

---

# PyCL Documentation

*Release 0.1a*

**Ken Watford**

January 19, 2016



<b>1</b>	<b>Installation</b>	<b>3</b>
<b>2</b>	<b>Module API Reference</b>	<b>5</b>
<b>3</b>	<b>Indices and tables</b>	<b>33</b>
	<b>Python Module Index</b>	<b>35</b>



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](#).



---

## 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](#).





---

## 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 aren'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 be 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 essentially 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 `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.



---

## Indices and tables

---

- `genindex`
- `modindex`
- `search`



**p**

`pyc1`, [5](#)





## A

address\_bits (pycl.cl\_device attribute), 18  
 argtypes (pycl.cl\_kernel attribute), 28  
 available (pycl.cl\_device attribute), 18

## B

base (pycl.cl\_buffer attribute), 25  
 binaries (pycl.cl\_program attribute), 27  
 binary\_sizes (pycl.cl\_program attribute), 27  
 buffer\_from\_ndarray() (in module pycl), 30  
 buffer\_from\_pyarray() (in module pycl), 31  
 buffer\_to\_ndarray() (in module pycl), 31  
 buffer\_to\_pyarray() (in module pycl), 31  
 build() (pycl.cl\_program method), 26  
 build\_log() (pycl.cl\_program method), 27  
 build\_options() (pycl.cl\_program method), 27  
 build\_status() (pycl.cl\_program method), 27  
 built\_in\_kernels (pycl.cl\_device attribute), 18

## C

CL\_A (pycl.cl\_channel\_order attribute), 12  
 CL\_ADDRESS\_CLAMP (pycl.cl\_addressing\_mode attribute), 14  
 CL\_ADDRESS\_CLAMP\_TO\_EDGE (pycl.cl\_addressing\_mode attribute), 14  
 CL\_ADDRESS\_MIRRORED\_REPEAT (pycl.cl\_addressing\_mode attribute), 14  
 CL\_ADDRESS\_NONE (pycl.cl\_addressing\_mode attribute), 14  
 CL\_ADDRESS\_REPEAT (pycl.cl\_addressing\_mode attribute), 14  
 cl\_addressing\_mode (class in pycl), 14  
 CL\_ARGB (pycl.cl\_channel\_order attribute), 12  
 CL\_BGRA (pycl.cl\_channel\_order attribute), 12  
 cl\_buffer (class in pycl), 25  
 cl\_buffer\_create\_type (class in pycl), 14  
 CL\_BUFFER\_CREATE\_TYPE\_REGION (pycl.cl\_buffer\_create\_type attribute), 14  
 cl\_buffer\_region (class in pycl), 17  
 CL\_BUILD\_ERROR (pycl.cl\_build\_status attribute), 15

CL\_BUILD\_IN\_PROGRESS (pycl.cl\_build\_status attribute), 15  
 CL\_BUILD\_NONE (pycl.cl\_build\_status attribute), 15  
 CL\_BUILD\_PROGRAM\_FAILURE (pycl.cl\_errnum attribute), 6  
 cl\_build\_status (class in pycl), 15  
 CL\_BUILD\_SUCCESS (pycl.cl\_build\_status attribute), 15  
 CL\_CGL\_SHAREGROUP\_KHR (pycl.cl\_context\_info attribute), 12  
 cl\_channel\_order (class in pycl), 12  
 cl\_channel\_type (class in pycl), 12  
 CL\_COMMAND\_ACQUIRE\_GL\_OBJECTS (pycl.cl\_command\_type attribute), 16  
 CL\_COMMAND\_COPY\_BUFFER (pycl.cl\_command\_type attribute), 16  
 CL\_COMMAND\_COPY\_BUFFER\_RECT (pycl.cl\_command\_type attribute), 16  
 CL\_COMMAND\_COPY\_BUFFER\_TO\_IMAGE (pycl.cl\_command\_type attribute), 16  
 CL\_COMMAND\_COPY\_IMAGE (pycl.cl\_command\_type attribute), 16  
 CL\_COMMAND\_COPY\_IMAGE\_TO\_BUFFER (pycl.cl\_command\_type attribute), 16  
 cl\_command\_execution\_status (class in pycl), 16  
 CL\_COMMAND\_MAP\_BUFFER (pycl.cl\_command\_type attribute), 16  
 CL\_COMMAND\_MAP\_IMAGE (pycl.cl\_command\_type attribute), 16  
 CL\_COMMAND\_MARKER (pycl.cl\_command\_type attribute), 16  
 CL\_COMMAND\_NATIVE\_KERNEL (pycl.cl\_command\_type attribute), 16  
 CL\_COMMAND\_NDRANGE\_KERNEL (pycl.cl\_command\_type attribute), 16  
 cl\_command\_queue (class in pycl), 24  
 cl\_command\_queue\_info (class in pycl), 12  
 cl\_command\_queue\_properties (class in pycl), 11  
 CL\_COMMAND\_READ\_BUFFER (pycl.cl\_command\_type attribute), 16  
 CL\_COMMAND\_READ\_BUFFER\_RECT

(pycl.cl\_command\_type attribute), 16  
CL\_COMMAND\_READ\_IMAGE (pycl.cl\_command\_type attribute), 16  
CL\_COMMAND\_RELEASE\_GL\_OBJECTS (pycl.cl\_command\_type attribute), 16  
CL\_COMMAND\_TASK (pycl.cl\_command\_type attribute), 16  
cl\_command\_type (class in pycl), 16  
CL\_COMMAND\_UNMAP\_MEM\_OBJECT (pycl.cl\_command\_type attribute), 16  
CL\_COMMAND\_USER (pycl.cl\_command\_type attribute), 16  
CL\_COMMAND\_WRITE\_BUFFER (pycl.cl\_command\_type attribute), 16  
CL\_COMMAND\_WRITE\_BUFFER\_RECT (pycl.cl\_command\_type attribute), 16  
CL\_COMMAND\_WRITE\_IMAGE (pycl.cl\_command\_type attribute), 16  
CL\_COMPILE\_PROGRAM\_FAILURE (pycl.cl\_errnum attribute), 7  
CL\_COMPILER\_NOT\_AVAILABLE (pycl.cl\_errnum attribute), 7  
CL\_COMPLETE (pycl.cl\_command\_execution\_status attribute), 16  
cl\_context (class in pycl), 22  
CL\_CONTEXT\_DEVICES (pycl.cl\_context\_info attribute), 11  
cl\_context\_info (class in pycl), 11  
CL\_CONTEXT\_NUM\_DEVICES (pycl.cl\_context\_info attribute), 12  
CL\_CONTEXT\_PLATFORM (pycl.cl\_context\_info attribute), 12  
cl\_context\_properties (class in pycl), 11  
CL\_CONTEXT\_PROPERTIES (pycl.cl\_context\_info attribute), 11  
CL\_CONTEXT\_PROPERTY\_USE\_CGL\_SHAREGROUP\_APPLE (pycl.cl\_context\_info attribute), 12  
CL\_CONTEXT\_REFERENCE\_COUNT (pycl.cl\_context\_info attribute), 11  
cl\_device (class in pycl), 18  
CL\_DEVICE\_ADDRESS\_BITS (pycl.cl\_device\_info attribute), 9  
cl\_device\_affinity\_domain (class in pycl), 11  
CL\_DEVICE\_AFFINITY\_DOMAIN\_L1\_CACHE (pycl.cl\_device\_affinity\_domain attribute), 11  
CL\_DEVICE\_AFFINITY\_DOMAIN\_L2\_CACHE (pycl.cl\_device\_affinity\_domain attribute), 11  
CL\_DEVICE\_AFFINITY\_DOMAIN\_L3\_CACHE (pycl.cl\_device\_affinity\_domain attribute), 11  
CL\_DEVICE\_AFFINITY\_DOMAIN\_L4\_CACHE (pycl.cl\_device\_affinity\_domain attribute), 11  
CL\_DEVICE\_AFFINITY\_DOMAIN\_NEXT\_PARTITIONABLE (pycl.cl\_device\_affinity\_domain attribute), 11  
CL\_DEVICE\_AFFINITY\_DOMAIN\_NUMA (pycl.cl\_device\_affinity\_domain attribute), 11  
CL\_DEVICE\_AVAILABLE (pycl.cl\_device\_info attribute), 9  
CL\_DEVICE\_BUILT\_IN\_KERNELS (pycl.cl\_device\_info attribute), 9  
CL\_DEVICE\_COMPILER\_AVAILABLE (pycl.cl\_device\_info attribute), 9  
CL\_DEVICE\_DOUBLE\_FP\_CONFIG (pycl.cl\_device\_info attribute), 8  
CL\_DEVICE\_ENDIAN\_LITTLE (pycl.cl\_device\_info attribute), 8  
CL\_DEVICE\_ERROR\_CORRECTION\_SUPPORT (pycl.cl\_device\_info attribute), 10  
cl\_device\_exec\_capabilities (class in pycl), 11  
CL\_DEVICE\_EXECUTION\_CAPABILITIES (pycl.cl\_device\_info attribute), 10  
CL\_DEVICE\_EXTENSIONS (pycl.cl\_device\_info attribute), 9  
cl\_device\_fp\_config (class in pycl), 10  
CL\_DEVICE\_GLOBAL\_MEM\_CACHE\_SIZE (pycl.cl\_device\_info attribute), 9  
CL\_DEVICE\_GLOBAL\_MEM\_CACHE\_TYPE (pycl.cl\_device\_info attribute), 9  
CL\_DEVICE\_GLOBAL\_MEM\_CACHESIZE (pycl.cl\_device\_info attribute), 8  
CL\_DEVICE\_GLOBAL\_MEM\_SIZE (pycl.cl\_device\_info attribute), 8  
CL\_DEVICE\_HALF\_FP\_CONFIG (pycl.cl\_device\_info attribute), 10  
CL\_DEVICE\_HOST\_UNIFIED\_MEMORY (pycl.cl\_device\_info attribute), 9  
CL\_DEVICE\_IMAGE2D\_MAX\_HEIGHT (pycl.cl\_device\_info attribute), 8  
CL\_DEVICE\_IMAGE2D\_MAX\_WIDTH (pycl.cl\_device\_info attribute), 9  
CL\_DEVICE\_IMAGE3D\_MAX\_DEPTH (pycl.cl\_device\_info attribute), 9  
CL\_DEVICE\_IMAGE3D\_MAX\_HEIGHT (pycl.cl\_device\_info attribute), 9  
CL\_DEVICE\_IMAGE3D\_MAX\_WIDTH (pycl.cl\_device\_info attribute), 9  
CL\_DEVICE\_IMAGE\_BASE\_ADDRESS\_ALIGNMENT (pycl.cl\_device\_info attribute), 9  
CL\_DEVICE\_IMAGE\_MAX\_ARRAY\_SIZE (pycl.cl\_device\_info attribute), 9  
CL\_DEVICE\_IMAGE\_MAX\_BUFFER\_SIZE (pycl.cl\_device\_info attribute), 8  
CL\_DEVICE\_IMAGE\_PITCH\_ALIGNMENT (pycl.cl\_device\_info attribute), 8  
CL\_DEVICE\_IMAGE\_SUPPORT (pycl.cl\_device\_info attribute), 8  
cl\_device\_info (class in pycl), 8  
CL\_DEVICE\_LINKER\_AVAILABLE

- (pycl.cl\_device\_info attribute), 9
- CL\_DEVICE\_LOCAL\_MEM\_SIZE (pycl.cl\_device\_info attribute), 9
- cl\_device\_local\_mem\_type (class in pycl), 10
- CL\_DEVICE\_LOCAL\_MEM\_TYPE (pycl.cl\_device\_info attribute), 10
- CL\_DEVICE\_MAX\_CLOCK\_FREQUENCY (pycl.cl\_device\_info attribute), 10
- CL\_DEVICE\_MAX\_COMPUTE\_UNITS (pycl.cl\_device\_info attribute), 8
- CL\_DEVICE\_MAX\_CONSTANT\_ARGS (pycl.cl\_device\_info attribute), 9
- CL\_DEVICE\_MAX\_CONSTANT\_BUFFER\_SIZE (pycl.cl\_device\_info attribute), 8
- CL\_DEVICE\_MAX\_MEM\_ALLOC\_SIZE (pycl.cl\_device\_info attribute), 10
- CL\_DEVICE\_MAX\_PARAMETER\_SIZE (pycl.cl\_device\_info attribute), 10
- CL\_DEVICE\_MAX\_READ\_IMAGE\_ARGS (pycl.cl\_device\_info attribute), 9
- CL\_DEVICE\_MAX\_SAMPLERS (pycl.cl\_device\_info attribute), 10
- CL\_DEVICE\_MAX\_WORK\_GROUP\_SIZE (pycl.cl\_device\_info attribute), 9
- CL\_DEVICE\_MAX\_WORK\_ITEM\_DIMENSIONS (pycl.cl\_device\_info attribute), 8
- CL\_DEVICE\_MAX\_WORK\_ITEM\_SIZES (pycl.cl\_device\_info attribute), 10
- CL\_DEVICE\_MAX\_WRITE\_IMAGE\_ARGS (pycl.cl\_device\_info attribute), 10
- CL\_DEVICE\_MEM\_BASE\_ADDR\_ALIGN (pycl.cl\_device\_info attribute), 8
- cl\_device\_mem\_cache\_type (class in pycl), 10
- CL\_DEVICE\_MIN\_DATA\_TYPE\_ALIGN\_SIZE (pycl.cl\_device\_info attribute), 8
- CL\_DEVICE\_NAME (pycl.cl\_device\_info attribute), 10
- CL\_DEVICE\_NATIVE\_VECTOR\_WIDTH\_CHAR (pycl.cl\_device\_info attribute), 9
- CL\_DEVICE\_NATIVE\_VECTOR\_WIDTH\_DOUBLE (pycl.cl\_device\_info attribute), 9
- CL\_DEVICE\_NATIVE\_VECTOR\_WIDTH\_FLOAT (pycl.cl\_device\_info attribute), 9
- CL\_DEVICE\_NATIVE\_VECTOR\_WIDTH\_HALF (pycl.cl\_device\_info attribute), 8
- CL\_DEVICE\_NATIVE\_VECTOR\_WIDTH\_INT (pycl.cl\_device\_info attribute), 9
- CL\_DEVICE\_NATIVE\_VECTOR\_WIDTH\_LONG (pycl.cl\_device\_info attribute), 8
- CL\_DEVICE\_NATIVE\_VECTOR\_WIDTH\_SHORT (pycl.cl\_device\_info attribute), 9
- CL\_DEVICE\_NOT\_AVAILABLE (pycl.cl\_errnum attribute), 6
- CL\_DEVICE\_NOT\_FOUND (pycl.cl\_errnum attribute), 7
- CL\_DEVICE\_OPENCL\_C\_VERSION (pycl.cl\_device\_info attribute), 10
- CL\_DEVICE\_PARENT\_DEVICE (pycl.cl\_device\_info attribute), 9
- CL\_DEVICE\_PARTITION\_AFFINITY\_DOMAIN (pycl.cl\_device\_info attribute), 8
- CL\_DEVICE\_PARTITION\_BY\_AFFINITY\_DOMAIN (pycl.cl\_device\_partition\_property attribute), 11
- CL\_DEVICE\_PARTITION\_BY\_COUNTS (pycl.cl\_device\_partition\_property attribute), 11
- CL\_DEVICE\_PARTITION\_BY\_COUNTS\_LIST\_END (pycl.cl\_device\_partition\_property attribute), 11
- CL\_DEVICE\_PARTITION\_EQUALLY (pycl.cl\_device\_partition\_property attribute), 11
- CL\_DEVICE\_PARTITION\_FAILED (pycl.cl\_errnum attribute), 7
- CL\_DEVICE\_PARTITION\_MAX\_SUB\_DEVICES (pycl.cl\_device\_info attribute), 9
- CL\_DEVICE\_PARTITION\_PROPERTIES (pycl.cl\_device\_info attribute), 10
- cl\_device\_partition\_property (class in pycl), 11
- CL\_DEVICE\_PARTITION\_TYPE (pycl.cl\_device\_info attribute), 9
- CL\_DEVICE\_PLATFORM (pycl.cl\_device\_info attribute), 9
- CL\_DEVICE\_PREFERRED\_INTEROP\_USER\_SYNC (pycl.cl\_device\_info attribute), 10
- CL\_DEVICE\_PREFERRED\_VECTOR\_WIDTH\_CHAR (pycl.cl\_device\_info attribute), 9
- CL\_DEVICE\_PREFERRED\_VECTOR\_WIDTH\_DOUBLE (pycl.cl\_device\_info attribute), 9
- CL\_DEVICE\_PREFERRED\_VECTOR\_WIDTH\_FLOAT (pycl.cl\_device\_info attribute), 10
- CL\_DEVICE\_PREFERRED\_VECTOR\_WIDTH\_HALF (pycl.cl\_device\_info attribute), 8
- CL\_DEVICE\_PREFERRED\_VECTOR\_WIDTH\_INT (pycl.cl\_device\_info attribute), 8
- CL\_DEVICE\_PREFERRED\_VECTOR\_WIDTH\_LONG (pycl.cl\_device\_info attribute), 8
- CL\_DEVICE\_PREFERRED\_VECTOR\_WIDTH\_SHORT (pycl.cl\_device\_info attribute), 8
- CL\_DEVICE\_PRINTF\_BUFFER\_SIZE (pycl.cl\_device\_info attribute), 9
- CL\_DEVICE\_PROFILE (pycl.cl\_device\_info attribute), 9
- CL\_DEVICE\_PROFILING\_TIMER\_RESOLUTION (pycl.cl\_device\_info attribute), 8
- CL\_DEVICE\_QUEUE\_PROPERTIES (pycl.cl\_device\_info attribute), 9
- CL\_DEVICE\_REFERENCE\_COUNT

(pycl.cl\_device\_info attribute), 8  
CL\_DEVICE\_SINGLE\_FP\_CONFIG  
(pycl.cl\_device\_info attribute), 9  
cl\_device\_type (class in pycl), 6  
CL\_DEVICE\_TYPE (pycl.cl\_device\_info attribute), 9  
CL\_DEVICE\_TYPE\_ACCELERATOR  
(pycl.cl\_device\_type attribute), 6  
CL\_DEVICE\_TYPE\_ALL (pycl.cl\_device\_type attribute), 6  
CL\_DEVICE\_TYPE\_CPU (pycl.cl\_device\_type attribute), 6  
CL\_DEVICE\_TYPE\_DEFAULT (pycl.cl\_device\_type attribute), 6  
CL\_DEVICE\_TYPE\_GPU (pycl.cl\_device\_type attribute), 6  
CL\_DEVICE\_VENDOR (pycl.cl\_device\_info attribute), 8  
CL\_DEVICE\_VENDOR\_ID (pycl.cl\_device\_info attribute), 10  
CL\_DEVICE\_VERSION (pycl.cl\_device\_info attribute), 9  
CL\_DRIVER\_VERSION (pycl.cl\_device\_info attribute), 8  
CL\_EGL\_DISPLAY\_KHR (pycl.cl\_context\_info attribute), 12  
cl\_errnum (class in pycl), 6  
cl\_event (class in pycl), 17  
CL\_EVENT\_COMMAND\_EXECUTION\_STATUS  
(pycl.cl\_event\_info attribute), 15  
CL\_EVENT\_COMMAND\_QUEUE (pycl.cl\_event\_info attribute), 15  
CL\_EVENT\_COMMAND\_TYPE (pycl.cl\_event\_info attribute), 15  
CL\_EVENT\_CONTEXT (pycl.cl\_event\_info attribute), 15  
cl\_event\_info (class in pycl), 15  
CL\_EVENT\_REFERENCE\_COUNT  
(pycl.cl\_event\_info attribute), 15  
CL\_EXEC\_KERNEL (pycl.cl\_device\_exec\_capabilities attribute), 11  
CL\_EXEC\_NATIVE\_KERNEL  
(pycl.cl\_device\_exec\_capabilities attribute), 11  
CL\_EXEC\_STATUS\_ERROR\_FOR\_EVENTS\_IN\_WAIT\_LIST  
(pycl.cl\_errnum attribute), 7  
CL\_FILTER\_LINEAR (pycl.cl\_filter\_mode attribute), 14  
cl\_filter\_mode (class in pycl), 14  
CL\_FILTER\_NEAREST (pycl.cl\_filter\_mode attribute), 14  
CL\_FLOAT (pycl.cl\_channel\_type attribute), 13  
CL\_FP\_DENORM (pycl.cl\_device\_fp\_config attribute), 10  
CL\_FP\_FMA (pycl.cl\_device\_fp\_config attribute), 10  
CL\_FP\_INF\_NAN (pycl.cl\_device\_fp\_config attribute), 10  
CL\_FP\_ROUND\_TO\_INF (pycl.cl\_device\_fp\_config attribute), 10  
CL\_FP\_ROUND\_TO\_NEAREST  
(pycl.cl\_device\_fp\_config attribute), 10  
CL\_FP\_ROUND\_TO\_ZERO (pycl.cl\_device\_fp\_config attribute), 10  
CL\_FP\_SOFT\_FLOAT (pycl.cl\_device\_fp\_config attribute), 10  
CL\_GL\_CONTEXT\_KHR (pycl.cl\_context\_info attribute), 11  
CL\_GLOBAL (pycl.cl\_device\_local\_mem\_type attribute), 10  
CL\_GLX\_DISPLAY\_KHR (pycl.cl\_context\_info attribute), 11  
CL\_HALF\_FLOAT (pycl.cl\_channel\_type attribute), 13  
cl\_image (class in pycl), 25  
CL\_IMAGE\_DEPTH (pycl.cl\_image\_info attribute), 14  
CL\_IMAGE\_ELEMENT\_SIZE (pycl.cl\_image\_info attribute), 14  
cl\_image\_format (class in pycl), 16  
CL\_IMAGE\_FORMAT (pycl.cl\_image\_info attribute), 14  
CL\_IMAGE\_FORMAT\_MISMATCH (pycl.cl\_errnum attribute), 7  
CL\_IMAGE\_FORMAT\_NOT\_SUPPORTED  
(pycl.cl\_errnum attribute), 8  
CL\_IMAGE\_HEIGHT (pycl.cl\_image\_info attribute), 14  
cl\_image\_info (class in pycl), 13  
CL\_IMAGE\_ROW\_PITCH (pycl.cl\_image\_info attribute), 14  
CL\_IMAGE\_SLICE\_PITCH (pycl.cl\_image\_info attribute), 14  
CL\_IMAGE\_WIDTH (pycl.cl\_image\_info attribute), 14  
CL\_INTENSITY (pycl.cl\_channel\_order attribute), 12  
CL\_INVALID\_ARG\_INDEX (pycl.cl\_errnum attribute), 8  
CL\_INVALID\_ARG\_SIZE (pycl.cl\_errnum attribute), 7  
CL\_INVALID\_ARG\_VALUE (pycl.cl\_errnum attribute), 7  
CL\_INVALID\_BINARY (pycl.cl\_errnum attribute), 7  
CL\_INVALID\_BUFFER\_SIZE (pycl.cl\_errnum attribute), 7  
CL\_INVALID\_BUILD\_OPTIONS (pycl.cl\_errnum attribute), 7  
CL\_INVALID\_COMMAND\_QUEUE (pycl.cl\_errnum attribute), 6  
CL\_INVALID\_CONTEXT (pycl.cl\_errnum attribute), 7  
CL\_INVALID\_DEVICE (pycl.cl\_errnum attribute), 7  
CL\_INVALID\_DEVICE\_TYPE (pycl.cl\_errnum attribute), 7  
CL\_INVALID\_EVENT (pycl.cl\_errnum attribute), 6  
CL\_INVALID\_EVENT\_WAIT\_LIST (pycl.cl\_errnum attribute), 7  
CL\_INVALID\_GL\_OBJECT (pycl.cl\_errnum attribute),

- 7
- CL\_INVALID\_GL\_SHAREGROUP\_REFERENCE\_KHR  
(pycl.cl\_errnum attribute), 6
- CL\_INVALID\_GLOBAL\_OFFSET (pycl.cl\_errnum attribute), 8
- CL\_INVALID\_GLOBAL\_WORK\_SIZE  
(pycl.cl\_errnum attribute), 7
- CL\_INVALID\_HOST\_PTR (pycl.cl\_errnum attribute), 7
- CL\_INVALID\_IMAGE\_FORMAT\_DESCRIPTOR  
(pycl.cl\_errnum attribute), 7
- CL\_INVALID\_IMAGE\_SIZE (pycl.cl\_errnum attribute), 6
- CL\_INVALID\_KERNEL (pycl.cl\_errnum attribute), 7
- CL\_INVALID\_KERNEL\_ARGS (pycl.cl\_errnum attribute), 7
- CL\_INVALID\_KERNEL\_DEFINITION  
(pycl.cl\_errnum attribute), 6
- CL\_INVALID\_KERNEL\_NAME (pycl.cl\_errnum attribute), 6
- CL\_INVALID\_MEM\_OBJECT (pycl.cl\_errnum attribute), 7
- CL\_INVALID\_MIP\_LEVEL (pycl.cl\_errnum attribute), 7
- CL\_INVALID\_OPERATION (pycl.cl\_errnum attribute), 7
- CL\_INVALID\_PLATFORM (pycl.cl\_errnum attribute), 7
- CL\_INVALID\_PROGRAM (pycl.cl\_errnum attribute), 6
- CL\_INVALID\_PROGRAM\_EXECUTABLE  
(pycl.cl\_errnum attribute), 6
- CL\_INVALID\_PROPERTY (pycl.cl\_errnum attribute), 7
- CL\_INVALID\_QUEUE\_PROPERTIES (pycl.cl\_errnum attribute), 6
- CL\_INVALID\_SAMPLER (pycl.cl\_errnum attribute), 7
- CL\_INVALID\_VALUE (pycl.cl\_errnum attribute), 6
- CL\_INVALID\_WORK\_DIMENSION (pycl.cl\_errnum attribute), 6
- CL\_INVALID\_WORK\_GROUP\_SIZE (pycl.cl\_errnum attribute), 6
- CL\_INVALID\_WORK\_ITEM\_SIZE (pycl.cl\_errnum attribute), 7
- cl\_kernel (class in pycl), 28
- CL\_KERNEL\_ARG\_INFO\_NOT\_AVAILABLE  
(pycl.cl\_errnum attribute), 7
- CL\_KERNEL\_COMPILE\_WORK\_GROUP\_SIZE  
(pycl.cl\_kernel\_work\_group\_info attribute), 15
- CL\_KERNEL\_CONTEXT (pycl.cl\_kernel\_info attribute), 15
- CL\_KERNEL\_FUNCTION\_NAME  
(pycl.cl\_kernel\_info attribute), 15
- cl\_kernel\_info (class in pycl), 15
- CL\_KERNEL\_LOCAL\_MEM\_SIZE  
(pycl.cl\_kernel\_work\_group\_info attribute), 15
- CL\_KERNEL\_NUM\_ARGS (pycl.cl\_kernel\_info attribute), 15
- CL\_KERNEL\_PREFERRED\_WORK\_GROUP\_SIZE\_MULTIPLE  
(pycl.cl\_kernel\_work\_group\_info attribute), 15
- CL\_KERNEL\_PRIVATE\_MEM\_SIZE  
(pycl.cl\_kernel\_work\_group\_info attribute), 15
- CL\_KERNEL\_PROGRAM (pycl.cl\_kernel\_info attribute), 15
- CL\_KERNEL\_REFERENCE\_COUNT  
(pycl.cl\_kernel\_info attribute), 15
- cl\_kernel\_work\_group\_info (class in pycl), 15
- CL\_KERNEL\_WORK\_GROUP\_SIZE  
(pycl.cl\_kernel\_work\_group\_info attribute), 15
- CL\_LINK\_PROGRAM\_FAILURE (pycl.cl\_errnum attribute), 8
- CL\_LINKER\_NOT\_AVAILABLE (pycl.cl\_errnum attribute), 7
- CL\_LOCAL (pycl.cl\_device\_local\_mem\_type attribute), 10
- CL\_LUMINANCE (pycl.cl\_channel\_order attribute), 12
- CL\_MAP\_FAILURE (pycl.cl\_errnum attribute), 7
- cl\_map\_flags (class in pycl), 14
- CL\_MAP\_READ (pycl.cl\_map\_flags attribute), 14
- CL\_MAP\_WRITE (pycl.cl\_map\_flags attribute), 14
- cl\_mem (class in pycl), 24
- CL\_MEM\_ALLOC\_HOST\_PTR (pycl.cl\_mem\_flags attribute), 13
- CL\_MEM\_ASSOCIATED\_MEMOBJECT  
(pycl.cl\_mem\_info attribute), 13
- CL\_MEM\_CONTEXT (pycl.cl\_mem\_info attribute), 13
- CL\_MEM\_COPY\_HOST\_PTR (pycl.cl\_mem\_flags attribute), 13
- CL\_MEM\_COPY\_OVERLAP (pycl.cl\_errnum attribute), 6
- cl\_mem\_flags (class in pycl), 13
- CL\_MEM\_FLAGS (pycl.cl\_mem\_info attribute), 13
- CL\_MEM\_HOST\_PTR (pycl.cl\_mem\_info attribute), 13
- cl\_mem\_info (class in pycl), 13
- CL\_MEM\_MAP\_COUNT (pycl.cl\_mem\_info attribute), 13
- cl\_mem\_migration\_flags (class in pycl), 13
- CL\_MEM\_OBJECT\_ALLOCATION\_FAILURE  
(pycl.cl\_errnum attribute), 7
- CL\_MEM\_OBJECT\_BUFFER  
(pycl.cl\_mem\_object\_type attribute), 13
- CL\_MEM\_OBJECT\_IMAGE2D  
(pycl.cl\_mem\_object\_type attribute), 13
- CL\_MEM\_OBJECT\_IMAGE3D  
(pycl.cl\_mem\_object\_type attribute), 13
- cl\_mem\_object\_type (class in pycl), 13
- CL\_MEM\_OFFSET (pycl.cl\_mem\_info attribute), 13
- CL\_MEM\_READ\_ONLY (pycl.cl\_mem\_flags attribute), 13
- CL\_MEM\_READ\_WRITE (pycl.cl\_mem\_flags attribute), 13
- CL\_MEM\_REFERENCE\_COUNT (pycl.cl\_mem\_info



attribute), 13  
CL\_MEM\_SIZE (pycl.cl\_mem\_info attribute), 13  
CL\_MEM\_TYPE (pycl.cl\_mem\_info attribute), 13  
CL\_MEM\_USE\_HOST\_PTR (pycl.cl\_mem\_flags attribute), 13  
CL\_MEM\_WRITE\_ONLY (pycl.cl\_mem\_flags attribute), 13  
CL\_MIGRATE\_MEM\_OBJECT\_CONTENT\_UNDEFINED (pycl.cl\_mem\_migration\_flags attribute), 13  
CL\_MIGRATE\_MEM\_OBJECT\_HOST (pycl.cl\_mem\_migration\_flags attribute), 13  
CL\_MISALIGNED\_SUB\_BUFFER\_OFFSET (pycl.cl\_errnum attribute), 7  
CL\_NONE (pycl.cl\_device\_mem\_cache\_type attribute), 10  
CL\_OUT\_OF\_HOST\_MEMORY (pycl.cl\_errnum attribute), 7  
CL\_OUT\_OF\_RESOURCES (pycl.cl\_errnum attribute), 7  
cl\_platform (class in pycl), 17  
CL\_PLATFORM\_EXTENSIONS (pycl.cl\_platform\_info attribute), 8  
cl\_platform\_info (class in pycl), 8  
CL\_PLATFORM\_NAME (pycl.cl\_platform\_info attribute), 8  
CL\_PLATFORM\_PROFILE (pycl.cl\_platform\_info attribute), 8  
CL\_PLATFORM\_VENDOR (pycl.cl\_platform\_info attribute), 8  
CL\_PLATFORM\_VERSION (pycl.cl\_platform\_info attribute), 8  
CL\_PROFILING\_COMMAND\_END (pycl.cl\_profiling\_info attribute), 16  
CL\_PROFILING\_COMMAND\_QUEUED (pycl.cl\_profiling\_info attribute), 16  
CL\_PROFILING\_COMMAND\_START (pycl.cl\_profiling\_info attribute), 16  
CL\_PROFILING\_COMMAND\_SUBMIT (pycl.cl\_profiling\_info attribute), 16  
cl\_profiling\_info (class in pycl), 16  
CL\_PROFILING\_INFO\_NOT\_AVAILABLE (pycl.cl\_errnum attribute), 6  
cl\_program (class in pycl), 26  
CL\_PROGRAM\_BINARIES (pycl.cl\_program\_info attribute), 15  
CL\_PROGRAM\_BINARY\_SIZES (pycl.cl\_program\_info attribute), 15  
cl\_program\_build\_info (class in pycl), 15  
CL\_PROGRAM\_BUILD\_LOG (pycl.cl\_program\_build\_info attribute), 15  
CL\_PROGRAM\_BUILD\_OPTIONS (pycl.cl\_program\_build\_info attribute), 15  
CL\_PROGRAM\_BUILD\_STATUS (pycl.cl\_program\_build\_info attribute), 15  
CL\_PROGRAM\_CONTEXT (pycl.cl\_program\_info attribute), 15  
CL\_PROGRAM\_DEVICES (pycl.cl\_program\_info attribute), 15  
cl\_program\_info (class in pycl), 14  
CL\_PROGRAM\_NUM\_DEVICES (pycl.cl\_program\_info attribute), 15  
CL\_PROGRAM\_REFERENCE\_COUNT (pycl.cl\_program\_info attribute), 15  
CL\_PROGRAM\_SOURCE (pycl.cl\_program\_info attribute), 15  
CL\_QUEUE\_CONTEXT (pycl.cl\_command\_queue\_info attribute), 12  
CL\_QUEUE\_DEVICE (pycl.cl\_command\_queue\_info attribute), 12  
CL\_QUEUE\_OUT\_OF\_ORDER\_EXEC\_MODE\_ENABLE (pycl.cl\_command\_queue\_properties attribute), 11  
CL\_QUEUE\_PROFILING\_ENABLE (pycl.cl\_command\_queue\_properties attribute), 11  
CL\_QUEUE\_PROPERTIES (pycl.cl\_command\_queue\_info attribute), 12  
CL\_QUEUE\_REFERENCE\_COUNT (pycl.cl\_command\_queue\_info attribute), 12  
CL\_QUEUED (pycl.cl\_command\_execution\_status attribute), 16  
CL\_R (pycl.cl\_channel\_order attribute), 12  
CL\_RA (pycl.cl\_channel\_order attribute), 12  
CL\_READ\_ONLY\_CACHE (pycl.cl\_device\_mem\_cache\_type attribute), 10  
CL\_READ\_WRITE\_CACHE (pycl.cl\_device\_mem\_cache\_type attribute), 10  
CL\_RG (pycl.cl\_channel\_order attribute), 12  
CL\_RGB (pycl.cl\_channel\_order attribute), 12  
CL\_RGBA (pycl.cl\_channel\_order attribute), 12  
CL\_RGBx (pycl.cl\_channel\_order attribute), 12  
CL\_RGx (pycl.cl\_channel\_order attribute), 12  
CL\_RUNNING (pycl.cl\_command\_execution\_status attribute), 16  
CL\_Rx (pycl.cl\_channel\_order attribute), 12  
CL\_SAMPLER\_ADDRESSING\_MODE (pycl.cl\_sampler\_info attribute), 14  
CL\_SAMPLER\_CONTEXT (pycl.cl\_sampler\_info attribute), 14  
CL\_SAMPLER\_FILTER\_MODE (pycl.cl\_sampler\_info attribute), 14  
cl\_sampler\_info (class in pycl), 14  
CL\_SAMPLER\_NORMALIZED\_COORDS (pycl.cl\_sampler\_info attribute), 14  
CL\_SAMPLER\_REFERENCE\_COUNT

(pycl.cl\_sampler\_info attribute), 14

CL\_SIGNED\_INT16 (pycl.cl\_channel\_type attribute), 13

CL\_SIGNED\_INT32 (pycl.cl\_channel\_type attribute), 12

CL\_SIGNED\_INT8 (pycl.cl\_channel\_type attribute), 12

CL\_SNORM\_INT16 (pycl.cl\_channel\_type attribute), 12

CL\_SNORM\_INT8 (pycl.cl\_channel\_type attribute), 12

CL\_SUBMITTED (pycl.cl\_command\_execution\_status attribute), 16

CL\_SUCCESS (pycl.cl\_errnum attribute), 7

CL\_UNORM\_INT16 (pycl.cl\_channel\_type attribute), 12

CL\_UNORM\_INT8 (pycl.cl\_channel\_type attribute), 13

CL\_UNORM\_INT\_101010 (pycl.cl\_channel\_type attribute), 12

CL\_UNORM\_SHORT\_555 (pycl.cl\_channel\_type attribute), 12

CL\_UNORM\_SHORT\_565 (pycl.cl\_channel\_type attribute), 12

CL\_UNSIGNED\_INT16 (pycl.cl\_channel\_type attribute), 13

CL\_UNSIGNED\_INT32 (pycl.cl\_channel\_type attribute), 13

CL\_UNSIGNED\_INT8 (pycl.cl\_channel\_type attribute), 13

CL\_WGL\_HDC\_KHR (pycl.cl\_context\_info attribute), 12

clBuildProgram() (in module pycl), 28

clCreateBuffer() (in module pycl), 25

clCreateCommandQueue() (in module pycl), 24

clCreateContext() (in module pycl), 23

clCreateContextFromType() (in module pycl), 23

clCreateKernel() (in module pycl), 29

clCreateProgramWithSource() (in module pycl), 27

clEnqueueFillBuffer() (in module pycl), 26

clEnqueueNDRangeKernel() (in module pycl), 30

clEnqueueReadBuffer() (in module pycl), 25

clEnqueueWriteBuffer() (in module pycl), 26

clGetCommandQueueInfo() (in module pycl), 24

clGetContextInfo() (in module pycl), 23

clGetDeviceIDs() (in module pycl), 22

clGetDeviceInfo() (in module pycl), 22

clGetEventInfo() (in module pycl), 17

clGetKernelInfo() (in module pycl), 29

clGetKernelWorkGroupInfo() (in module pycl), 29

clGetMemObjectInfo() (in module pycl), 26

clGetPlatformIDs() (in module pycl), 18

clGetPlatformInfo() (in module pycl), 18

clGetProgramBuildInfo() (in module pycl), 27

clGetProgramInfo() (in module pycl), 27

clSetKernelArg() (in module pycl), 30

clWaitForEvents() (in module pycl), 17

compile\_work\_group\_size() (pycl.cl\_kernel method), 29

compiler\_available (pycl.cl\_device attribute), 18

context (pycl.cl\_command\_queue attribute), 24

context (pycl.cl\_event attribute), 17

context (pycl.cl\_kernel attribute), 28

context (pycl.cl\_mem attribute), 25

context (pycl.cl\_program attribute), 27

## D

device (pycl.cl\_command\_queue attribute), 24

devices (pycl.cl\_context attribute), 23

devices (pycl.cl\_platform attribute), 18

devices (pycl.cl\_program attribute), 27

double\_fp\_config (pycl.cl\_device attribute), 18

## E

empty\_like\_this() (pycl.cl\_mem method), 25

endian\_little (pycl.cl\_device attribute), 18

error\_correction\_support (pycl.cl\_device attribute), 19

execution\_capabilities (pycl.cl\_device attribute), 19

extensions (pycl.cl\_platform attribute), 18

## F

flags (pycl.cl\_mem attribute), 25

## G

global\_mem\_cache\_size (pycl.cl\_device attribute), 19

global\_mem\_cache\_type (pycl.cl\_device attribute), 19

global\_mem\_cacheline\_size (pycl.cl\_device attribute), 19

global\_mem\_size (pycl.cl\_device attribute), 19

## H

half\_fp\_config (pycl.cl\_device attribute), 19

host\_unified\_memory (pycl.cl\_device attribute), 19

hostptr (pycl.cl\_mem attribute), 25

## I

image2d\_max\_height (pycl.cl\_device attribute), 19

image2d\_max\_width (pycl.cl\_device attribute), 19

image3d\_max\_depth (pycl.cl\_device attribute), 19

image3d\_max\_height (pycl.cl\_device attribute), 19

image3d\_max\_width (pycl.cl\_device attribute), 19

image\_base\_address\_alignment (pycl.cl\_device attribute), 19

image\_channel\_data\_type (pycl.cl\_image\_format attribute), 17

image\_channel\_order (pycl.cl\_image\_format attribute), 17

image\_max\_array\_size (pycl.cl\_device attribute), 19

image\_max\_buffer\_size (pycl.cl\_device attribute), 19

image\_pitch\_alignment (pycl.cl\_device attribute), 19

image\_support (pycl.cl\_device attribute), 19

## L

linker\_available (pycl.cl\_device attribute), 19

local\_mem\_size (pycl.cl\_device attribute), 19

local\_mem\_size() (pycl.cl\_kernel method), 29  
local\_mem\_type (pycl.cl\_device attribute), 19  
localmem (class in pycl), 29

## M

map\_count (pycl.cl\_mem attribute), 25  
max\_clock\_frequency (pycl.cl\_device attribute), 20  
max\_compute\_units (pycl.cl\_device attribute), 20  
max\_constant\_args (pycl.cl\_device attribute), 20  
max\_constant\_buffer\_size (pycl.cl\_device attribute), 20  
max\_mem\_alloc\_size (pycl.cl\_device attribute), 20  
max\_parameter\_size (pycl.cl\_device attribute), 20  
max\_read\_image\_args (pycl.cl\_device attribute), 20  
max\_samplers (pycl.cl\_device attribute), 20  
max\_work\_group\_size (pycl.cl\_device attribute), 20  
max\_work\_item\_dimensions (pycl.cl\_device attribute), 20  
max\_work\_item\_sizes (pycl.cl\_device attribute), 20  
max\_write\_image\_args (pycl.cl\_device attribute), 20  
mem\_base\_addr\_align (pycl.cl\_device attribute), 20  
min\_data\_type\_align\_size (pycl.cl\_device attribute), 20

## N

name (pycl.cl\_device attribute), 20  
name (pycl.cl\_kernel attribute), 28  
name (pycl.cl\_platform attribute), 17  
native\_vector\_width\_char (pycl.cl\_device attribute), 20  
native\_vector\_width\_double (pycl.cl\_device attribute), 20  
native\_vector\_width\_float (pycl.cl\_device attribute), 20  
native\_vector\_width\_half (pycl.cl\_device attribute), 20  
native\_vector\_width\_int (pycl.cl\_device attribute), 20  
native\_vector\_width\_long (pycl.cl\_device attribute), 20  
native\_vector\_width\_short (pycl.cl\_device attribute), 21  
num\_args (pycl.cl\_kernel attribute), 28  
num\_devices (pycl.cl\_context attribute), 23  
num\_devices (pycl.cl\_program attribute), 27

## O

offset (pycl.cl\_buffer attribute), 25  
on() (pycl.cl\_kernel method), 29  
opencl\_c\_version (pycl.cl\_device attribute), 21  
OpenCLError, 17

## P

parent\_device (pycl.cl\_device attribute), 21  
partition\_affinity\_domain (pycl.cl\_device attribute), 21  
partition\_max\_sub\_devices (pycl.cl\_device attribute), 21  
partition\_properties (pycl.cl\_device attribute), 21  
partition\_type (pycl.cl\_device attribute), 21  
platform (pycl.cl\_context attribute), 23  
platform (pycl.cl\_device attribute), 21  
preferred\_interop\_user\_sync (pycl.cl\_device attribute), 21

preferred\_vector\_width\_char (pycl.cl\_device attribute), 21  
preferred\_vector\_width\_double (pycl.cl\_device attribute), 21  
preferred\_vector\_width\_float (pycl.cl\_device attribute), 21  
preferred\_vector\_width\_half (pycl.cl\_device attribute), 21  
preferred\_vector\_width\_int (pycl.cl\_device attribute), 21  
preferred\_vector\_width\_long (pycl.cl\_device attribute), 21  
preferred\_vector\_width\_short (pycl.cl\_device attribute), 21  
preferred\_work\_group\_size\_multiple() (pycl.cl\_kernel method), 29  
printf\_buffer\_size (pycl.cl\_device attribute), 21  
private\_mem\_size() (pycl.cl\_kernel method), 29  
profile (pycl.cl\_device attribute), 21  
profile (pycl.cl\_platform attribute), 18  
profiling\_timer\_resolution (pycl.cl\_device attribute), 21  
program (pycl.cl\_kernel attribute), 28  
properties (pycl.cl\_command\_queue attribute), 24  
properties (pycl.cl\_context attribute), 23  
pycl (module), 5

## Q

queue (pycl.cl\_event attribute), 17  
queue\_properties (pycl.cl\_device attribute), 21

## R

reference\_count (pycl.cl\_command\_queue attribute), 24  
reference\_count (pycl.cl\_context attribute), 23  
reference\_count (pycl.cl\_device attribute), 21  
reference\_count (pycl.cl\_event attribute), 17  
reference\_count (pycl.cl\_kernel attribute), 28  
reference\_count (pycl.cl\_mem attribute), 25  
reference\_count (pycl.cl\_program attribute), 27

## S

setarg() (pycl.cl\_kernel method), 28  
single\_fp\_config (pycl.cl\_device attribute), 22  
size (pycl.cl\_mem attribute), 25  
source (pycl.cl\_program attribute), 27  
status (pycl.cl\_event attribute), 17

## T

type (pycl.cl\_device attribute), 22  
type (pycl.cl\_event attribute), 17  
type (pycl.cl\_mem attribute), 25

## V

vendor (pycl.cl\_device attribute), 22  
vendor (pycl.cl\_platform attribute), 17



`vendor_id` (`pycl.cl_device` attribute), [22](#)

`version` (`pycl.cl_device` attribute), [22](#)

`version` (`pycl.cl_platform` attribute), [18](#)

## W

`wait()` (`pycl.cl_event` method), [17](#)

`work_group_size()` (`pycl.cl_kernel` method), [29](#)