Brown Deer Technology

COPRTHR® API Reference

1 Overview

1.1 Purpose

The CO-PRocessing THReads® SDK provides the STDCL API for programming compute offload accelerators. STDCL contains a rich set of features designed around the application programmer in order to provide a high-level API that is intuitive, powerful and easy to use. The COPRTHR SDK also provides OpenCL implementations for various processors that can be leveraged using STDCL. For many scenarios this will be sufficient for the programmer.

In some cases, a lower level API may be advantageous to access a co-processor device more directly and expose features that are not supported in higher level APIs such as STDCL and OpenCL. This proved to be the case for the Epiphany RISC array processor, which had features inaccessible from the higher level APIs designed for GPUs. Additionally, some projects building middleware for such a processor may find it desireable to use a more direct and light-weight API with less overhead. This was the motivation for formalizing into an API the "machinery" used to build the COPRTHR OpenCL implementations, and this is the origin of the COPRTHR API. The basic functions may appear similar in principle to other APIs for co-processors since ulimately all compute offload APIs have the same basic requirements.

As a by-product of the effort, and a demonstration of the usefulness of such an API, a direct extension of Pthreads for co-processors was built as a part of this COPRTHR API utilizing the basic machinery to create functionality unsupported by the higher-level APIs. This likely serves as only one example of what might be constructed from such a low-level API for accessing a processor like Epiphany.

At present this low-level COPRTHR API can be used to target x86 CPUs, ARM CPUs, and the Epiphany RISC array processors. Support will be extended to Intel MIC in the near future.

1.2 Components

There are four elements of the COPRTHR API aiming to address specific requirements for complex development projects:

- COPRTHR/cc: a cross-compiler library for co-processors
- COPRTHR direct run-time API
- COPRTHR/thread: pthreads extension for co-processors
- COPRTHR/dev: low-level device API

---

2 COPRTHR/cc: a cross-compiler library for co-processors

The COPRTHR/cc API is a generalized cross-compiler library for co-processor devices provided as part of the COPRTHR API. The functionality is completely separated from the associated run-time for the device, which may or may not be available. Therefore cross-compilation support may be used on platforms where the target device is not even present.

The interface for cross-compilation is extremely general and simple, consisting of one essential function and two additional functions added for convenience. Program source code is provided as a string with the result being a program struct that may be used by the COPTHR API run-time. The interface is deliberately simple and generic, designed without complicated features, with the intent being that it may be used as part of more complex higher-level tools. An example of this can be found in the COPRTHR clcc compiler tool which uses this interface for cross-compilation. At the same time, programmers may find the interface to be a convenient and simple method for prgramming co-processor devices.

Note that no special programming language is introduced for supporting co-processor devices since none is needed. C *without extensions* is perfectly suited for programming co-processors and attempts to introduce special device languages that break C compliance have been misguided. A design principle of COPRTHR is to avoid such mistakes.

SYNOPSIS

```
#include <coprthr.h>
#include <coprthr_cc.h>

typedef struct coprthr_program* coprthr_program_t;

coprthr_program_t
    coprthr_cc( const char* src, size_t len, const char* opt, char** log );

size_t
    coprthr_cc_write_bin(const char* path, coprthr_program_t prg, int flags);

coprthr_program_t
    coprthr_cc_read_bin( const char* path, int flags );

coprthr_sym_t
    coprthr_getsym( coprthr_program_t program, const char* symbol );

Link with -lcoprthrcc
```

DESCRIPTION

The function coprthr_cc() will cross-compile the provided program and return a pointer to a struct representing the compiled program object that may be used by the COPRTHR API run-time.

The source code for the program *src* of *len* bytes long will be compiled for the target device.
The arguement *opt* must be used to select the cross-compile target. Any additional compiler options may be provided as a string of the form that would be provided to a compiler at the command line. The cross-compile target is selected using the command line style options, "-mtarget=".

For convenience, several predefined targets are available as macros:

```
#define COPRTHR_CC_TARGET_X86_64 "-mtarget=x86_64"
#define COPRTHR_CC_TARGET_I386 "-mtarget=i386"
#define COPRTHR_CC_TARGET_ARM32 "-mtarget=arm32"
```

```
#define COPRTHR_CC_TARGET_E32_EMEK "-mtarget=e32_emek"
#define COPRTHR_CC_TARGET_E32 "-mtarget=e32"
#define COPRTHR_CC_TARGET_ARM32_ANDROID "-mtarget=arm32_android"
#define COPRTHR_CC_TARGET_MIC "-mtarget=mic"
```

Not all of these targets may be supported on a given platform. Passing the flag "–targets" will cause a list of supported targets to be provided as output in the compiler log.

The argument *log* provides a pointer to a log buffer where the output log from the compiler can be stored. When this argument points to a null pointer, a buffer of suitable size will be allocated, and should be subsequently released using the free() call.

A final call mentioned here is not actually part of the COPRTHR/cc library, but is instead part of the COPRTHR API run-time. However it provides the bridge between the two and is useful to mention. Once a call to coprthr_cc() has produced a compiled program, it will be necessary to obtain a handle to the one or more entry points within the program that are defined by the thread functions of the program. The coprthr_getsym() function is used to get a handle for the thread function identified by name.

---

3 COPRTHR direct run-time API

The COPRTHR direct run-time API provides a convenient set of calls for accessing a co-processor device including malloc()-style functions for device memory allocation, a stream model for non-blocking asynchronous device operations, a threading model based on a direct extension of the pthread API for co-processors, and signaling support between threads of execution on the host and co-processor device.

3.1 Opening a co-processor device

SYNOPSIS

```
#include <coprthr.h>

int coprthr_dopen( const char* path, int flags);

int coprthr_dclose( int dd );

Link with -lcoprthr
```

DESCRIPTION

Access to a co-processor devices requires opening the device using the *coprthr_dopen()* call which returns a *device descriptor* that is used in all subsequent operations. The *path* may be a literal path to a device special file or one of the pre-defined macros for known supported devices. Currently supported devices include:

```
COPRTHR_DEVICE_X86_64
COPRTHR_DEVICE_I386
COPRTHR_DEVICE_ARM32
```

```
COPRTHR_DEVICE_E32_EMEK
COPRTHR_DEVICE_E32
COPRTHR_DEVICE_MIC
```

The *flags* argument controls the behavior of the opened device. The `COPRTHR_O_NONBLOCK` flag caues the call to return with an error if the device is temporarily unavailable.
The flag `COPRTHR_O_EXCLUSIVE` causes the call to return with an error if exclusive access to the device could not be established.

The flags `COPRTHR_O_STREAM` and `COPRTHR_O_THREAD` specify the mode of operation in which the co-processor device should be opened.

Finally, the flag `COPRTHR_O_DEFAULT` may be used to select the default flags configured by the installation.

Access to the device is closed using the coprthr_dclose() function where the arguement *dd* is the device descriptor that was returned from the coprthr_dopen() call.

3.2 Device Memory Allocation

SYNOPSIS

```
#include <coprthr.h>

coprthr_mem_t
    coprthr_dmalloc( int dd, size_t size, int flags );

coprthr_mem_t
    coprthr_drealloc( int dd, coprthr_mem_t mem, size_t size, int flags );

void coprthr_dfree( int dd, coprthr_mem_t mem );

Link with -lcoprthr
```

DESCRIPTION

Device memory is allocated using the familiar semantics of the conventional malloc() calls extended to a co-processor device. The coprthr_dmalloc() call returns a pointer to struct containing the necessary information abou the device allocation. In order to obtain the actual address of the device memory, the coprth_devmemptr() call, discussed below, may be used. Thge returned address is suitable, e.g., to use as an argument to a thread function executing on the co-processor. The size of an allocation may be altered using coprthr_drealloc().
Allocated device memory may be released using coprthr_dfree().

3.3 Distributed Memory Management

SYNOPSIS

```
#include <coprthr.h>

coprthr_event_t
```

```
    coprthr_dread(int dd, coprthr_mem_t mem, size_t offset, void* ptr,
        size_t len, int flags);

coprthr_event_t
    coprthr_dwrite(int dd, coprthr_mem_t mem, size_t offset, void* ptr,
        size_t len, int flags)

coprthr_event_t
    coprthr_dcopy(int dd, coprthr_mem_t mem_src, size_t offset_src,
        coprthr_mem_t mem_dst, size_t offset_dst, size_t len, int flags)

Link with -lcoprthr
```

DESCRIPTION

Memory management of device memory is supported with three functions, coprthr_dread(), coprthr_dwrite(), and coprthr_dcopy(). The coprthr_dread() call is used for reading device memory, copying its contents to a buffer on the host. Conversely, coprthr_dwrite() is used for writing to device memory the contents of a buffer on the host. The coprthr_dcopy() call is used to initiate from the host the copying of memory from one device memory allocation to another. THe offset argument is the offset in bytes into the device memory allocation. The use of non-zero offsets may or may not be supported by a given device.

Using the steam model for the opened device will cause these operations to be queued in order and executed asynchronously. Synchronization with the host is discussed below.

3.4 Executing device kernels

SYNOPSIS

```
#include <coprthr.h>

coprthr_event_t
    coprthr_dexec( int dd, coprthr_kernel_t krn, unsigned int nargs,
        void** args, unsigned int nthr, void* reserved, int flags );

coprthr_event_t
    coprthr_dnexec( int dd, unsigned int nkrn, coprthr_kernel_t v_krn[],
        unsigned int v_nargs[], void** v_args[], unsigned int v_nthr[],
        void* v_reserved[], int flags );

Link with -lcoprthr
```

DESCRIPTION

The coprthr_dexec() call executes a kernel on the device. The arguments `nargs` and `args` represent the number of arguments and an array of pointers to the arguments for the kernel. The argument `nthr` specifies the number of threads over which the kernel should be executed.

The coprthr_dnexec() call is a variant that allows multiple kernels to be executed simultaneously and treated as a single operation. The argument `nkrn` is the number of kernels to be executd. The remainder of the arguments are vectorized versions of those in the coprthr_dexec() call over the multiple kernels.

Using the steam model for the opened device will cause these operations to be queued in order and executed asynchronously. Synchronization with the host is discussed below.

3.5 Event Synchronization

SYNOPSIS

```
#include <coprthr.h>

int coprthr_dwaitev( int dd, coprthr_event_t ev );

int coprthr_dwait( int dd );

Link with -lcoprthr
```

DESCRIPTION

In general, the distributed memory management and kernel execution operations are executed asynchronously with the host and other devices. Therefore a synchronization mechanism is needed.

The coprthr_dwaitev() call will cause the host to block until the operation associated with the event `ev` has completed.

The coprthr_dwait() call will cause the host to block until all operations scheduled on the device have completed.

3.6 Cross-compilation

SYNOPSIS

```
#include <coprthr.h>
#include <coprthr_cc.h>

coprthr_program_t
    coprthr_dcompile( int dd, char* src, size_t len, char* opt, char** log );

coprthr_sym_t
    coprthr_getsym( coprthr_program_t prg, const char* symbol );

Link with -lcoprthr -lcoprthrcc
```

DESCRIPTION

The coprthr_dcompile() call is a convenience wrapper for the cross-compilation interface call coprthr_cc() and will compile code targeting the device associated with the device descriptor `dd`.

The coprthr_getsym() call is used to get the named symbol in the program, e.g., to get a handle to a given device kernel.

---

4 COPRTHR/thread

The COPRTHR direct API depends on a lower level API, COPRTHR/dev, described in a following section. Here lower level means primitive, and this API will be of little interest to the casual application programmer. The utility of this primitive API can be seen in what it enables. The above described COPRTHR direct API is augmented with a thread model designed as a logical and minimal extension Pthreads to support co-processors. This was the original concept for thr COPRTHR project, and the refactoring discussed at the begining of this document enabled the implementation of basic functionality for "pthreads for co-processors" in a very short amount of time. This thread support is included in the baic COPRTHR API.

4.1 Pthread-style threads

SYNOPSIS

```
#include <coprthr.h>

int coprthr_attr_init( coprthr_td_attr_t* attr );

int coprthr_attr_destroy( coprthr_td_attr_t* attr );

int coprthr_attr_setdetachstate( coprthr_td_attr_t* attr, int state );

int coprthr_attr_setdevice( coprthr_td_attr_t* attr, int dd ); /*NEW*/

int coprthr_attr_setinit( coprthr_td_attr_t* attr, int action ); /*NEW*/

int coprthr_create( coprthr_td_t* td, coprthr_td_attr_t* attr,
    coprthr_sym_t thr, void* arg );

int coprthr_ncreate( unsigned int nthr, coprthr_td_t* td,
    coprthr_td_attr_t* attr, coprthr_sym_t thrfunc, void* arg );

int coprthr_join( coprthr_td_t td, void** val );

Link with -lcoprthr
```

DESCRIPTION

By design, the API for the extension of pthreads to co-processors mirrors that of conventional POSIX threads calls in every possible way. Therefore most calls require description - just read the man page for the equivalent pthreads call.

The mechanism for maintaining near transparency with pthreads is to attach the device specification to the attribute used in the creation of the conventional pthread objects. The coprthr_attr_setdevice() call is introduced to attach the device descriptor to the attribute that will be used for thread creation.

An additional call is added, coprthr_attr_setinit(), to allow control over what happens when a thread is created. It as decided long ago that a pthread should not require an explicit "execute" call, and that the thread should be executed immediately and implicityl upon creation. In the world of co-processors, this behavior may not be ideal, so the flag `COPRTHR_A_CREATE_SUSPEND` requests that the thread be suspended upon creation. A scheduling call is then used to "execute" the thread at a later time. Conventional behavior (execute upon creation) can be requested with the flag `COPRTHR_A_CREATE_EXECUTE`.

4.2 Pthread-style mutexes

SYNOPSIS

```
#include <coprthr.h>

int coprthr_mutex_attr_init( coprthr_mutex_attr_t* mtxattr);

int coprthr_mutex_attr_destroy( coprthr_mutex_attr_t* mtxattr);

int coprthr_mutex_attr_setdevice( coprthr_mutex_attr_t* mtxattr, int dd ); /*NEW*/

int coprthr_mutex_init( coprthr_mutex_t* mtx, coprthr_mutex_attr_t* mtxattr);

int coprthr_mutex_destroy( coprthr_mutex_t* mtx );

int coprthr_mutex_lock( coprthr_mutex_t* mtx);

int coprthr_mutex_unlock( coprthr_mutex_t* mtx);

Link with -lcoprthr
```

DESCRIPTION

Pthread style mutexes are supported with the only additional call being coprthr_mutex_attr_setdevice() which sets the device descriptor to which the mutex is associated.

The coprthr_mutex_lock() and coprthr_mutex_unlock() calls may be used symmetrically on the host and device.

4.3 Pthread-style ccondition variables

SYNOPSIS

```
#include <coprthr.h>

int coprthr_cond_attr_init( coprthr_cond_attr_t* condattr);

int coprthr_cond_attr_destroy( coprthr_cond_attr_t* condattr);

int coprthr_cond_attr_setdevice( coprthr_cond_attr_t* condattr, int dd);

int coprthr_cond_init( coprthr_cond_t* cond, coprthr_cond_attr_t* condattr);

int coprthr_cond_destroy( coprthr_cond_t* cond);

int coprthr_wait( coprthr_cond_t* cond, coprthr_mutex_t* mtx);

int coprthr_signal( coprthr_cond_t* cond);

Link with -lcoprthr
```

DESCRIPTION

Condition variables are supported with the only additional call being coprthr_cond_attr_setdevice() which sets the device descriptor to which the condition variable is associated.

---

5 COPRTHR/dev (low-level)

5.1 Overview

The COPRTHR/dev API is the lowest-level API for accessing a co-processor device, and is therefore expectd to be quite primitive. All calls are blocking with operations immediately and without any dependencies. In order to ensure the safe execution of these operations, the programmer should first acquire a lock on the device. There is no requirement implied by the API for all devices to suport all functionality. The programmer may test suport for key functionality using a set of coprthr_testsup_*() calls.

5.2 Types

For convenience the following typedefs are provided for common structs used by the COPRTHR API. Each defined type is simply a pointer to a corresponding C struct.

```
typedef struct coprthr_program* coprthr_program_t;
typedef struct coprthr_kernel* coprthr_kernel_t;
typedef struct coprthr_device* coprthr_dev_t;
typedef struct coprthr_mem* coprthr_mem_t;
typedef struct coprthr_sym* coprthr_sym_t;
typedef struct coprthr_event* coprthr_event_t;
```

5.3 General device-dependent operations

General device-dependent opertions may be performed using a call very similar to the UNIX ioctl() call. These operations are entirely dependent upon the specific device implementation.

```
int coprthr_devctl( coprthr_dev_t dev, int request, ... )
```

5.4 Testing device suport

A single mandatory operation required of all device implementations is providing information about the functionality suported by the device. This may be obtained using the request `COPRTHR_DEVCTL_TESTSUP` returning a 32-bit word with the appropriate bits set to represent the suported functionality.

```
int devsup = coprthr_devctl(dev,COPRTHR_DEVCTL_TESTSUP);
```

The following flags may be used to check for specific suport:

```
COPRTHR_DEVSUP_F_RUNTIME
COPRTHR_DEVSUP_F_COMPILER
COPRTHR_DEVSUP_F_STREAM
COPRTHR_DEVSUP_F_THREAD
COPRTHR_DEVSUP_F_SIGNAL

COPRTHR_DEVSUP_F_MEM_BUFFER
COPRTHR_DEVSUP_F_MEM_MUTEX
COPRTHR_DEVSUP_F_MEM_SIGNAL
COPRTHR_DEVSUP_F_MEM_REGISTER
COPRTHR_DEVSUP_F_MEM_FIFO
COPRTHR_DEVSUP_F_MEM_STACK

COPRTHR_DEVSUP_F_MEM_PROT
COPRTHR_DEVSUP_F_MEM_OFFSET
```

Alternatively, the following wrappers are provided purely for convenience to simplify the testing for suport, returning 1 if the functionality is suported and 0 if it is not.

```
int coprthr_testsup_runtime( coprthr_dev_t dev);
int coprthr_testsup_compiler( coprthr_dev_t dev);
int coprthr_testsup_stream( coprthr_dev_t dev);
int coprthr_testsup_thread( coprthr_dev_t dev);
int coprthr_testsup_signal( coprthr_dev_t dev);
int coprthr_testsup_mem_buffer( coprthr_dev_t dev);
int coprthr_testsup_mem_mutex( coprthr_dev_t dev);
int coprthr_testsup_mem_signal( coprthr_dev_t dev);
int coprthr_testsup_mem_register( coprthr_dev_t dev);
int coprthr_testsup_mem_fifo( coprthr_dev_t dev);
int coprthr_testsup_mem_stack( coprthr_dev_t dev);
int coprthr_testsup_mem_prot( coprthr_dev_t dev);
int coprthr_testsup_mem_offset( coprthr_dev_t dev);
```

5.5 Locking a Device

SYNOPSIS

```
#include <coprthr.h>
#include <coprthr_dev.h>

coprthr_dev_t coprthr_getdev( const char* path, int flags );

int coprthr_devlock( coprthr_dev_t dev, int flags );

int coprthr_devunlock( coprthr_dev_t dev, int flags );

Link with -lcoprthr
```

DESCRIPTION

The device struct for a co-processor device can be obtained using the coprthr_getdev() call. The `path` argument may specify a device file associated with the physical device. Alternatively, one of several pre-defined macros may be used to identify specific supported devices (that may or may not be present on the platform). The pre-defined macros are:

```
COPRTHR_ARCH_ID_X86_64
COPRTHR_ARCH_ID_I386
COPRTHR_ARCH_ID_ARM32
COPRTHR_ARCH_ID_E32_EMEK
COPRTHR_ARCH_ID_E32
COPRTHR_ARCH_ID_MIC
```

Prior to performing any low-level operation on a device the programmer should acquire a lock on the device. This lock should be held until the device is no longer being used at which point *all* resources and other constructs instantiated during the use of the device will be invalidated.

The flag `COPRTHR_DEVLOCK_NOWAIT` may be used to request that the call not block in the event that the device is temporarily unavailable.

The flag `COPRTHR_DEVLOCK_NOINIT` is used to prevent initializing the device, and is used in conjunction with the flag `COPRTHR_DEVUNLOCK_PERSIST` described below.

When the device is no longer needed the following call should be used to release the lock on the device, invalidating any resources and constructs instantiatd during it use.

The exception to the normal behavior may be requested using the the flag `COPRTHR_DEVUNLOCK_PERSIST` which will cause all resources to persist in anticipation of a lock on the device being re-acquired in the future. The subsequent call to coprthr_devlock() must set the flag `COPRTHR_DEVLOCK_NOINIT` to prevent the device from being initialized.

5.6 Device Memory Allocation

SYNOPSIS

```
#include <coprthr.h>
#include <coprthr_dev.h>

coprthr_mem_t coprthr_devmemalloc( coprthr_dev_t dev, void* addr, size_t nemb,
    size_t size, int flags );

void coprthr_devmemfree( coprthr_dev_t dev, coprthr_mem_t mem );

size_t coprthr_memsize( coprthr_mem_t mem );

void* coprthr_memptr( coprthr_mem_t mem, int flag );

Link with -lcoprthr
```

DESCRIPTION

Device memory may be allocated using a generalized allocator coprthr_devmemalloc(). The allocation of a specific type of memory is selected using one of the following flags:

```
COPRTHR_DEVMEM_TYPE_BUFFER
COPRTHR_DEVMEM_TYPE_MUTEX
COPRTHR_DEVMEM_TYPE_SIGNAL
COPRTHR_DEVMEM_TYPE_REGISTER
COPRTHR_DEVMEM_TYPE_FIFO
COPRTHR_DEVMEM_TYPE_STACK
```

Additionally, the certain types of allocated memory may have read/write protection enabled using the following flags:

```
COPRTHR_DEVMEM_PROT_ENABLED
COPRTHR_DEVMEM_PROT_READ
COPRTHR_DEVMEM_PROT_WRITE

COPRTHR_DEVMEM_FIXED
```

The allocated device memory may be released using coprthgr_devmemfree().

For convenience, the size of any allocation can be obtained with the coprthr_memsize() call. In many cases, the programmer will need the device memory address associated with a specific allocation. A typical example is for passing pointer arguments to functions executed on the device. The device address may be obtained using the coprthr_memptr() call.

5.7 Accessing device memory

SYNOPSIS

```
#include <coprthr.h>
#include <coprthr_dev.h>

size_t coprthr_devread( coprthr_dev_t dev, coprthr_mem_t mem,
    size_t offset, void* buf, size_t len, int flags);

size_t coprthr_devwrite( coprthr_dev_t dev, coprthr_mem_t mem,
    size_t offset, void* buf, size_t len, int flags);

size_t coprthr_devcopy( coprthr_dev_t dev, coprthr_mem_t mem_src,
    size_t offset_src, coprthr_mem_t mem_dst, size_t offset_dst,
    size_t len, int flags);

Link with -lcoprthr
```

DESCRIPTION

The functions coprthr_devread, coprthr_devwrite(), and coprthr_devcopy() are provided to access device memory. The coprthr_devwrite() call writes the contents of a host buffer to a device memory allocation. The coprthr_devread() call reads the contents of a device memory allocation to a host buffer. The coprthr_devcopy() copies the contents of one device memory allocation to another.

Some devices may not support offsets into device memory. Each call returns the number of bytes successfully copied. If a non-zero value is provided for the offset arguement targeting a device that does not support this capability, a zero is returned and `errno` is set to ENOTSUP.

5.8 Executing Code on a Co-Processor Device

SYNOPSIS

```
#include <coprthr.h>
#include <coprthr_dev.h>

int coprthr_devexec( coprthr_dev_t dev, int nthr, void* reserved,
    coprthr_kernel_t krn, unsigned int narg, void** args);

Link with -lcoprthr
```

DESCRIPTION

The coprthr_devexec() call is used to execute a kernel `krn` on a co-processor device using `nthr` threads. The arguments `narg` and `args` specify the number of arguments and an array of pointers to the arguments to be used when the kernel is called.

5.9 Cross-compilation

SYNOPSIS

```
#include <coprthr.h>
#include <coprthr_cc.h>

coprthr_program_t coprthr_devcompile( coprthr_dev_t dev, char* src, size_t len,
    char* opt, char** log );

coprthr_sym_t
    coprthr_getsym( coprthr_program_t prg, const char* symbol );

Link with -lcoprthr -lcoprthrcc
```

DESCRIPTION

The coprthr_devcompile() call is a convenience wrapper for the cross-compilation interface call coprthr_cc() and will compile code targeting the device associated with the device `dev`.

The coprthr_getsym() call is used to get the named symbol in the program, e.g., to get a handle to a given device kernel.

---

6 Examples

The following four examples provide a glimpse of how the COPRTHR API can be used.

6.1 Example #1

As a simple example showing the use of these calls, the program below performs the following steps: 1) check cross-compiler version, 2) check available cross-compiler targets, 3) compile a simple thread function creating a a program struct suitable for use with the COPRTHR API run-time, 4) write the compiled binary object to a file, and 5) read the written file creating a new program struct.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "coprthr.h"
#include "coprthr_cc.h"

/* define a simple thread function for adding two vectors */
char src[] = \
    "#include <coprthr.h>" \
    "void\n my_kern( float* a, float* b, float* c) {\n" \
    "  int tid = coprthr_get_thread_index();\n" \
    "  c[tid] = a[tid]+b[tid];\n" \
    "}\n";

int main()
{
    int i;
    coprthr_program_t prg, prg2;

    /* get cross-compiler version info */
    char* log = 0;
    coprthr_cc(0,0,"--version",&log);
    printf("%s\n",log);
    free(log);

    /* get cross-compiler supported targets */
    log = 0;
    coprthr_cc(0,0,"--targets",&log);
    printf("%s\n",log);
    free(log);

    /* compile program for Epiphany RISC array */
    prg = coprthr_cc(src,sizeof(src),COPRTHR_CC_TARGET_E32,0);
    if (prg) {

        /* write out the binary */
        coprthr_cc_write_bin("./bin_e32.o",prg,0);

        /* then read it back into a new program struct */
        prg2 =  coprthr_cc_read_bin( "./bin_e32.o", 0 );
    }

}
```

6.2 Example #2

In this example a simple kernel that adds two vectors is compiled and executed using 16 threads.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "coprthr.h"

#define SIZE 16

char src[] = \
   "__kernel void\n" \
   "my_kern( float* a, float* b, float* c) {\n" \
   "  int idx = coprthr_get_thread_index();\n" \
   "  c[idx] = a[idx]+b[idx];\n" \
   "}\n";

int main()
{
   int i;

   int dd = coprthr_dopen(COPRTHR_DEVICE_X86_64,COPRTHR_O_STREAM);

   coprthr_program_t prg = coprthr_dcompile(dd,src,sizeof(src),"",0);
   coprthr_sym_t krn = coprthr_getsym(prg,"my_kern");

   float* a = (float*)malloc(SIZE*sizeof(float));
   float* b = (float*)malloc(SIZE*sizeof(float));
   float* c = (float*)malloc(SIZE*sizeof(float));

   for(i=0; i<SIZE; i++) {
      a[i] = 1.0f * i;
      b[i] = 2.0f * i;
      c[i] = 0.0f;
   }

   coprthr_mem_t mema = coprthr_dmalloc(dd,SIZE*sizeof(float),0);
   coprthr_mem_t memb = coprthr_dmalloc(dd,SIZE*sizeof(float),0);
   coprthr_mem_t memc = coprthr_dmalloc(dd,SIZE*sizeof(float),0);

   coprthr_dwrite(dd,mema,a,SIZE*sizeof(float),COPRTHR_E_NOWAIT);
   coprthr_dwrite(dd,memb,b,SIZE*sizeof(float),COPRTHR_E_NOWAIT);
   coprthr_dwrite(dd,memc,c,SIZE*sizeof(float),COPRTHR_E_NOWAIT);

   unsigned int nargs = 3;
   void* args[] = { &mema, &memb, &memc };
   unsigned int nthr = SIZE;

   coprthr_dexec(dd,krn,nargs,args,nthr,0,COPRTHR_E_NOWAIT);

   coprthr_dcopy(dd,memc,memb,SIZE*sizeof(float),COPRTHR_E_NOWAIT);
```

```
   coprthr_dread(dd,memc,c,SIZE*sizeof(float),COPRTHR_E_NOWAIT);

    coprthr_dwait(dd);

   for(i=0; i<SIZE; i++)
      printf("%f + %f = %f\n",a[i],b[i],c[i]);

   coprthr_dfree(dd,mema);
   coprthr_dfree(dd,memb);
   coprthr_dfree(dd,memc);

   free(a);
   free(b);
   free(c);

   coprthr_dclose(dd);
}
```

6.3 Example #3

In this example two kernels, one adding vectors and one subtracting vectors, are executed simultaneously.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "coprthr.h"
#include "test.h"

#define SIZE 256

char src_add[] = \
   "__kernel void\n" \
   "my_kern( float* a, float* b, float* c) {\n" \
   "  int gtid = get_global_id(0);\n" \
   "  c[gtid] = a[gtid]+b[gtid];\n" \
   "}\n";

char src_sub[] = \
   "__kernel void\n" \
   "my_kern( float* a, float* b, float* c) {\n" \
   "  int gtid = get_global_id(0);\n" \
   "  c[gtid] = a[gtid]-b[gtid];\n" \
   "}\n";

int main()
{
   int i;
```

```
int dd = coprthr_dopen(TEST_COPRTHR_DEVICE,COPRTHR_O_STREAM);

coprthr_program_t prg_add
   = coprthr_dcompile(dd,src_add,sizeof(src_add),0,0);
coprthr_sym_t krn_add = coprthr_getsym(prg_add,"my_kern");

coprthr_program_t prg_sub
   = coprthr_dcompile(dd,src_sub,sizeof(src_sub),0,0);
coprthr_sym_t krn_sub = coprthr_getsym(prg_sub,"my_kern");

size_t size = SIZE;
size_t size2 = SIZE/2;

float* a = (float*)malloc(size*sizeof(float));
float* b = (float*)malloc(size*sizeof(float));
float* c = (float*)malloc(size*sizeof(float));

for(i=0; i<SIZE; i++) {
   a[i] = 1.0f * i;
   b[i] = 2.0f * i;
   c[i] = 0.0f;
}

coprthr_mem_t mema1 = coprthr_dmalloc(dd,size2*sizeof(float),0);
coprthr_mem_t memb1 = coprthr_dmalloc(dd,size2*sizeof(float),0);
coprthr_mem_t memc1 = coprthr_dmalloc(dd,size2*sizeof(float),0);

coprthr_mem_t mema2 = coprthr_dmalloc(dd,size2*sizeof(float),0);
coprthr_mem_t memb2 = coprthr_dmalloc(dd,size2*sizeof(float),0);
coprthr_mem_t memc2 = coprthr_dmalloc(dd,size2*sizeof(float),0);

coprthr_dwrite(dd,mema1,a,size2*sizeof(float),COPRTHR_E_NOWAIT);
coprthr_dwrite(dd,memb1,b,size2*sizeof(float),COPRTHR_E_NOWAIT);
coprthr_dwrite(dd,memc1,c,size2*sizeof(float),COPRTHR_E_NOWAIT);

coprthr_dwrite(dd,mema2,a+size2,size2*sizeof(float),COPRTHR_E_NOWAIT);
coprthr_dwrite(dd,memb2,b+size2,size2*sizeof(float),COPRTHR_E_NOWAIT);
coprthr_dwrite(dd,memc2,c+size2,size2*sizeof(float),COPRTHR_E_NOWAIT);

unsigned int nargs = 3;
void* args_add[] = { &mema1, &memb1, &memc1 };
void* args_sub[] = { &mema2, &memb2, &memc2 };
unsigned int nthr = size2;

coprthr_kernel_t v_krn[] = { krn_add, krn_sub };
unsigned int v_nargs[] = { nargs, nargs };
void** v_args[] = { args_add, args_sub };
unsigned int v_nthr[] = { nthr, nthr };

coprthr_dnexec(dd,2,v_krn,v_nargs,v_args,v_nthr,0,COPRTHR_E_NOWAIT);

coprthr_dread(dd,memc1,c,size2*sizeof(float),COPRTHR_E_NOWAIT);
coprthr_dread(dd,memc2,c+size2,size2*sizeof(float),COPRTHR_E_NOWAIT);
```

```
    coprthr_dwait(dd);

   for(i=0; i<SIZE; i++)
      printf("%f + %f = %f\n",a[i],b[i],c[i]);

   coprthr_dfree(dd,mema1);
   coprthr_dfree(dd,memb1);
   coprthr_dfree(dd,memc1);

   coprthr_dfree(dd,mema2);
   coprthr_dfree(dd,memb2);
   coprthr_dfree(dd,memc2);

   free(a);
   free(b);
   free(c);

   coprthr_dclose(dd);
}
```

6.4 Example #4

This example shows a simple test of the pthreads extension to co-processors. A thread is created on the co-processor that requires acquiring a mutex prior to performing a calculation. The host code aquires the same mutex prior to thread cration, thereby blocking the thread from completion. The host code then waits 3 seconds, changes the input to a trivial calculation, and then releases the mutex. The examples demonstrates the use of mutexes since without proper operation the wrong value will be calulated on the device.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "coprthr.h"
#include "coprthr_thread.h"

#include "test.h"

struct my_args {
   void* mtx;
   int data;
};


char src[] = \
   "#include <coprthr.h>\n" \
   "typedef struct { void* mtx; int data; } my_args_t;\n" \
   "__kernel void\n" \
```

```
   "my_thread( void* p) {\n" \
   "  my_args_t* pargs = (my_args_t*)p;\n" \
   "  int data = pargs->data;\n" \
   "  coprthr_mutex_lock(pargs->mtx);\n" \
   "  pargs->data = pargs->data - 332211;\n" \
   "  coprthr_mutex_unlock(pargs->mtx);\n" \
   "}\n";

int main()
{
   /* open device for threads */

   int dd = coprthr_dopen(COPRTHR_DEVICE_E32,COPRTHR_O_THREAD);

   if (dd<0) {
      printf("deice open failed\n");
      exit(-1);
   }


   /* compile thread function */

   char* log = 0;
   coprthr_program_t prg = coprthr_compile(dd,src,sizeof(src),"",&log);
   if (log) printf("%s",log);
   coprthr_sym_t thr = coprthr_getsym(prg,"my_thread");


   /* create mutex on device */

   coprthr_mutex_attr_t mtxattr;
   coprthr_mutex_attr_init(&mtxattr);
   coprthr_mutex_attr_setdevice(&mtxattr,dd);

   coprthr_mutex_t mtx;
   coprthr_mutex_init(&mtx,&mtxattr);

   coprthr_mutex_attr_destroy(&mtxattr);


   /* allocate memory on device and write a value */

   struct my_args args;
   args.data = 997766;
   args.mtx = coprthr_devmemptr( (coprthr_mem_t)mtx);
   coprthr_mem_t mem = coprthr_dmalloc(dd,sizeof(struct my_args),0);
   coprthr_dwrite(dd,mem,&args,sizeof(struct my_args),COPRTHR_E_NOW);


   /* lock the mutex on the device */

   coprthr_mutex_lock(&mtx);
```

```
    /* create thread */

    coprthr_attr_t attr;
    coprthr_td_t td;
    void* status;

    coprthr_attr_init( &attr );
    coprthr_attr_setdetachstate(&attr,COPRTHR_CREATE_JOINABLE);
    coprthr_attr_setdevice(&attr,dd);

    coprthr_create( &td, &attr, thr, (void*)&mem );

    coprthr_attr_destroy( &attr);


    /* wait 3 seconds to give the thread time to work */

    sleep(3);


    /* change the value stored in memory */

    args.data = 887766;
    coprthr_dwrite(dd,mem,&args,sizeof(struct my_args),COPRTHR_E_NOW);
    args.data = -1;


    /* unlock the mutex on the device */

    coprthr_mutex_unlock(&mtx);


    /* join the thread */

    coprthr_join(td,&status);

    printf("status %d\n",(int)status);


    /* read back value from memory on device */

    coprthr_dread(dd,mem,&args,sizeof(struct my_args),COPRTHR_E_NOW);
    fprintf(stderr,"data %d 0x%x\n",args.data,args.data);


    /* clean up */

    coprthr_mutex_destroy(&mtx);

    coprthr_dfree(dd,mem);

    coprthr_dclose(dd);
}
```

Document revision 1.6.0.0