZES

class pycl.**cl_program_build_info**
Parameter names for *clGetProgramBuildInfo()*

CL_PROGRAM_BUILD_STATUS

CL_PROGRAM_BUILD_LOG

CL_PROGRAM_BUILD_OPTIONS

class pycl.**cl_build_status**
Returned by *clGetProgramBuildInfo()*. Indicates build status for the program on the specified device.

CL_BUILD_SUCCESS

CL_BUILD_IN_PROGRESS

CL_BUILD_NONE

CL_BUILD_ERROR

class pycl.**cl_kernel_info**
Parameter names for *clGetKernelInfo()*

CL_KERNEL_PROGRAM

CL_KERNEL_NUM_ARGS

CL_KERNEL_CONTEXT

CL_KERNEL_REFERENCE_COUNT

CL_KERNEL_FUNCTION_NAME

class pycl.**cl_kernel_work_group_info**
Parameter names for *clGetKernelWorkGroupInfo()*

CL_KERNEL_COMPILE_WORK_GROUP_SIZE

CL_KERNEL_LOCAL_MEM_SIZE

CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE

CL_KERNEL_PRIVATE_MEM_SIZE

CL_KERNEL_WORK_GROUP_SIZE

class pycl.**cl_event_info**
Parameter names for *clGetEventInfo()*

CL_EVENT_COMMAND_TYPE

CL_EVENT_COMMAND_EXECUTION_STATUS

CL_EVENT_COMMAND_QUEUE

CL_EVENT_CONTEXT

CL_EVENT_REFERENCE_COUNT

class pycl.**cl_command_type**
Command types recorded on events and returned by *clGetEventInfo()*.

CL_COMMAND_NDRANGE_KERNEL

CL_COMMAND_COPY_BUFFER

CL_COMMAND_RELEASE_GL_OBJECTS

CL_COMMAND_COPY_IMAGE

CL_COMMAND_READ_BUFFER_RECT

CL_COMMAND_COPY_BUFFER_RECT

CL_COMMAND_WRITE_BUFFER

CL_COMMAND_MARKER

CL_COMMAND_WRITE_IMAGE

CL_COMMAND_ACQUIRE_GL_OBJECTS

CL_COMMAND_COPY_BUFFER_TO_IMAGE

CL_COMMAND_READ_BUFFER

CL_COMMAND_READ_IMAGE

CL_COMMAND_UNMAP_MEM_OBJECT

CL_COMMAND_NATIVE_KERNEL

CL_COMMAND_MAP_IMAGE

CL_COMMAND_COPY_IMAGE_TO_BUFFER

CL_COMMAND_MAP_BUFFER

CL_COMMAND_USER

CL_COMMAND_WRITE_BUFFER_RECT

CL_COMMAND_TASK

class pycl.**cl_command_execution_status**
Status of the command associated with an event, returned by *clGetEventInfo()*.

CL_QUEUED

CL_COMPLETE

CL_SUBMITTED

CL_RUNNING

class pycl.**cl_profiling_info**
Parameter names for `clGetEventProfilingInfo()`. Indicates the point in time of the event's life that should be queried.

CL_PROFILING_COMMAND_SUBMIT

CL_PROFILING_COMMAND_START

CL_PROFILING_COMMAND_END

CL_PROFILING_COMMAND_QUEUED

class pycl.**cl_image_format**
Represents image formats. See clCreateImage2D().

image_channel_order
A *cl_channel_order* value

image_channel_data_type
A *cl_channel_type* value

class pycl.**cl_buffer_region**
A buffer region has two fields: origin and size. Both are of type size_t.

See clCreateSubBuffer() for usage.

exception pycl.**OpenCLError**
The base class from which all of the (generated) OpenCL errors are descended. These exceptions correspond to the *cl_errnum* status codes.

class pycl.**cl_event**
An OpenCL Event object. Returned by functions that add commands to a *cl_command_queue*, and often accepted (singly or in lists) by the wait_for argument of these functions to impose ordering.

Use *wait()* to wait for a particular event to complete, or *clWaitForEvents()* to wait for several of them at once.

These objects participate in OpenCL's reference counting scheme.

queue
The queue this event was emitted from.

context
The context this event exists within.

type
The type of command this event is linked to. See *cl_command_type*.

status
Execution status of the command the event is linked to. See cl_command_exec_status.

reference_count
Reference count for OpenCL garbage collection.

wait()
Blocks until this event completes.

pycl.**clWaitForEvents**(**events*)
Accepts several events and blocks until they all complete.

pycl.**clGetEventInfo**(*event*, *param_name*)

Parameters **param_name** – An instance of *cl_event_info*.

Event information can be more easily obtained by querying the properties of the event object, which in turn will call this function.

class pycl.**cl_platform**
Represents an OpenCL Platform. Should not be directly instantiated by users of PyCL. Use *clGetPlatformIDs()* or the platform attribute of some OpenCL objects to procure a cl_platform instance.

name
Name of the platform. (str)

vendor
Vendor that distributes the platform. (str)

version
Platform version. Likely starts with 'OpenCL 1.1'. (str)

extensions
Platform extensions supported. (list of str) Note that devices have their own set of extensions which should be inspected separately.

profile
One of 'FULL_PROFILE' or 'EMBEDDED_PROFILE'.

devices
All devices available on this platform. (list of cl_device)

pycl.**clGetPlatformIDs**()
Returns a list of *cl_platform* objects available on your system. It should probably not be possible for this list to be empty if you are able to call this function.

```
>>> clGetPlatformIDs()
(<cl_platform '...'>...)
```

pycl.**clGetPlatformInfo**(*platform*, *param_name*)

Parameters **param_name** – One of *cl_platform_info*.

cl_platform objects have attributes that will call this for you, so you should probably use those instead of calling this directly.

```
>>> plat = clGetPlatformIDs()[0]
>>> clGetPlatformInfo(plat, cl_platform_info.CL_PLATFORM_VERSION)
'OpenCL ...'
>>> plat.version
'OpenCL ...'
```

Note that *CL_PLATFORM_EXTENSIONS* returns a string while the `extensions` attribute returns a list:

```
>>> clGetPlatformInfo(plat, cl_platform_info.CL_PLATFORM_EXTENSIONS)
'...'
>>> plat.extensions
[...]
```

class pycl.**cl_device**
Represents an OpenCL Device belonging to some platform. Should not be directly instantiated by users of PyCL. Use *clGetDeviceIDs()* or the `devices` attribute of some OpenCL objects to procure a cl_device instances.

address_bits
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_ADDRESS_BITS*

available
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_AVAILABLE*

built_in_kernels
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_BUILT_IN_KERNELS*

compiler_available
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_COMPILER_AVAILABLE*

double_fp_config
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_DOUBLE_FP_CONFIG*

endian_little
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_ENDIAN_LITTLE*

error_correction_support
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_ERROR_CORRECTION_SUPPORT*

execution_capabilities
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_EXECUTION_CAPABILITIES*

global_mem_cache_size
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_GLOBAL_MEM_CACHE_SIZE*

global_mem_cache_type
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_GLOBAL_MEM_CACHE_TYPE*

global_mem_cacheline_size
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE*

global_mem_size
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_GLOBAL_MEM_SIZE*

half_fp_config
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_HALF_FP_CONFIG*

host_unified_memory
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_HOST_UNIFIED_MEMORY*

image2d_max_height
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_IMAGE2D_MAX_HEIGHT*

image2d_max_width
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_IMAGE2D_MAX_WIDTH*

image3d_max_depth
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_IMAGE3D_MAX_DEPTH*

image3d_max_height
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_IMAGE3D_MAX_HEIGHT*

image3d_max_width
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_IMAGE3D_MAX_WIDTH*

image_base_address_alignment
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_IMAGE_BASE_ADDRESS_ALIGNMENT*

image_max_array_size
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_IMAGE_MAX_ARRAY_SIZE*

image_max_buffer_size
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_IMAGE_MAX_BUFFER_SIZE*

image_pitch_alignment
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_IMAGE_PITCH_ALIGNMENT*

image_support
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_IMAGE_SUPPORT*

linker_available
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_LINKER_AVAILABLE*

local_mem_size
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_LOCAL_MEM_SIZE*

local_mem_type
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_LOCAL_MEM_TYPE*

max_clock_frequency
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_MAX_CLOCK_FREQUENCY*

max_compute_units
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_MAX_COMPUTE_UNITS*

max_constant_args
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_MAX_CONSTANT_ARGS*

max_constant_buffer_size
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE*

max_mem_alloc_size
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_MAX_MEM_ALLOC_SIZE*

max_parameter_size
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_MAX_PARAMETER_SIZE*

max_read_image_args
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_MAX_READ_IMAGE_ARGS*

max_samplers
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_MAX_SAMPLERS*

max_work_group_size
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_MAX_WORK_GROUP_SIZE*

max_work_item_dimensions
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS*

max_work_item_sizes
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_MAX_WORK_ITEM_SIZES*

max_write_image_args
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_MAX_WRITE_IMAGE_ARGS*

mem_base_addr_align
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_MEM_BASE_ADDR_ALIGN*

min_data_type_align_size
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_MIN_DATA_TYPE_ALIGN_SIZE*

name
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_NAME*

native_vector_width_char
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_NATIVE_VECTOR_WIDTH_CHAR*

native_vector_width_double
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_NATIVE_VECTOR_WIDTH_DOUBLE*

native_vector_width_float
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_NATIVE_VECTOR_WIDTH_FLOAT*

native_vector_width_half
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_NATIVE_VECTOR_WIDTH_HALF*

native_vector_width_int
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_NATIVE_VECTOR_WIDTH_INT*

native_vector_width_long
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_NATIVE_VECTOR_WIDTH_LONG*

native_vector_width_short
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_NATIVE_VECTOR_WIDTH_SHORT*

opencl_c_version
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_OPENCL_C_VERSION*

parent_device
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_PARENT_DEVICE*

partition_affinity_domain
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_PARTITION_AFFINITY_DOMAIN*

partition_max_sub_devices
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_PARTITION_MAX_SUB_DEVICES*

partition_properties
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_PARTITION_PROPERTIES*

partition_type
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_PARTITION_TYPE*

platform
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_PLATFORM*

preferred_interop_user_sync
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_PREFERRED_INTEROP_USER_SYNC*

preferred_vector_width_char
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_PREFERRED_VECTOR_WIDTH_CHAR*

preferred_vector_width_double
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE*

preferred_vector_width_float
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_PREFERRED_VECTOR_WIDTH_FLOAT*

preferred_vector_width_half
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_PREFERRED_VECTOR_WIDTH_HALF*

preferred_vector_width_int
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_PREFERRED_VECTOR_WIDTH_INT*

preferred_vector_width_long
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_PREFERRED_VECTOR_WIDTH_LONG*

preferred_vector_width_short
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_PREFERRED_VECTOR_WIDTH_SHORT*

printf_buffer_size
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_PRINTF_BUFFER_SIZE*

profile
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_PROFILE*

profiling_timer_resolution
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_PROFILING_TIMER_RESOLUTION*

queue_properties
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_QUEUE_PROPERTIES*

reference_count
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_REFERENCE_COUNT*

single_fp_config
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_SINGLE_FP_CONFIG*

type
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_TYPE*

vendor
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_VENDOR*

vendor_id
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_VENDOR_ID*

version
Same as calling *clGetDeviceInfo()* with *CL_DEVICE_VERSION*

pycl.**clGetDeviceIDs**(*platform=None*, *device_type=CL_DEVICE_TYPE_ALL*)

Parameters

- **platform** – The *cl_platform* whose devices you are interested in. If none is provided, the first platform on the system is used.
- **device_type** – A *cl_device_type* bitfield indicating which devices should be listed. By default, all are listed.

```
>>> clGetDeviceIDs()
(<cl_device '...'>...)
```

pycl.**clGetDeviceInfo**(*device*, *param_name*)

Parameters

- **device** – A *cl_device*.
- **param_name** – The *cl_device_info* item to be queried.

cl_device objects have attributes that will call this for you, so you should probably use those instead of calling this directly.

```
>>> d = clGetDeviceIDs()[0]
>>> clGetDeviceInfo(d, cl_device_info.CL_DEVICE_NAME)
'...'
>>> clGetDeviceInfo(d, cl_device_info.CL_DEVICE_TYPE)
CL_DEVICE_TYPE_...
>>> d.available
True
>>> d.max_work_item_sizes
(...)
```

Note that *CL_DEVICE_EXTENSIONS* returns a string while the `extensions` attribute returns a list:

```
>>> clGetDeviceInfo(d, cl_device_info.CL_DEVICE_EXTENSIONS)
'...'
>>> d.extensions
[...]
```

class pycl.**cl_context**
Represents an OpenCL Context instance.

Use *clCreateContext()* or *clCreateContextFromType()* to create a new context.

Participates in OpenCL's reference counting scheme.

platform
Retrieve the platform this context was made using. (cl_platform)

reference_count
Reference count for OpenCL's internal garbage collector. (int) Using `clReleaseContext()` via pycl is an excellent way to generate segmentation faults.

num_devices
Number of devices present in this particular context. (int)

devices
List of devices present in this particular context. (list of *cl_device*)

properties
Low-level ctypes array that is probably not user-interpretable.

pycl.**clCreateContext**(*devices=None*, *platform=None*, *other_props=None*)
Create a context with the given devices and platform.

Parameters

- **devices** – A list of devices. If None, the first device from the given platform is used.
- **platform** – If no platform or devices are provided, the first platform found will be used. If a device list is provided but no platform, the platform will be recovered from the devices.

If you just need a context and don't care what you get, calling with no arguments should hopefully get you something usable.

```
>>> clCreateContext()
<cl_context ...>
>>> one_device = clGetDeviceIDs()[0]
>>> clCreateContext(devices = [one_device])
<cl_context ...>
```

pycl.**clCreateContextFromType**(*device_type=CL_DEVICE_TYPE_DEFAULT*, *platform=None*, *other_props=None*)
Like *clCreateContext()*, but works by device type instead of expecting you to list the desired devices. This can, for instance, be used to create a context with GPU devices without the user having to pick a platform and inspect its device list.

Parameters

- **device_type** – A *cl_device_type* field indicating which types of devices should be included.
- **platform** – A *cl_platform*. If no platform is provided, each platform will be tried in turn until a context with the specified device type can created.

If you just need a context and don't care what you get, calling with no arguments should hopefully get you something usable.

```
>>> clCreateContextFromType(CL_DEVICE_TYPE_CPU | CL_DEVICE_TYPE_GPU)
<cl_context ...>
```

pycl.**clGetContextInfo**(*context*, *param_name*)
Retrieve context info.

Parameters

- **context** – *cl_context*.

- **param_name** – One of the *cl_context_info* values.

cl_context objects have attributes that will call this for you, so you should probably use those instead of calling this directly.

```
>>> ctx = clCreateContext()
>>> clGetContextInfo(ctx, cl_context_info.CL_CONTEXT_DEVICES)
(<cl_device ...>...)
>>> ctx.platform
<cl_platform ...>
>>> ctx.reference_count
1
>>> ctx.properties
<...cl_context_properties_Array...>
```

class pycl.**cl_command_queue**

Represents an OpenCL Command Queue instance. Should not be directly instantiated by users of PyCL. Use *clCreateCommandQueue()* to create a new queue.

context

The context associated with the command queue. (*cl_context*)

device

The device associated with the command queue. (*cl_device*)

properties

Command queue property bitfield. (*cl_command_queue_properties*)

reference_count

Reference count for OpenCL's garbage collector. (int)

pycl.**clCreateCommandQueue**(*context=None*, *device=None*, *properties=None*)

Parameters

- **context** – *cl_context*. If not provided, one will be generated for you by calling *clCreateContext()* with no arguments. (it can later be retrieved via the `context` attribute)
- **device** – The *cl_device* that will be fed by this queue. If no device is provided, the first device in the context will be used.
- **properties** – A *cl_command_queue_properties* bitfield.

pycl.**clGetCommandQueueInfo**(*queue*, *param_name*)

Parameters

- **queue** – *cl_command_queue*.
- **param_name** – One of the *cl_command_queue_info* values.

```
>>> q = clCreateCommandQueue()
>>> q.context
<cl_context ...>
>>> q.device
<cl_device ...>
>>> q.properties
NONE
>>> q.reference_count
1
```

class pycl.**cl_mem**

Represents an OpenCL memory object, typically a buffer or image.

See the individual types (*cl_buffer* and *cl_image*) for details. PyCL should probably never give you a direct instance of this class - treat it as abstract.

Memory objects are reference counted.

size
Memory size, in bytes.

reference_count
Reference count for OpenCL garbage collector.

map_count
Number of memory maps currently active for this object.

hostptr
Pointer to host address associated with this memory object at the time of creation. The meaning varies depending on the flags. (type is `void*`)

flags
The *cl_mem_flags* the object was created with.

type
The `cl_mem_type` of the object.

context
The *cl_context* the memory belongs to.

empty_like_this()
Creates an empty read/write buffer of the same size in the same context and returns it.

class pycl.**cl_buffer**
A subclass of *cl_mem* representing memory buffers. Create these with *clCreateBuffer()* or `clCreateSubBuffer()`

base
Base memory object (for sub-buffers)

offset
Offset, in bytes, from origin (for sub-buffers)

class pycl.**cl_image**
A subclass of *cl_mem* representing 2D or 3D images. Create these with `clCreateImage2D()` or `clCreateImage3D()`.

pycl.**clCreateBuffer**(*context*, *size*, *flags=CL_MEM_READ_WRITE*, *host_ptr=None*)

Parameters

- **context** – *cl_context* that will own this memory.
- **size** – Desired size (in bytes) of the memory.
- **flags** – *cl_mem_flags* to control the memory.
- **host_ptr** – `void*` to associated with this memory. The meaning of the association depends on the flags. (An integer representation of a pointer is fine).

See also *buffer_from_ndarray()*, *buffer_from_pyarray()*

pycl.**clEnqueueReadBuffer**(*queue*, *mem*, *pointer*, *size=None*, *blocking=True*, *offset=0*, *wait_for=None*)
Read from a *cl_mem* buffer into host memory.

Parameters

- **queue** – *cl_command_queue* to queue it on.

- **mem** – *cl_mem* to read from. Must be a buffer.
- **pointer** – `void*` pointer, the address to start writing into. (An integer representation of the pointer is fine).
- **size** – Number of bytes to read. If not specified, the entire buffer is read out, which might be hazardous if the place you're writing it to isn't big enough.
- **blocking** – Wait for the transfer to complete. Default is True. If False, you can use the returned event to check its status.
- **offset** – Offset in the buffer at which to start reading. Default is 0.
- **wait_for** – *cl_event* (or a list of them) that must complete before the memory transfer will commence.

Returns *cl_event*

See also *buffer_to_ndarray()* and *buffer_to_pyarray()*.

```
>>> ctx = clCreateContext()
>>> queue = clCreateCommandQueue(ctx)
>>> array1 = (cl_int * 8)()  # 32 bytes
>>> for i in range(8): array1[i] = i
>>> m = clCreateBuffer(ctx, 32)
>>> clEnqueueWriteBuffer(queue, m, array1, 32)
<cl_event ...>
>>> array2 = (cl_int * 8)()
>>> clEnqueueReadBuffer(queue, m, array2, 32)
<cl_event ...>
>>> [x.value for x in array2]
[0, 1, 2, 3, 4, 5, 6, 7]
```

`pycl.`**clEnqueueWriteBuffer**(*queue*, *mem*, *pointer*, *size=None*, *blocking=True*, *offset=0*, *wait_for=None*)

Write to a *cl_mem* buffer from a location in host memory.

See *clEnqueueReadBuffer()* for the meanings of the parameters.

`pycl.`**clEnqueueFillBuffer**(*queue*, *mem*, *pattern*, *offset=0*, *size=None*, *wait_for=None*)

Enqueues a command to fill a buffer object with a pattern of a given pattern size.

TODO: Automatically determine pattern_size, perhaps in a wrapper function?

`pycl.`**clGetMemObjectInfo**(*mem*, *param_name*)

Parameters

- **mem** – *cl_mem*
- **param_name** – One of the *cl_mem_info* values.

Memory objects have properties that will retrieve these values for you, so you should probably use those.

class `pycl.`**cl_program**

Represents an OpenCL program, a container for kernels.

Use *clCreateProgramWithSource()* or `clCreateProgramWithBinary()` to make a program.

Remember to call *build()* to compile source programs.

You can retrieve a kernel like so: >>> my_kernel = my_program['my_kernel'] # doctest: +SKIP

Programs participate in reference counting.

build(**args*, ***kw*)
Calls *clBuildProgram()* on the program, passing along any arguments you provide. The program itself will be returned, so you can use this idiom:

```
>>> source = 'kernel void foo(float bar) {}'
>>> ctx = clCreateContext()
>>> prog = clCreateProgramWithSource(ctx, source).build()
```

context
Returns the context the program exists within.

reference_count
Reference count for OpenCL garbage collector.

num_devices
Number of devices the program exists on.

devices
Devices on which the program exists.

source
Program's source code, if available.

binary_sizes
Sizes, in bytes, of the binaries for each of the devices the program is compiled for.

binaries
Acquires the binaries for each device.

build_status(*device=None*)
Retrieves the `cl_program_build_status` for one of more devices. See also *clGetProgramBuildInfo()*

build_options(*device=None*)
Retrieves the build options, as a string, for one of more devices. See also *clGetProgramBuildInfo()*.

build_log(*device=None*)
Returns the build log, as a string, for one or more devices. Mostly useful for checking compiler errors. See also *clGetProgramBuildInfo()*.

pycl.**clGetProgramInfo**(*program*, *param_name*)

Parameters

- **program** – *cl_program*
- **param_name** – One of the *cl_program_info* values.

pycl.**clGetProgramBuildInfo**(*program*, *param_name*, *device=None*)

Parameters

- **program** – The *cl_program* to check.
- **param_name** – One of the *cl_program_build_info* values.
- **device** – A *cl_device* instance, or list of them.

If a list of devices is provided, info will be returned for each of them in a list.

If no device is specified, all devices associated with the program will be used.

The *build_status()*, *build_options()*, and *build_log()* methods of program objects are equivalent to using this, so they may be preferable.

pycl.**clCreateProgramWithSource**(*context*, *source*)

Parameters

- **context** – Context in which the program will exist
- **source** – Source code, as a string.

Remember to call *build()* on the program.

pycl.**clBuildProgram**(*program*, *options=None*, *devices=None*)
Compiles a source program to run on one or more devices.

Parameters

- **program** – The *cl_program* to build.
- **options** – (optional) string with compiler options. See your OpenCL spec and platform provider's docs for possible values.
- **devices** – A list of devices to compile the program for. If not provided, it will be built for all devices in the context.

If the build fails, it will raise a `ProgramBuildFailureError` with details.

class pycl.**cl_kernel**
Represents an OpenCL kernel found in a *cl_program*.

After compiling a program, the kernels will be accessible as items whose keys are the kernel names.

Kernels are reference counted.

name
Name of the kernel function.

program
The *cl_program* this kernel lives in.

context
The *cl_context* this kernel lives in.

num_args
Number of arguments required to call this kernel.

reference_count
Reference count for OpenCL garbage collector.

setarg(*index*, *value=None*, *size=None*)
Sets one of the kernel's arguments.

Parameters

- **index** – 0-based argument number to set.
- **value** – Value to set it to. Can be a *cl_mem*, a Python int or float, or a *localmem* object to indicate local memory allocation.
- **size** – The size of the parameter, in bytes. PyCL will attempt to guess if you don't tell it here or by setting *argtypes*. Guessing is bad.

This does some extra work to try to ensure that the data is in a form suitable for the lower-level *clSetKernelArg()* call. The OpenCL API doesn't give us much help in determining what type an argument should be, so if possible you should set the elements of the kernel's *argtypes* field to a list of types. The types should be either *cl_mem*, *localmem*, a scalar type such as `cl_int`, or a ctypes structure type.

argtypes
Represents the data types of the kernel function arguments. There is no way to ask OpenCL for this information, so short of actually parsing the C code the only way to fill this in is to infer it from the way the user tries to call the kernel.

Since this is error prone, we encourage you to fill in the list yourself.

on(*queue*, **args*, ***kw*)
Enqueue the kernel (hopefully after setting its arguments) upon a command queue. This is essetially a shortcut for *clEnqueueNDRangeKernel()*.

work_group_size(*device=None*)
The maximum size of workgroups for this kernel on the specified device.

compile_work_group_size(*device=None*)
The work group size specified by the kernel source, if any. Otherwise, will return (0,0,0).

local_mem_size(*device=None*)
The amount of local memory that would be used by this kernel on the given device with its current argument set.

preferred_work_group_size_multiple(*device=None*)
Suggests a workgroup size multiplier for each dimension. That is, if a multiple is 8, then workgroup sizes should preferably be multiples of 8.

private_mem_size(*device=None*)
Amount of private memory needed to execute each workitem on the device.

pycl.**clCreateKernel**(*program*, *kernel_name*)

Parameters

- **program** – *cl_program*
- **kernel_name** – String naming a kernel function in the program.

Using the the `program[kernel_name]` syntax is preferable.

pycl.**clGetKernelInfo**(*kernel*, *param_name*)

Parameters

- **kernel** – *cl_kernel*
- **param_name** – One of the *cl_kernel_info* values.

Kernel objects have properties that call this function, so it is probably preferable to use those instead.

pycl.**clGetKernelWorkGroupInfo**(*kernel*, *param_name*, *device=None*)

Parameters

- **kernel** – *cl_kernel*
- **param_name** – One of the *cl_kernel_work_group_info* values.
- **device** – *cl_device*. If no device is specified, the first device in the kernel's context is queried.

Retrieves information about the kernel specific to a particular device that it might be run on. This information is also available through specific methods of kernel objects, which may be preferable to calling this.

class pycl.**localmem**(*size*)
When a kernel defines an argument to be in local memory, no value can be passed in to that argument. Instead,

the size of the local memory is specified. While you could do this directly with *clSetKernelArg()*, localmem allows you to set this using the kernel call syntax. So if you had a kernel whose third argument was a local memory pointer, you could set the arguments like so:

```
>>> mykernel(x, y, localmem(1024))
```

localmem is also accepted in *argtypes*, in which case the kernel can be called using just the desired size:

```
>>> mykernel.argtypes = (cl_mem, cl_mem, localmem)
>>> mykernel(x, y, 1024)
```

pycl.**clSetKernelArg**(*kernel*, *index*, *value=None*, *size=None*)

Parameters

- **kernel** – *cl_kernel*
- **index** – 0-based argument index to set.
- **value** – Should be None or a pointer to a ctypes scalar or a *cl_mem* object. Does not accept *localmem*.
- **size** – Size in bytes of the referenced value. That is, if the argument is a 32-bit integer, this should be 4. If the argument is a *cl_mem*, it should be `sizeof(cl_mem)`.

Unlike most of the wrappers in PyCL, this one doesn't do much to help you out. Use *cl_kernel.setarg()* if you want some help setting individual arguments. Calling the kernel object itself with the desired argument sequence is more preferable still. Set *cl_kernel.argtypes* if it can't guess the types properly.

pycl.**clEnqueueNDRangeKernel**(*queue*, *kernel*, *gsize=(1,)*, *lsize=None*, *offset=None*, *wait_for=None*)

Enqueue a kernel for execution. The kernel's arguments should be set already. For a more idiomatic calling syntax, set the kernel arguments by calling it and use its *on()* method to queue it.

Parameters

- **queue** – *cl_command_queue* to enqueue it upon.
- **kernel** – The *cl_kernel* object you want to run.
- **gsize** – Global work size. A 1-, 2-, or 3-tuple of integers indicating the dimensions of the work to be done. A scalar is fine too. Default is a single work item.
- **lsize** – Local work size. Should have the same dimension as `gsize`. If None (the default), OpenCL will pick a size for you.
- **offset** – Global work item offset. By default, the global id of work items start at 0 in each dimension. Provide a tuple of the same dimension as `gsize` to offset the ids.
- **wait_for** – A *cl_event* or list of them that should complete prior to this kernel's execution.

Returns *cl_event* which will identify when the kernel has completed.

Note that the OpenCL `clEnqueueTask()` function is equivalent to calling this function with the default gsize, lsize, and offset values, so we haven't bothered to wrap it.

pycl.**buffer_from_ndarray**(*queue*, *ary*, *buf=None*, ***kw*)

Creates (or simply writes to) an OpenCL buffer using the contents of a Numpy array.

Parameters

- **queue** – *cl_command_queue* to enqueue the write to.

- **ary** – `numpy.ndarray` object, or other object implementing the array interface. We haven't wrapped the rectangular read/write functions yet, so if the array isn't contiguous, a copy will be made. Note that the entirety of the provided array will be written, so be sure to slice it down to just the part you want to write.
- **buf** – *cl_mem* object. If not provided, one the size of the array will be created. In any event, it should hopefully be large enough to hold the provided array.

Returns `(buf, evt)`, where `evt` is the *cl_event* returned by the write operation.

Any additional provided keyword arguments are passed along to *clEnqueueWriteBuffer()*.

pycl.**buffer_to_ndarray**(*queue*, *buf*, *out=None*, *like=None*, *dtype='uint8'*, *shape=None*, ***kw*)
Reads from an OpenCL buffer into an ndarray.

Parameters

- **queue** – The queue to put the read operation on.
- **buf** – The *cl_mem* to read from
- **out** – The `numpy.ndarray` to read into. If not provided, one will be created based on the following arguments. Unlike `buffer_from_array()`, this must currently be an actual contiguous `ndarray` object.
- **like** – Only relevant if no out array is provided. The new array will have the same shape and dtype as this value.
- **dtype** – Only relevant if no out array or `like` parameter are provided. A `numpy.dtype` or anything that can pass for one. Defaults to `'uint8'`.
- **shape** – Only relevant if no out array or `like` parameter are provided. Integer or tuple determining the array's shape. If no shape is given, the array will be 1d and will have a number of elements based on the buffer's size and the itemsize of the dtype.

Returns `(ary, evt)`, where `evt` is the *cl_event* returned by the read operation.

Any further keyword arguments are passed directly to *clEnqueueReadBuffer()*.

pycl.**buffer_from_pyarray**(*queue*, *ary*, *buf=None*, ***kw*)
Essentially the same as *buffer_from_ndarray()*, except that it accepts arrays from the `array` module in Python's standard library.

pycl.**buffer_to_pyarray**(*queue*, *buf*, *out=None*, *like=None*, *typecode='B'*, *length=None*, ***kw*)
Essentially the same as *buffer_to_ndarray()*, except that it produces arrays from the `array` module in Python's standard library. The `dtype` and `shape` parameters are replaced:

Parameters

- **typecode** – A character indicating the array typecode. See the documentation for the mappings to C data types. The default is 'B', for unsigned bytes.
- **length** – The number of elements that should be in the array. If not provided, it will be determined based on the buffer size and the size of the selected typecode.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

p

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

V

W