Brown Deer Technology

COPRTHR-2
API Reference

This document describes the COPRTHR-2 API for *host* and *device* code. The *host* API provides an interface to the coprocessor allowing control and the offload of parallel work to the coprocessor. The *device* API supports the development of parallel algorithms executed on the coprocessor device.

The API reference is organized as follows. The basic host API using an ordered command queue for device operations is described in Section 1. Support for a Pthreads interface extended to coprocessors is described in Section 1.6. The basic coprocessor device API used for programming the threads executed on the coprocessor is described in Section 2. The threaded MPI coprocessor device API supports a parallel programming model for the coprocessor sing standard MPI syntax and is described in Section 3.

Document revision b (20160629)

Contents

1 Host API

The COPRTHR-2 host API provides an interface for accessing a coprocessor device from the host platform including device memory allocation similar to malloc() and a stream model for non-blocking asynchronous device operations.

1.1 Device Initialization

Synopsis:

```
#include <coprthr.h>
int coprthr_dopen( const char* path, int flags )
int coprthr_dclose( int dd )
Link with -lcoprthr
```

Functions:

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

Open a coprocessor device identified by *path* and return a device descriptor that may be used in subsequent operations on the device. Here *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_E32 for the Epiphany-III coprocessor. The *flags* argument controls the behavior of the opened device.

The COPRTHR_O_NONBLOCK flag causes 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 coprocessor device should be opened. Finally, the flag COPRTHR_O_DEFAULT may be used to select the default flags configured by the installation.

`int coprthr_dclose( int dd )`

Close the coprocessor devices associated with the device descriptor *dd* that was returned from the coprthr_dopen( ) call.

1.2 Device Memory Management with UVA support

The following calls are used for memory allocation and moving data on the coprocessor device. If Unified Virtual Address-space (UVA) support is enabled the device pointer returned from `coprthr_devmemptr()` can be dereferenced directly without any specialized calls for accessing the allocated device memory.

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 )

void* coprth_devmemptr( coprthr_mem_t mem )

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
```

Functions:

`coprthr_mem_t` **`coprthr_dmalloc`**`( int dd, size_t size, int flags )`

Allocate memory associated with the coprocessor device specified by the device descriptor *dd*. The allocator is similar to conventional `malloc()` extended to a coprocessor device. The `coprthr_dmalloc()` call returns an opaque memory object of type `coprthr_mem_t` that may be used in subsequent calls.

`coprthr_mem_t` **`coprthr_drealloc`**`( int dd, coprthr_mem_t mem, size_t size, int flags )`

Resize an existing memory allocation associated with the coprocessor device specified by the device descriptor *dd*. Behavior is similar to conventional `realloc()` extended to a coprocessor device. The `coprthr_drealloc()` call returns an opaque memory object of type `coprthr_mem_t` that may be used in subsequent calls.

`void` **`coprthr_dfree`**`( int dd, coprthr_mem_t mem )`

Free an existing memory allocation associated with the coprocessor device specified by the device descriptor *dd.*

`void*` **`coprthr_devmemptr`**`( coprthr_mem_t mem )`

Return the device pointer associated with the opaque memory object. This is the address of the memory allocation as seen by the coprocessor device. With UVA suport this same address can be used on the host platform.

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

Read *len* bytes from the coprocessor device memory allocation *mem* to the host platform memory buffer *ptr*. The *offset* argument is the offset in bytes into the coprocessor device memory allocation; the use of non-zero offsets may or may not be supported by a given device.

Note: with UVA support enabled this call may be replaced with a conventional `memcpy()`.

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

Write *len* bytes to the coprocessor device memory allocation *mem* from the host platform memory buffer *ptr*. The *offset* argument is the offset in bytes into the coprocessor device memory allocation; the use of non-zero offsets may or may not be supported by a given device.

Note: with UVA support enabled this call may be replaced with a conventional `memcpy()`.

```
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 )
```

Copy the *len* bytes to the coprocessor device memory allocation *mem* from the host platform memory buffer *ptr*. The offset arguments are the offset in bytes into the respective coprocessor device memory allocations; the use of non-zero offsets may or may not be supported by a given device.

Note: with UVA support enabled this call may be replaced with a conventional `memcpy()`.

1.3 Executing Threads on a Coprocessor Device

The following calls are used for executing threads on a coprocessor device.

Synopsis:

```
#include <coprthr.h>
coprthr_event_t coprthr_dexec( int dd, unsigned int nthr, coprthr_kernel_t krn,
    void* pargs, int flags )
coprthr_event_t coprthr_dexecv( int dd, char* path, char* const argv[], int flags )
Link with -lcoprthr
```

Functions:

```
coprthr_event_t coprthr_dexec( int dd, unsigned int nthr,
    coprthr_kernel_t krn, void* pargs, int flags )
```

Execute the specified kernel on the coprocessor device specified by the device descriptor *dd* using *nthr* threads with Pthread-style argument passing.

```
coprthr_event_t coprthr_dexecv( int dd, char* path, char* const argv[],
    int flags )
```

Load and execute the coprocessor kernel specified by *path* on the coprocessor device specified by the device descriptor *dd* with argv-style argument passing. This call is a direct extension of the Linux command execv. The number of threads used on the coprocessor is determined by the environment variable `COPRTHR_DEVICE_NTHR` which can be controlled using the run-time shell command *coprsh*.

See Also: `coprthr_ncreate()`, `coprthr_mpiexec()`, `coprsh`.

1.4 Event Synchronization

The following calls are used for synchronization of operations between the host platform and a coprocessor device.

Synopsis:

```
#include <coprthr.h>
int coprthr_dwaitev( int dd, coprthr_event_t ev )
int coprthr_dwait( int dd )
Link with -lcoprthr
```

Functions:

`int coprthr_dwaitev( int dd, coprthr_event_t ev )`

Block until the operation associated with the event *ev* has completed on the specified by the device descriptor *dd*.

`int coprthr_dwait( int dd )`

Block until all operations scheduled on the coprocessor device specified by the device descriptor *dd* have completed.

1.5 Cross-compilation

The following calls will be used for run-time cross-compilation. At present this feature is not enabled. However the `coprthr_getsym()` call must be used to identify an entry point into a compiled coprocessor binary.

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
```

Functions:

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

Cross-compile *src* for the coprocessor device specified by the device descriptor *dd*.

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

Get the named symbol in the program binary. A common use of this call is to get the entry point or kernel from a compiled binary targeting a coprocessor device.

1.6 Pthreads for Coprocessors

This section describes the extension of Pthreads to coprocessors supported by COPRTHR-2.

By design, the API for the extension of Pthreads to coprocessors mirrors that of conventional POSIX threads calls in every possible way. Therefore the syntax and behavior of most calls does not differ from that of the corresponding Pthreads call.

The mechanism for maintaining near transparency with Pthreads is to attach the device descriptor to the conventional Pthreads attribute used in the creation of Pthread execution objects. The `coprthr_attr_setdevice()` call is introduced for this purpose.

The `coprthr_attr_setinit()` call is also added to allow control over what happens when a thread is created. By convention, Pthreads implicitly launches threads at the time of creation without an explicit "execute" call. 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` which is the default behavior.

Note: Pthread mutex and conditional variable support will be enabled in a future software release.

Synopsis:

```
#include <coprthr.h>
#include <coprthr_thread.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 )
int coprthr_attr_setinit( coprthr_td_attr_t* attr, int action )
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
```

Functions:

`int` **`coprthr_attr_init`**`( coprthr_td_attr_t* attr )`

Initialize thread attributes object.

`int` **`coprthr_attr_destroy`**`( coprthr_td_attr_t* attr )`

Destroy thread attributes object.

`int` **`coprthr_attr_setdetachstate`**`( coprthr_td_attr_t* attr, int state )`

Set detach state attribute in thread attributes object.

`int` **`coprthr_attr_setdevice`**`( coprthr_td_attr_t* attr, int dd )`

Set device descriptor in the thread attributes object.

`int` **`coprthr_attr_setinit`**`( coprthr_td_attr_t* attr, int action )`

Set initialization behavior in the thread attributes object. Allowable actions include `COPRTHR_A_CREATE_SUSPEND` and `COPRTHR_A_CREATE_EXECUTE`.

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

Create a new thread.

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

Create *nthr* new threads.

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

Join with a terminated thread.

2 Coprocessor Device API

The coprocessor device API is described below and may be used for developing code for execution on the coprocessor device. Compilation requires the use of the *coprcc* compiler front-end, which is described in *COPRTHR-2 Development Tools*.

2.1 Threads and Cores

The calls below that may be used for extracting information about threads and the cores they are executing on.

Synopsis:

```
#include <coprthr.h>
unsigned int coprthr_get_num_threads( void )
int coprthr_get_thread_id( void )
unsigned int coprthr_corenum( void )
unsigned int coprthr_coremap( unsigned int n)
unsigned int coprthr_threadmap( unsigned int n)
Link with -lcoprthr
```

Functions:

`unsigned int coprthr_get_num_threads( void )`

Gets the number of threads in the thread group to which the executing thread belongs.

`int coprthr_get_thread_id( void )`

Returns the thread ID for the executing thread.

`unsigned int coprthr_corenum( void )`

Returns the physical core number associated with the core executing the current thread.

`unsigned int coprthr_coremap( unsigned int tid)`

Returns the core number mapped to the thread ID *tid*.

`unsigned int coprthr_threadmap( unsigned int n)`

Returns the thread ID mapped to the core number *n*.

2.2 Memory Management

The calls below are used for allocating memory and copying data.

Synopsis:

```
#include <coprthr.h>
int coprthr_tls_brk( void* addr)
void* coprthr_tls_sbrk( intptr_t increment )
coprthr2_event_t coprthr_memcopy_align( void* dst, void* src, size_t n, int flags )
coprthr2_event_t coprthr_memcopy2d_align( void* dst, void* src,  size_t w_dst,
    size_t w_src, size_t w, size_t h, int flags )
void coprthr_wait( coprthr2_event_t ev )
Link with -lcoprthr
```

Functions:

`int coprthr_tls_brk( void* addr)`

Sets the end of the thread local data segment (USRCORE) to the value specified by *addr*.

`void* coprthr_tls_sbrk( intptr_t increment )`

Increments the thread local data segment (USRCORE) by increment bytes. Calling with an increment of 0 can be used to find the current break point.

Below is an example showing how these calls can be used to allocate and free thread local data using the conventional brk/sbrk method:

```
void* memfree = coprthr_tls_sbrk(0); // get current break point
void* buffer = coprthr_tls_sbrk(256); // allocate buffer of 256 bytes
...
coprthr_tls_brk(memfree); // free allocated buffer
```

`coprthr2_event_t coprthr_memcopy_align( void* dst, void* src, size_t n, int flags )`

Copy *n* bytes from memory *src* to *dst*.

`coprthr2_event_t coprthr_memcopy2d_align( void* dst, void* src, size_t w_dst, size_t w_src, size_t w, size_t h, int flags )`

Copy a two-dimensional memory area of dimensions *w* and *h* from *src* to *dst* which may have different leading dimensions *w_src* and *w_dst,* respectively.

`void coprthr_wait( coprthr2_event_t ev )`

Wait for the asynchronous event *ev* to complete. This is used to wait completion of asynchronous memory movement using DMA engines.

2.3 Timers

This section describes coprocessor device timers that can be used for measuring the elapsed time of execution on a single core. These are down-counters so that the returned timer values will be decreasing.

Synopsis:

```
#include <coprthr.h>
void coprthr_ctimer_reset( void )
unsigned int coprthr_ctimer_get( void )
Link with -lcoprthr
```

Functions:

`void coprthr_ctimer_reset( void )`

Reset the ctimer.

`unsigned int coprthr_ctimer_get( void )`

Return the current value of the ctimer.

Example:

```
unsigned int t0, t1;
coprthr_ctimer_reset();
...
t0 = coprthr_ctimer_get();
// ... code to be timed...
t1 = coprthr_ctimer_get();
unsigned int elapsed = t0 – t1; // elapsed time in clock cycles
```

2.4 Synchronization

The calls below are used for inter-thread synchronization with barriers and mutexes.

Synopsis:

```
#include <coprthr.h>
void coprthr_barrier( int flags )
void coprthr_mutex_init( unsigned int* mtx, int flags )
void coprthr_mutex_unlock( unsigned int* mtx )
int coprthr_mutex_testlock( unsigned int* mtx )
int coprthr_mutex_testlock_self( unsigned int* mtx, )
int coprthr_mutex_trylock( unsigned int* mtx )
int coprthr_mutex_trylock_self( unsigned int* mtx )
```

```
void coprthr_mutex_lock( unsigned int* mtx )
void coprthr_mutex_lock_self( unsigned int* mtx )
Link with -lcoprthr
```

Functions:

`void coprthr_barrier( int flags )`

Block until all threads reach barrier.

`void coprthr_mutex_init(unsigned int* mtx, int flags )`

Initialize mutex.

`void coprthr_mutex_unlock( unsigned int* mtx )`

Unlock mutex.

`int coprthr_mutex_testlock( unsigned int* mtx )`

Test if remote mutex can be acquired.

`int coprthr_mutex_testlock_self( unsigned int* mtx )`

Test if local mutex can be acquired.

`int coprthr_mutex_trylock( unsigned int* mtx )`

Attempt to acquire remote mutex.

`int coprthr_mutex_trylock_self( unsigned int* mtx )`

Attempt to acquire local mutex.

`void coprthr_mutex_lock( unsigned int* mtx )`

Block until remote mutex is acquired.

`void coprthr_mutex_lock_self( unsigned int* mtx )`

Block until local mutex is acquired.

2.5 Function Attributes

This section describes attributes that may be used to control placement of code within the coprocessor device binary.

Synopsis:

```
#include <coprthr.h>
__entry
__usrcore_call
```

`__usrmem_call`

Attributes:

`__entry`

Mark the entry point for thread execution. In the example below thread execution would begin with the function `my_thread()`:

```
void __entry my_thread( void* parg )
{
	int tid = coprthr_get_thread_id();
	printf("tid=%d\n",tid);
}
```

`__usrcore_call`

Mark function for placement in USRCORE segment. This allows performance-critical code to be placed for fast execution. This is the default placement.

`__usrmem_call`

Mark function for placement in USRMEM segment. This allows code that is not critical for performance to be placed in global DRAM to save core-local memory in the USRCORE segment.

2.6 User-defined Host Calls

The macro `USRCALL()` is actually part of the host API however it is described here due to its relationship with the coprocessor device API. COPRTHR-2 supports interoperability between the coprocessor device and the host Linux platform. Many calls are supported by default. It is also possible to declare a host call in a user application to be exported to the coprocessor device. This is done with the `USRCALL()` macro and is best illustrated by example.

In this example the host call foo() will be exported to the coprocessor device. First, the user-defined host call `foo()` must be defined and marked for export to the coprocessor within the host applicaiton. Then this call may be declared and executed from the coprocessor device.

`Host code:`

```
int foo( int a, int b );

USRCALL(foo,1);

int foo( int a, int b ) { return a+b; }
```

`Coprocessor device code:`

```
int foo( int a, int b );

...
```

```
int main() {
	int a,b,c;
	...
	c = foo(a,b)
	...
}
```

3 Threaded MPI

In this section the coprocessor device API supporting the threaded MPI parallel programming model is described. Note: Defining the macro `COPRTHR_MPI_COMPAT` prior to including the coprthr_mpi.h header will provide the conventional MPI aliases for all threaded MPI calls.

At present only a subset of the MPI calls are documented here.

Synopsis:

```
#include <coprthr.h>
#include <coprthr_mpi.h>
coprthr_mpi_comm_t MPI_COMM_THREAD
int coprthr_mpi_init( _coprthr_mpi_comm* comm, size_t bufsize )
int coprthr_mpi_finalize( void )
int coprthr_mpi_comm_size( coprthr_mpi_comm_t comm, int* size)
int coprthr_mpi_comm_rank( coprthr_mpi_comm_t comm, int* rank)
int coprthr_mpi_cart_create(coprthr_mpi_comm_t comm_old, int ndims, const int
    dims[], const int periods[], int reorder, coprthr_mpi_comm_t* comm_cart)
int coprthr_mpi_comm_free( coprthr_mpi_comm_t* comm)
int coprthr_mpi_cart_coords( coprthr_mpi_comm_t comm, int rank, int maxdims, int*
    coords)
int coprthr_mpi_cart_shift( coprthr_mpi_comm_t comm, int dir, int disp,
   int* rank_source, int* rank_dest )
int coprthr_mpi_sendrecv_replace( void* buf, int count,
   coprthr_mpi_datatype_t datatype, int dest, int sendtag, int source,
   int recvtag, coprthr_mpi_comm_t comm, coprthr_mpi_status_t* status )
```

Link with **`-lcoprthr_mpi`**

Optional: Compile with **`-DCOPRTHR_MPI_COMPAT`**

Functions:

`int` **`coprthr_mpi_init`**`( _coprthr_mpi_comm* comm, size_t bufsize )`

Initial MPI execution environment.

`int` **`coprthr_mpi_finalize`**`( void )`

Terminate MPI execution environment.

`int` **`coprthr_mpi_comm_size`**`( coprthr_mpi_comm_t comm, int* size)`

Return number of threads associated with the communicator.

```
int coprthr_mpi_comm_rank( coprthr_mpi_comm_t comm, int* rank)
```

Return rank of the calling thread in the communicator.

```
int coprthr_mpi_cart_create(coprthr_mpi_comm_t comm_old, int ndims, const int
    dims[], const int periods[], int reorder, coprthr_mpi_comm_t* comm_cart)
```

Create new communicator with Cartesian topology.

```
int coprthr_mpi_comm_free( coprthr_mpi_comm_t* comm)
```

Free a communicator.

```
int coprthr_mpi_cart_coords( coprthr_mpi_comm_t comm, int rank, int maxdims,
    int* coords)
```

Determines thread coordinates in Cartesian topology for the given rank.

```
int coprthr_mpi_cart_shift( coprthr_mpi_comm_t comm, int dir, int disp,
   int* rank_source, int* rank_dest )
```

Returns shifted source and destination ranks for given shift direction and amount.

```
int coprthr_mpi_sendrecv_replace( void* buf, int count,
   coprthr_mpi_datatype_t datatype, int dest, int sendtag, int source,
   int recvtag, coprthr_mpi_comm_t comm, coprthr_mpi_status_t* status )
```

Send and receive message using a single buffer.