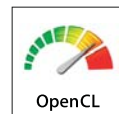




OpenCL Utility Library

API Reference and User's Guide

Version 0.3





© Copyright International Business Machines Corporation, 2011

All Rights Reserved

Printed in the United States of America March 15, 2011

IBM, IBM logo, Power6, Power7, Power Architecture, and Power Systems are trademarks of International Business Machines Corporation in the United States, or other countries, or both. Such trademarks may also be registered or common law trademarks in other countries. A complete and current list of IBM trademarks is available on the Web at <http://www.ibm.com/legal/copytrade.shtml>

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos.

Other company, product, and service names may be trademarks or service marks of others.

All information contained in this document is subject to change without notice. The products described in this document are NOT intended for use in applications such as implantation, life support, or other hazardous uses where malfunction could result in death, bodily injury, or catastrophic property damage. The information contained in this document does not affect or change IBM product specifications or warranties. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of IBM or third parties. All information contained in this document was obtained in specific environments, and is presented as an illustration. The results obtained in other operating environments may vary.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS PROVIDED ON AN "AS IS" BASIS. In no event will IBM be liable for damages arising directly or indirectly from any use of the information contained in this document.

IBM Systems and Technology Group

2070 Route 52, Bldg. 330

Hopewell Junction, NY 12533-6351

The IBM home page can be found at <http://www.ibm.com>

March 15, 2011

Table of Contents

1	OVERVIEW	4
1.1	Tested Configurations.....	4
1.2	Installation and Compilation.....	4
1.3	Related Documentation.....	5
1.4	What Is New.....	5
2	API REFERENCE AND IMPLEMENTATION DETAILS	6
2.1	Implementation Details.....	6
2.1.1	Interaction with OpenCL APIs	6
2.1.2	Error Handling.....	6
2.1.3	Thread Safety.....	6
2.2	CLU Objects	7
2.2.1	clu_t.....	7
2.3	Simplifying Functions	8
2.3.1	cluInit.....	8
2.3.2	cluCreateCmdQueue.....	9
2.3.3	cluCreateKernel.....	10
2.3.4	cluSetKernelNDRange	12
2.3.5	cluSetKernelDependency	14
2.3.6	cluSetKernelCmdQueue	15
2.3.7	cluRunKernel.....	16
2.3.8	cluEnableKernelProfiling	18
2.3.9	cluGetKernelExecTime	19
2.3.10	cluDisableKernelProfiling	21
2.3.11	cluGetAvailableLocalMem	22
2.3.12	cluCheckLocalWorkgroupSize.....	23
2.3.13	cluGetCLDeviceTypeString	24
2.3.14	cluGetCLContext.....	25
2.3.15	cluGetCLPlatformID	26
2.3.16	cluGetDeviceID.....	27
2.3.17	cluCheckDeviceExtensions	28
2.3.18	cluDestroy	29
2.4	Helper Functions	30
2.4.1	cluGetDeviceName.....	30
2.4.2	cluGetErrorString	30
2.4.3	cluPrintDeviceInfo	31
2.4.4	cluPrintPlatformInfo.....	32
2.4.5	CLU_CHECK_ERROR	33
2.4.6	CLU_EXIT_ERROR.....	34

1 Overview

The IBM's OpenCL™ Utility Library (CLU) is a set of utility functions that simplify the task of developing an OpenCL application. There are several ways the CLU APIs accomplish this goal:

- The CLU APIs group certain OpenCL APIs together to reduce the number of OpenCL calls a typical OpenCL application needs to make.
- The CLU APIs provide utility functions to help with the development process.
- The CLU APIs make certain assumptions about the application and are thus able to eliminate parameters from several OpenCL functions.

CLU is most helpful for simple to moderately complex OpenCL applications. For more complex programs, CLU can be used; however, its usefulness might be limited.

1.1 Tested Configurations

The CLU implementation provided in the OpenCL Samples version 0.3-0 package has been tested in conjunction with the OpenCL Development Kit for Linux® on Power version 0.3. As such, the tested configurations are the same as the Development Kit. See the OpenCL Development Kit for Linux® on Power, Installation and User's Guide for a list of tested configurations.

While this implementation of CLU will likely to be fully operational with other conforming OpenCL version 1.0 and 1.1 runtimes and compilers, it is untested and is not guaranteed to be without defects.

1.2 Installation and Compilation

CLU is part of the OpenCL Samples version 0.3-0 package. To install CLU, simply unzip the OpenCL-sample-0.3-0.zip package into a directory of your own choosing. CLU is shipped as source only. You can browse the CLU source code by viewing the files in the clu sub-directory.

All the OpenCL samples use the CLU APIs.

To compile an application with CLU, include clu.h in the source code and compile and link with clu.o as part of the program build process.

1.3 Related Documentation

1. *OpenCL Development Kit for Linux on Power – Installation and User’s Guide – Version 0.3* which includes information on the IBM implementation for Power Architecture®.
<http://www.alphaworks.ibm.com/tech/openccl/>.
2. *Khronos OpenCL API registry* which includes the OpenCL Core API Specification, Headers, and Extension Documentation. <http://www.khronos.org/registry/cl>.
3. *OpenCL 1.0 Reference Pages*, Khronos Group,
<http://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/>
4. *OpenCL 1.1 Reference Pages*, Khronos Group,
<http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/>
5. *OpenCL API 1.0 Quick Reference Card*, Revision 0409, Copyright © 2009 Khronos Group,
<http://www.khronos.org/files/openccl-quick-reference-card.pdf>.
6. *OpenCL API 1.1 Quick Reference Card*, Revision 0610, Copyright © 2010 Khronos Group,
<http://www.khronos.org/files/openccl-1-1-quick-reference-card.pdf>

1.4 What Is New

CLU Revision	Summary of Changes
0.3	This version of CLU has been enabled to support OpenCL 1.1. In addition, 5 new APIs have been provided -- cluGetDeviceID , cluCheckDeviceExtensions , cluGetDeviceName , cluGetErrorString , and CLU_EXIT_ERROR .

2 API Reference and Implementation Details

Currently, CLU APIs are divided into two categories:

- APIs that help simplifying OpenCL interface. See [Simplifying Functions](#).
- APIs that help users to develop OpenCL programs. The final program might not include invocations of these functions, however, during the development process, these functions might be useful. See [Helper Functions](#).

2.1 Implementation Details

2.1.1 Interaction with OpenCL APIs

The CLU interface is designed to co-exist with other OpenCL functions. Users can choose to use any of the CLU APIs to simplify development of their OpenCL applications or use the OpenCL APIs to take advantage of all the flexibilities that OpenCL offers.

2.1.2 Error Handling

CLU does not return error codes. Upon encountering an error condition, CLU will output an error message to *stderr* and exit with an `EXIT_FAILURE` error code.

2.1.3 Thread Safety

CLU is not thread safe. The user is responsible for setting up an appropriate mechanism to ensure thread safety in their applications.

2.2 CLU Objects

2.2.1 `clu_t`

A *clu_t* object is the main object for the CLU APIs. Most CLU APIs require a valid, non-zero *clu_t* object as input. Once initialized, this object contains information about the OpenCL platform, context, command queues, programs, kernels, kernel executing instances and other CLU specific information. A call to [cluInit](#) will return a valid *clu_t* object.

2.3 Simplifying Functions

2.3.1 cluInit

Syntax:

```
clu\_t cluInit (cl_platform_id platform)
```

Description:

This function initializes the OpenCL platform and creates a context for all the devices on the platform. If the *platform* parameter is NULL, the OpenCL platform that CLU uses is the first platform found in the system by the [clGetPlatformIDs](#) function.

Parameters:

platform specifies the OpenCL cl_platform_id.

Return:

A valid, non-zero [clu_t](#) object.

2.3.2 cluCreateCmdQueue

Syntax:

```
cl_command_queue cluCreateCmdQueue (clu\_t clu,
                                     cl_device_id dev_id,
                                     cl_device_type dtype,
                                     cl_command_queue_properties properties)
```

Description:

This function creates a `cl_command_queue` object with the given *properties* for either the device specified by *dev_id* or if *dev_id* is 0 (zero), the first device in the platform that matches the given device type *dtype*.

Calling this interface multiple times with the same `cl_device_id` can create multiple `cl_command_queue` objects for the same device.

Parameters:

<i>clu</i>	specifies a valid clu_t object.
<i>dev_id</i>	specifies the <code>cl_device_id</code> of the device that is being initialized. If <i>dev_id</i> is zero, it is ignored.
<i>dtype</i>	type of the device to initialize. If <i>dev_id</i> is non-zero, this parameter is ignored.
<i>properties</i>	specifies a list of properties for the command queue. This is a bit-field and is described in table 5.1 of the OpenCL version 1.0 rev 48 specification.

Return:

A valid, non-zero `cl_command_queue` object.

2.3.3 cluCreateKernel

Syntax:

```
cl_kernel cluCreateKernel (clu_t clu,
                          cl_command_queue cmd_queue,
                          const char* filename,
                          const char* kernel_name,
                          const char* build_options,
                          clu_create_program_flag_t flag)
```

Description:

This function creates a `cl_program` from either a source file or a binary file and builds the program for the command queue's `cl_device_id` with the given *build_options*. The function then creates a kernel with the given kernel name *kernel_name* and returns the newly created `cl_kernel` object.

Parameters:

<i>clu</i>	specifies a valid <code>clu_t</code> object.
<i>cmd_queue</i>	specifies a non-zero, valid <code>cl_command_queue</code> object.
<i>filename</i>	name of either the source file that contains the kernel code or the binary file that contains a pre-compiled kernel.
<i>kernel_name</i>	name of the kernel to create.
<i>build_options</i>	options for building the program on the specified device.
<i>flag</i>	specifies a list of options on how the program is built. This is a bit-field and is described as followed:

CLU_SOURCE: The OpenCL program is created from source. The name of the source file is given by *filename*. On the IBM's implementation, when a binary program is created from source, the binary is saved in the current working directory with a file extension of `ocl_bin`. The current working directory is the cache. See implementation details below for more information.

CLU_BINARY : The OpenCL program is created from binary. The name of the binary file is given by *filename*.

CLU_NO_CACHE: The OpenCL program is rebuilt regardless whether there's a cache version of the built program. The newly built program will

not be saved to cache.

Return:

A valid, non-zero `cl_kernel` object.

Implementation details:

On the IBM OpenCL runtime, the CLU implementation will attempt to cache already built programs. When creating an OpenCL program from source, CLU will attempt to detect whether there is an appropriate version of the program that's already built and use that instead of rebuilding. To force CLU to rebuild program, regardless of the availability of a cached version, the `CLU_NO_CACHE` `clu_create_program_flag_t` can be used to force this behavior.

Currently, a simple cache mechanism is used. The file system time stamp of the binary and source files are used to detect whether the source file is newer.

A built program binary is saved with the following name:

`source_file.device_name.build_options.ocl_bin`

All non-alphanumeric characters in the queried device name or the build options will be converted to underscores.

The current implementation cannot detect changes in a kernel source file's `#include` files. An environment variable is provided (`IBM_OPENCL_CLU_NO_CACHE=Y`) to turn off caching regardless of input from the user to avoid the cache system not working appropriately with user provided kernel `#include` files.

2.3.4 cluSetKernelNDRange

Syntax:

```
void cluSetKernelNDRange (clu_t clu,
                          cl_kernel kernel,
                          cl_uint work_dim,
                          const size_t* global_work_offset,
                          const size_t* global_work_size,
                          const size_t* local_work_size)
```

Description:

This function declares the given *kernel* to be an NDRange kernel and initializes all the NDRange parameters for the given *kernel*.

Calling this function multiple times will reset the NDRange parameters for the given *kernel*.

Parameters:

<i>clu</i>	specifies a valid clu_t object.
<i>kernel</i>	specifies a valid <code>cl_kernel</code> object.
<i>work_dim</i>	number of dimensions used to specify the global work-items and work-items in the work-group. <i>work_dim</i> must be greater than 0 and less than or equal to <code>CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS</code> .
<i>global_work_offset</i>	points an array of <i>work_dim</i> unsigned values that describe the offset used to calculate the global ID of a work-item. If <i>global_work_offset</i> is NULL, the global IDs start at offset (0, 0, ... 0). <i>global_work_offset</i> must be a NULL for OpenCL 1.0 devices.
<i>global_work_size</i>	points to an array of <i>work_dim</i> <code>size_t</code> values that describe the number of global work-items in <i>work_dim</i> dimensions that will execute the kernel function. The total number of global work-items is computed as <code>global_work_size[0] * ... * global_work_size[work_dim - 1]</code> .
<i>local_work_size</i>	points to an array of <i>work_dim</i> <code>size_t</code> values that describe the number of work-items that make up a work-group that will execute the kernel. A full description of this parameter can be found in section 5.6 of the OpenCL 1.0 rev 48 specification.

Return:

void

2.3.5 cluSetKernelDependency

Syntax:

```
void cluSetKernelDependency (clu_t clu,  
                             cl_kernel kernel,  
                             clu_uint num_events_in_wait_list,  
                             const cl_event* event_wait_list)
```

Description:

This function specifies the events that need to complete before this particular kernel can be executed.

Parameters:

<i>clu</i>	specifies a valid clu_t object.
<i>kernel</i>	specifies a valid <code>cl_kernel</code> object.
<i>num_events_in_wait_list</i>	the number of events that need to complete before this kernel can be executed. If this is 0, then <i>event_wait_list</i> must be NULL.
<i>event_wait_list</i>	an array contains the events that need to complete before this kernel can be executed. If this is NULL, then, <i>num_events_in_wait_list</i> must be 0.

Return:

void

2.3.6 cluSetKernelCmdQueue

Syntax:

```
void cluSetKernelCmdQueue (clu\_t clu,  
                           cl_kernel kernel,  
                           cl_command_queue cmd_queue)
```

Description:

This function binds the given *kernel* with a `cl_command_queue` *cmd_queue* for execution. This function is needed if the kernel was created outside of CLU. This function can also be used to change the command queue that a kernel executes on.

If the kernel's program has not been built for the command queue's device, CLU will print out an error message and exit.

Since CLU did not create the kernel, it will not release it.

Parameters:

<i>clu</i>	specifies a valid clu_t object.
<i>kernel</i>	specifies a valid <code>cl_kernel</code> object.
<i>cmd_queue</i>	specifies a <code>cl_command_queue</code> object.

Return:

void

2.3.7 cluRunKernel

Syntax:

```
void cluRunKernel (clu\_t clu,  
                  cl\_kernel kernel,  
                  cl\_event* event,  
                  cl\_uint num_args,  
                  ...)
```

Description:

This function sets up the arguments for the given [cl_kernel](#) *kernel* then en-queues the *kernel* onto the command queue associated with the kernel's device for execution.

The input *num_args* has to agree with the number of arguments in the kernel. One can find the number of arguments in a kernel by calling the OpenCL function [clGetKernelInfo](#) (*kernel*, CL_KERNEL_NUM_ARGS, ...)

The input [cl_kernel](#) *kernel* can either be an NDRange kernel or a Task kernel. If it's an NDRange kernel, a prior call to [cluSetKernelNDRange](#) is needed. If it's a Task kernel, no other calls are needed.

If the kernel has not been created by [cluCreateKernel](#), then [cluSetKernelCmdQueue](#) must be called prior to calling [cluRunKernel](#).

If the event parameter is not NULL, a [cl_event](#) object will be returned to the user. CLU will not release this event object.

Parameters:

<i>clu</i>	specifies a valid clu_t object.
<i>kernel</i>	specifies the cl_kernel object that user wants to execute.
<i>event</i>	returns an event object that identifies this particular kernel execution instance. If event is NULL, no event will be created for this kernel execution instance and therefore it will not be possible for the application to query or queue a wait for this kernel execution instance.
<i>num_args</i>	the number of arguments for the kernel that we are setting. This is the total number of arguments the kernel has.

The next two parameters are the variable arguments and can be repeated as necessary.

arg_size the size of the argument's value in bytes. The type for this parameter is `size_t`.

arg_val the address of the argument. The type for this parameter is `void*`.

Return:

This function returns the `cl_event` object that identifies this particular kernel execution instance via the *event* input pointer.

Motivation:

The OpenCL kernel execution functions, [clEnqueueNDRangeKernel](#) and [clEnqueueTask](#) have been wrapped by CLU to improve the flow and readability of an application.

The new [cluRunKernel](#) function supports both NDRange and Task kernels. Users can provide NDRange parameters for a specific `cl_kernel` object via the [cluSetKernelNDRange](#) call. When a kernel is created via the [cluCreateKernel](#) function, it is designated as a Task kernel. The kernel is changed to an NDRange kernel when [cluSetKernelNDRange](#) is invoked.

The [cluSetKernelDependency](#) function separates event dependency settings from the execution of the kernel. It has been observed that many OpenCL programs do not use event dependencies at all and the parameters for events in the [clEnqueueNDRangeKernel](#) function are often set to NULL. Separating event dependency setting for a kernel can simplify the kernel en-queuing process.

The [cluRunKernel](#) function offers a more C-friendly style, allowing users to execute a kernel using the kernel object and a list of parameters.

2.3.8 cluEnableKernelProfiling

Syntax:

```
void cluEnableKernelProfiling (clu\_t clu,  
                               cl\_kernel kernel)
```

Description:

This function enables profiling for the specified kernel. This function must be called prior to calling [cluRunKernel](#). It is not necessary to specify a returned event in the [cluRunKernel](#) invocation; CLU will provide a valid event pointer parameter to [cluRunKernel](#) if one is not provided. The event will be retained and released by CLU.

There must be a command queue associated with the kernel either through the [cluCreateKernel](#) API or through the [cluSetKernelCmdQueue](#) API. The command queue must have been created with [CL_QUEUE_PROFILING_ENABLE](#).

Parameters:

<i>clu</i>	specifies a valid, non-zero clu_t object.
<i>kernel</i>	specifies a valid cl_kernel object. This kernel must have a cl_command_queue associated with it.

Return:

void

2.3.9 cluGetKernelExecTime

Syntax:

```
float cluGetKernelExecTime (clu_t clu,  
                             cl_kernel kernel,  
                             clu_profiling_flag_t flag)
```

Description:

This function returns the execution time for the input cl_kernel *kernel*.

If the input clu_profiling_flag_t *flag* is CLU_PROFILING_ELAPSED_TIME then this function returns the total time elapsed between the time when the first executing instance of the kernel starts to run and the time when the last executing instance of the kernel is finished running.

If the input clu_profiling_flag_t *flag* is CLU_PROFILING_ACCUM_TIME then this function returns the accumulated executing time for all executing instances of each kernel where the executing time for each instance is the difference between the time when the executing instance starts and the time when the executing instance ends. Note that executing instances may be happen in parallel so the accumulated time might not be similar to the total elapsed time.

All outstanding executing instances of the kernel must be complete for this function to work properly.

There must be a command queue associated with the kernel either through the [cluCreateKernel](#) API or through the [cluSetKernelCmdQueue](#) API.

Once this function returns, all events retained by CLU for this kernel will be released. The next invocation of this function for the same kernel will return the executing time for the new executing instances of this kernel.

The CLU function [cluEnableKernelProfiling](#) must be previously invoked.

Parameters:

<i>clu</i>	specifies a valid clu_t object.
<i>kernel</i>	specifies a valid cl_kernel object. This kernel must have a cl_command_queue associated with it.
<i>flag</i>	can be either of the following values:

- CLU_PROFILING_ELAPSED_TIME

- CLU_PROFILING_ACCUM_TIME

Return:

This function returns the kernel executing time as described above.

2.3.10 cluDisableKernelProfiling

Syntax:

```
void cluDisableKernelProfiling (clu\_t clu,  
                                cl_kernel kernel)
```

Description:

This function disables profiling for the specified `cl_kernel`. Once this function is invoked, further invocations to [cluGetKernelExecTime](#) will return a value of 0.0f.

Parameters:

<i>clu</i>	specifies a valid clu_t object
<i>kernel</i>	specifies a valid <code>cl_kernel</code> object.

Return:

void

2.3.11 cluGetAvailableLocalMem

Syntax:

```
clu_ulong cluGetAvailableLocalMem (cl_device_id device_id,  
                                   cl_kernel kernel)
```

Description:

This function returns the amount (in bytes) of available local memory on the given device after the given kernel is loaded. This number is calculated by subtracting the kernel local memory size ([CL_KERNEL_LOCAL_MEM_SIZE](#)) from the total device local memory ([CL_DEVICE_LOCAL_MEM_SIZE](#)).

Parameters:

device_id specifies the cl_device_id

kernel specifies the cl_kernel

Return:

This function returns the amount (in bytes) of available local memory.

2.3.12 cluCheckLocalWorkgroupSize

Syntax:

```
cl_bool cluCheckLocalWorkgroupSize (cl_device_id device_id,
                                     cl_kernel kernel,
                                     cl_uint work_dim,
                                     const size_t* local_work_sizes)
```

Description:

This function checks whether the user's required number of work item dimensions *work_dim* and the values in the array *local_work_sizes* can be supported by the given device and kernel.

Parameters:

<i>device_id</i>	specifies the <code>cl_device_id</code> .
<i>kernel</i>	specifies the <code>cl_kernel</code> .
<i>work_dim</i>	specifies the number of dimensions in the work group. This is also the number of elements in the <i>local_work_sizes</i> array.
<i>local_work_sizes</i>	points to an array of <code>work_dim</code> <code>size_t</code> values that describe the number of work-items that make-up a work-group.

Return:

This function returns `CL_TRUE` if the work item dimensions *work_dim* and the *local_work_sizes* are supported by the device and the kernel. This function returns `CL_FALSE` otherwise.

2.3.13 cluGetCLDeviceTypeString

Syntax:

```
const char * cluGetCLDeviceTypeString (cl_device_type device_type)
```

Description:

This function returns a character array representing the type of the device. If the device type is:

- CL_DEVICE_TYPE_DEFAULT: the returned string is “DEFAULT”
- CL_DEVICE_TYPE_ACCELERATOR: the returned string is “ACCELERATOR”
- CL_DEVICE_TYPE_CPU: the returned string is “CPU”
- CL_DEVICE_TYPE_GPU: the returned string is “GPU”
- otherwise, the returned string is “UNKNOWN”

Parameters:

device_type specifies a valid cl_device_type.

Return:

This function returns a string representing the type of the device.

2.3.14 cluGetCLContext

Syntax:

cl_context cluGetCLContext ([clu_t](#) *clu*)

Description:

This function returns the cl_context for the input [clu_t](#) object.

Parameters:

clu specifies the [clu_t](#) object.

Return:

This function returns the cl_context for this [clu_t](#) object.

2.3.15 cluGetCLPlatformID

Syntax:

cl_platform_id cluGetCLPlatformID ([clu_t](#) *clu*)

Description:

This function returns the cl_platform_id for the input [clu_t](#) object.

Parameters:

clu specifies the [clu_t](#) object.

Return:

The cl_platform_id.

2.3.16 cluGetDeviceID

Syntax:

```
cl_device_id cluGetDeviceID (clu_t clu, cl_device_type device_type, const char*
                             device_vendor, const char* device_name)
```

Description:

This function returns the *cl_device_id* that matches the given *device_type* with a CL_DEVICE_VENDOR and CL_DEVICE_NAME that contains the substring *device_vendor* and *device_name*, respectively. When searching the CL_DEVICE_NAME for the *device_name*, the case for both strings is ignored.

If multiple devices match the *device_type*, *device_vendor*, and the *device_name* search criteria, then the function returns the first matching device.

If *device_vendor* or *device_name* is NULL this function ignores these inputs and returns the first device that matches the given *device_type*.

Parameters:

<i>clu</i>	specifies the <i>clu_t</i> object.
<i>device_type</i>	specifies the <i>cl_device_type</i>
<i>device_vendor</i>	specifies the substring the device's CL_DEVICE_VENDOR string must contain. If NULL, then the CL_DEVICE_VENDOR string is not consider during the search for a device.
<i>device_name</i>	specifies the substring the device's CL_DEVICE_NAME string must contain. If NULL, then the CL_DEVICE_NAME string is not considered when searching for a device.

Return:

A valid *cl_device_id* if it can find one that matches. Otherwise NULL is returned.

Implementation:

The returned *cl_device_id* is added to a hash table maintained by CLU.

2.3.17 cluCheckDeviceExtensions

Syntax:

```
cl_bool cluCheckDeviceExtensions (cl_device_id dev_id, const char* ext_names)
```

Description:

This function checks the device specified by *dev_id* for support of the extensions specified by *ext_names*. If all the extensions are supported, then CL_TRUE is returned. Otherwise, CL_FALSE is returned.

Parameters:

dev_id specifies the device to be checked for extension support.

ext_names a space separated list of OpenCL device extension names.

Return:

CL_TRUE is returned if the device supports all extensions specified in *ext_names*. Otherwise, CL_FALSE is returned.

2.3.18 cluDestroy

Syntax:

```
void cluDestroy (clu\_t clu)
```

Description:

This function frees all memories that were allocated by CLU. It also releases the OpenCL resources that were created or retained by CLU. This includes the following:

- all the `cl_command_queue` objects that were created using [cluCreateCmdQueue](#),
- all the `cl_context` objects created via [cluInit](#),
- all the `cl_kernel` objects created via [cluCreateKernel](#),
- all the `cl_program` objects created via [cluCreateKernel](#), and
- all the `cl_event` objects retained in [cluRunKernel](#) when [cluEnableKernelProfiling](#) is used.

Once this function returns, any reference to `clu` will result in undetermined behavior. References to all the OpenCL objects will return OpenCL errors as appropriate.

Parameters:

clu specifies the [clu_t](#) object

Return:

void

2.4 Helper Functions

2.4.1 cluGetDeviceName

Syntax:

```
const char * cluGetDeviceName (clu_t clu, cl_device_id device_id)
```

Description:

This function returns a pointer to the CL_DEVICE_NAME string corresponding to the device specified by *device_id*. The memory allocated for the device name is managed and maintained by CLU and must/need not be freed by the caller.

Parameters:

clu specifies the clu_t object

device_id specifies the cl_device_id

Return:

A pointer to the device's name string.

Implementation:

The implementation of this function searches through the device hash table of the clu_t object for the clu_device_t object with the given *device_id*. If it cannot find the clu_device_t object, it allocates one, along with a copy of its device name string.

2.4.2 cluGetErrorString

Syntax:

```
const char * cluGet ErrorString (cl_int errcode)
```

Description:

This function returns a pointer to a descriptive string corresponding to the specified OpenCL error code, *errcode*. If the error code is not known, then a pointer to the string "Unknown" is returned.

Parameters:

errorcode specifies the OpenCL error code for which an error string is to be returned.

Return:

A descriptive string pointer corresponding to the specified error code.

Implementation:

The error strings are statically defined and thus should not be freed by the user/application.

2.4.3 cluPrintDeviceInfo**Syntax:**

```
void cluPrintDeviceInfo (cl_device_id dev)
```

Description:

This function prints to *stdout* all information that can be queried about the device. This includes all the device information in table 4.3 of the OpenCL Specification Version 1.0, revision 48.

Parameters:

dev specifies a valid, non-zero input `cl_device_id` object

Return:

void

2.4.4 cluPrintPlatformInfo

Syntax:

```
void cluPrintPlatformInfo (clu_t clu)
```

Description:

This function prints to *stdout* all information that can be queried about the platform associated with the given `clu_t` object. This includes all platform information in table 4.1 of the OpenCL Specification Version 1.0, revision 48.

Parameters:

clu specifies a valid, non-zero input `clu_t` object

Return:

void

2.4.5 CLU_CHECK_ERROR

Syntax:

CLU_CHECK_ERROR(prefix, errcode)

Description:

This convenience macro prints to *stderr* an error message if the given *errcode* is a value other than CL_SUCCESS and exits the program with EXIT_FAILURE exit code. An error string represents the CL error listed cl.h is part of the print message unless the error number is not known. In this case, the error string is “unknown error”. In addition, a back trace of maximum of 10 calls deep is printed to *stderr*.

For example, the following piece of code in the file *myopencl_code.c*

```
void myfunc()
{
    ...
    cl_int err = -30;
    CLU_CHECK_ERROR("clGetDeviceInfo CL_DEVICE_NAME", err);
    ...
}
```

exits the program with an EXIT_FAILURE code and prints to *stderr* the following error message:

```
ERROR in function myfunc, in file myopencl_code.c, line 100: clGetDeviceInfo
CL_DEVICE_NAME = -30 (CL_INVALID_VALUE)
```

2.4.6 CLU_EXIT_ERROR

Syntax:

CLU_EXIT_ERROR(fmt, arg ...)

Description:

This convenient macro prints an error message in a format that's consistent with all CLU error messages and exits the program with an EXIT_FAILURE error code.

For example, the following code snippet:

```
char * allocateBuffer(size_t buf_size)
{
    char *buffer;
    if ((buffer = (char *)malloc(buf_size)) == NULL) {
        CLU_EXIT_ERROR("Unsuccessful allocation of %zd bytes.\n", buf_size);
    }
    return buffer;
}
```

results in the following output and exits if the malloc failed.

```
CLU ERROR in function allocateBuffer, file example.c, line 8 - Unsuccessful
allocation of 12467998 bytes.
```

END OF DOCUMENT