NVIDIA DOCA UCX

Programming Guide

MLNX-15-060540 _v2.2.0 | October 2023

Table of Contents

Chapter 1. Introduction

Unified Communication X (UCX) is an optimized point-to-point communication framework.

UCX exposes a set of abstract communication primitives that utilize the best available hardware resources and offloads, such as active messages, tagged send/receive, remote memory read/write, atomic operations, and various synchronization routines. The supported hardware types include RDMA (InfiniBand and RoCE), TCP, GPUs, and shared memory.

UCX facilitates rapid development by providing a high-level API, masking the low-level details, while maintaining high-performance and scalability.

UCX implements best practices for transfer of messages of all sizes, based on the accumulated experience gained from applications running on the world's largest datacenters and supercomputers.

Chapter 2. Prerequisites

UCX runtime libraries are installed as part of the DOCA installation.

UCX is used the same way from the host and the DPU side.

Any active network device available on the system might be used by UCX, including network devices that might be unreachable to the remote peer.

If one of the destinations is not reachable via a certain network device (e.g., a BlueField cannot reach another BlueField via `tmfifo_net0`), UCX communication may fail.

To resolve this, use the UCX environment variable `UCX_NET_DEVICES` to specify which devices UCX can use. For example:

```
export UCX_NET_DEVICES=enp3s0f0s0,enp3s0f1s0
```

Or:

```
env UCX_NET_DEVICES=enp3s0f0s0,enp3s0f1s0 <UCX-program>
```

Using the command `show_gids` on the BlueField one can obtain the mlx device name and the port of an SF. Then that can be used to limit the UCX network interfaces and allow IB. For example:

```
dpu> show_gids
DEV     PORT    INDEX   GID                                     IPv4
     VER  DEV
---     ----    -----   ---                                     ------------
     ---  ---
mlx5_2  1       0       fe80:0000:0000:0000:0052:72ff:fe63:1651
     v2   enp3s0f0s0
mlx5_3  1       0       fe80:0000:0000:0000:0032:6bff:fe13:f13a
     v2   enp3s0f1s0
dpu> env UCX_NET_DEVICES=mlx5_2:1,mlx5_3:1 <UCX-program>
```

When RDMACM is not available, it is also required to list the Ethernet devices in UCX_NET_DEVICES configuration, so they could be used for TCP-based connection establishment. For example:

```
dpu> env UCX_NET_DEVICES=enp3s0f0s0,enp3s0f1s0,mlx5_2:1,mlx5_3:1 <UCX-program>
```

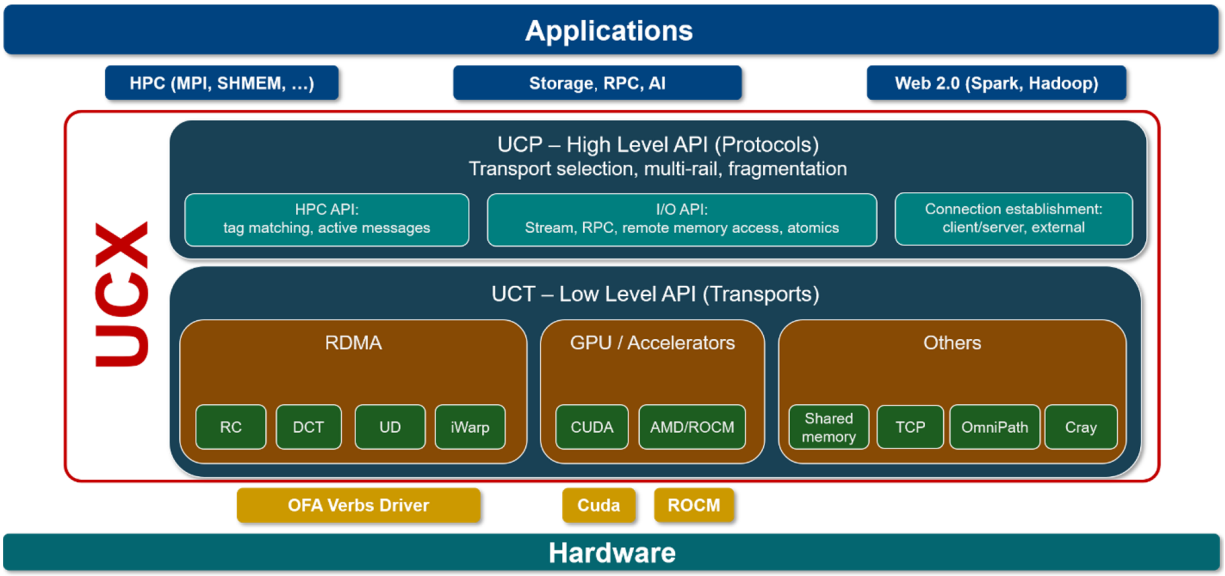
Chapter 3. Architecture

The following image describes the software layers of UCX middleware.

On the upper layer, various applications that utilize high-speed communications are built on top of the UCX high-level API (UCP).

UCP layer implements the business logic to utilize, combine, and manipulate different transports to achieve the best possible performance for different use cases. This logic decides which transports must be used for each message, which types of basic hardware communication primitives to use, how to fragment messages, etc.

UCT, the transport API, is a hardware abstraction layer that brings different types of communication devices to a common denominator. There are multiple communication primitives defined by UCT API, but each transport service may implement only some of them—preferably the ones that are natively supported by the underlying hardware. UCT users (e.g., UCP) are expected to handle the missing communication primitives defined by UCT API but not implemented by a transport service.



3.1. UCP Objects

This section describes the high-level communication objects that are used by most applications written on top of UCX.

3.1.1. UCP Context (ucp_context_h)

The context is the top-level object and it defines the scope of all other UCX objects. It is possible to create multiple contexts in the same process to have a complete separation of hardware and memory resources.

3.1.2. UCP Worker (ucp_worker_h)

The worker represents a communication state and its associated network resources. It is responsible for sending and processing incoming messages and handling all network-related events. All point-to-point connections are created in the scope of a particular worker.

A worker object can be defined to support usage from multiple threads. However, due to lock contention, the performance is better when a given worker is used most of the time from one thread.

The worker progresses communications either by active polling, waiting for asynchronous events, or a combination of both.

3.1.3. UCP Endpoint (ucp_ep_h)

The endpoint represents a connection from a local worker to a remote worker. That remote worker may be created in any place that is reachable by one of the communication networks supported by UCT layer. That could be, for example, on a different host in the fabric, the same host, on the DPU, or even in the same process.

3.1.4. UCP Listener (ucp_listener_h)

The listener binds to a network port number on the underlying operating system, and dispatches incoming connection requests. The incoming connection request can be used to create a matching endpoint on the server (passive) side or rejected and released.

3.1.5. UCP Request (ucp_request_h)

The request object is created by one of the non-blocking communications primitives in a case where the operation could not be completed immediately in-place. The application is expected to check the request for completion, either by testing it directly, or by associating a custom callback with the request.

Chapter 4. API

This section describes the main UCX APIs for high-speed communications. For the full reference, refer to UCX API specification.

UCX exposes two kinds of API: the high-level UCP API and the low-level UCT (transport) API. For most applications, it is recommended to use only the UCP API, since it relieves much of the burden of handling each transport's capabilities, limitations, and performance traits.

Many of the APIs accept a structure pointer with a `field_mask` as an argument. This method is used to provide backward ABI/API compatibility: If new function arguments are introduced, they are added as new fields in the struct, so the function signature does not change. In addition, `field_mask` specifies which struct fields are valid from the caller's (user application) perspective. UCX only accesses the fields enabled by this bitmask and uses default values for the remaining struct fields.

Some APIs require passing user-defined callbacks as a method to get notifications about specific events. Unless otherwise specified, such callbacks are called from the context of the `ucp_worker_progress()` call (detailed below), and are expected to complete quickly or defer some of their tasks to another thread (to avoid timeouts and starvation of processing from other network events).

Note: The pkg-config (`*.pc` file) for the UCX library is named `ucx`.

The following sections provide additional details about the library API.

4.1. ucs_status_t

An `enum` type that holds all UCX error codes.

4.2. ucp_init

```
ucs_status_t ucp_init(const ucp_params_t *params, const ucp_config_t *config,
 ucp_context_h *context_p)
```

Where:

- `params [in]` – points to a structure with optional parameters. All fields are optional except `features`, which must be set.

- `config [in]` – optional, can be NULL for default behavior. Configuration can be obtained by calling `ucp_config_read()`.

  Note: The supported configuration options can change between UCX versions. The full list can be obtained by running the `ucx_info` CLI tool:

  ```
  ucx_info -c -f
  ```

- `context_p [out]` – a pointer to a location in memory for the created UCP context

The function returns an error code as defined by `ucs_status_t`.

This function creates a new UCP top-level context and returns it by value in the `context_p` argument.

4.3. ucp_cleanup

```
void ucp_cleanup(ucp_context_h context_p)
```

Where:

- `context_p [in]` – a UCP context instance

This function destroys a previously created context. Prior to calling this function, any other resources created on this context (e.g., workers or endpoints) must be destroyed.

4.4. ucp_worker_create

```
ucs_status_t ucp_worker_create(ucp_context_h context, const ucp_worker_params_t
 *params, ucp_worker_h *worker_p)
```

Where:

- `context [in]` – an existing UCP context
- `params [in]` – points to a structure with configuration parameters. All fields are optional. Commonly, only the field `thread_mode` is used. Possible `thread_mode` values are as follows:
  - `UCS_THREAD_MODE_SINGLE` – only one specific thread (typically, the one that created the worker) is used to access the worker and its associated endpoints.
  - `UCS_THREAD_MODE_SERIALIZED` – multiple threads can access the worker and its associated endpoints, but only one at a time. This implies an exclusion mechanism (e.g., locking) implemented in the application. Sometimes, more expensive bus flushing instructions are needed with serialized mode, compared to single thread mode.
  - `UCS_THREAD_MODE_MULTI` – multiple threads can access the worker at any given time. UCX takes care of the locking internally. As of version 1.12, it is implemented as a global lock on the worker.
- `worker_p [out]` – a pointer to a location in memory for the created worker

The function returns an error code as defined by `ucs_status_t`.

This function creates a new UCP worker on a previously created context and returns it by value in the `worker_p` argument.

Note: When `ucp_worker_create()` succeeds, the caller is still expected to check the actual thread mode the worker was created with by `calling ucp_worker_query()` API, and take the necessary actions (for example, report an error or fallback) if the returned thread mode is not as expected to be.

4.5. ucp_worker_destroy

```
void ucp_worker_destroy(ucp_worker_h worker)
```

Where:

- `context_p [in]` – an UCP worker instance

This function destroys a previously created worker. Prior to calling this function, all associated endpoints and listeners must be destroyed.

Destroying the worker may cause communication errors on any remote peer that has an open endpoint to this worker. These errors are handled according to that endpoint's error handling configuration (detailed in section ucp_ep_create).

4.6. ucp_listener_create

```
ucs_status_t ucp_listener_create(ucp_worker_h worker, const ucp_listener_params_t
 *params, ucp_listener_h *listener_p)
```

Where:

- `worker [in]` – an existing UCP worker
- `params [in]` – points to a structure with configuration parameters. The fields `sockaddr` and `conn_handler` are mandatory, but the rest of the fields are optional.
  - `sockaddr` – specifies IPv4/IPv6 address to listen for connections. The semantics are similar to the built-in `bind()` function. `INADDR_ANY/INADDR6_ANY` can be used to listen on all network interfaces. If the port number is set to 0, a random unused port is selected. The actual port number can be obtained by calling the `ucp_listener_query()` API.
  - `conn_handler` – a callback for handling incoming connection requests along with an associated user-defined argument. The callback type is defined as:

    ```
    void (*ucp_listener_conn_callback_t) (ucp_conn_request_h conn_request, void
     *arg)
    ```

    Whenever a remote endpoint is created through this listener, this callback is called on the listener side with a new `conn_request` object representing the incoming connection, and the user-defined argument `arg` that is passed to `ucp_listener_create()`.

    The callback is expected to process this connection request by either creating an endpoint for it (pass `conn_request` as a parameter to `ucp_ep_create`, including

on a different worker), or rejecting and destroying it (call `ucp_listener_reject`). This does not have to happen immediately. The callback may put the connection request on an internal application queue and process it later.

- `listener_p [out]` – a pointer to a location in memory for the created listener

The function returns an error code as defined by `ucs_status_t`.

This function creates a new listener object to accept incoming connections on a specific network port, and returns it by value in the `listener_p` argument.

4.7. ucp_listener_destroy

```
void ucp_listener_destroy(ucp_listener_h listener_p)
```

Where:

- `listener_p [in]` – a listener instance

This function destroys a previously created listener. Prior to calling this function, any connection requests that were reported by `conn_handler` are expected to be processed. Pending connection requests that have not been reported to the application yet, or new connection requests that arrive after this function is called, are rejected.

4.8. ucp_ep_create

```
ucs_status_t ucp_ep_create(ucp_worker_h worker, const ucp_ep_params_t *params,
 ucp_ep_h *ep_p)
```

Where:

- `worker [in]` – an existing UCP worker
- `params [in]` – Points to a structure with configuration parameters. A creation mode field must be set. Other fields are optional. Commonly used fields are described in the following subsections.
- `ep_p [in]` – a pointer to a location in memory for the created endpoint

The function returns an error code as defined by `ucs_status_t`.

This function creates a new connection to a remote peer and returns it by value in the `ep_p` parameter. The new endpoint can be used for communication immediately after it is created, though some operations may be queued internally and sent after the underlying connection is established.

4.8.1. Create Modes (ucp_ep_params_t)

There are three ways the endpoint can be created:

- Client connects to a remote listener

  In this case, the `sockaddr` field specifies the remote IPv4/IPv6 address and port number. The `flags` field must be enabled and must include the

`UCP_EP_PARAMS_FLAGS_CLIENT_SERVER` flag. Optionally, from UCX version 1.13 on, the `local_sockaddr` field may be used to specify a local source device address to bind to.

- Server creates an endpoint due to an incoming connection request

  In this case, the `conn_request` field must be set to this connection request. Such endpoint can optionally be created on a different worker, not the same one this connection request was accepted on.

- Create an endpoint to a specific worker address

  In this case, the field `address` must be set to point to a remote worker's address. That address (and its length) must be obtained on the remote side by calling `ucp_worker_query()` and sent using an application-defined method (e.g., TCP socket, or other existing communication mechanism). The internal structure of the address is opaque and may change in different versions.

4.8.2. User-defined Error Handling (ucp_ep_params_t)

By default, unexpected errors on the connection (e.g., network disconnection or aborted remote process) generate a fatal failure. To enable graceful error handing, several parameters must be set during endpoint creation:

- The `err_mode` field must be set to `UCP_ERR_HANDLING_MODE_PEER`. This guarantees that send requests are always completed (successfully or error). Otherwise, network errors are considered fatal and abort the application without giving it a chance to perform cleanup or fallback flows.
- The `err_handler.cb` field must be set to a user-defined callback which is called if a connection error occurs. The error handler is defined as follows:

  ```
  void (*ucp_err_handler_cb_t)(void *arg, ucp_ep_h ep, ucs_status_t status)
  ```

  The callback parameters are the user-defined argument (passed in `user_data`), the endpoint handle on which the error happened, and the error code.

  After this callback, no more communications should be done on the endpoint. The application is expected to close the endpoint.

- The `user_data` field must be set to a user-defined argument passed to the `err_handler` callback

4.9. ucs_status_ptr_t

```
typedef void* ucs_status_ptr_t;
```

This function is commonly used as a return value for non-blocking operations.

The return value of `ucs_status_ptr_t` combines a status code and a request pointer which may be one of the following:

- A NULL pointer indicating that the operation has completed successfully in-place. The user-provided callback, if there is one, is not called.

- An error status, that can be detected by the `UCS_PTR_IS_ERR(status)` macro and extracted by `UCS_PTR_STATUS(status)`.
- Otherwise, the status is a request pointer which can also be detected by the `UCS_PTR_IS_PTR(status)` macro. This means that the communication operation has started (or was queued) but not yet completed. The completion is reported by calling the user-provided callback (in `ucp_request_param_t`) or through an explicit check on the request status by calling `ucp_request_check_status()`.

4.10. ucp_ep_close_nbx

```
ucs_status_ptr_t ucp_ep_close_nbx(ucp_ep_h ep, const ucp_request_param_t *param)
```

Where:

- `ep [in]` – an existing UCP endpoint
- `param [in]` – points to a structure that defines how the closing operation is performed. The `flags` field of the `param` structure specifies which method to use to close the endpoint:
  - `UCP_EP_CLOSE_MODE_FORCE` – close the endpoint immediately without attempting to flush outstanding operation. Some requests already completed on the transport level may complete successfully, others may be completed with an error status. In the latter case, it is not known whether they have reached the destination process or completed there.

    Closing an endpoint this way is equivalent to calling `close()` on a TCP socket and can generate a connection error on the remote side. Therefore, to use this mode, both the local and remote endpoints must be created with the `err_mode` parameter set to `UCP_ERR_HANDLING_MODE_PEER`.
  - `UCP_EP_CLOSE_MODE_FLUSH` – synchronize with the remote peer and flush outstanding operations. Some operations may be canceled and complete with the status `UCS_ERR_CANCELED`. However, it is guaranteed that they did not complete on the remote peer as well.

The function returns a status pointer to check the operation's status. NULL means success.

This function starts the process of closing a previously created endpoint. The function is non-blocking, and the returned value is a status pointer used to indicate when the endpoint is fully destroyed. For more information, refer to section Communications.

4.11. ucp_request_param_t

```
struct ucp_request_param_t {
 uint32_t op_attr_mask;
 uint32_t flags;
 union ucp_request_param_t cb;
 void *user_data;
 ucp_datatype_t datatype;
 /* Some other fields that are rarely used */
 …
```

```
}
```

Where:

- `op_attr_mask [in]` – mask of enabled fields and several control flags
- `flags [in]` – operation-specific flags. Each API method defines its own set of flags for this field.
- `cb [in]` – callback for when the operation is completed
- `user_data [in]` – user-defined argument passed to the completion callback
- `datatype [in]` – may be used to specify a custom data layout for the data buffer (not user_data) that is provided to the communication API. If this parameter is not set, the data buffer is treated as a contiguous byte buffer.

The fields of `ucp_request_param_t` specify several common attributes and flags that are used to control how the communications request is allocated and completed. This is aimed to optimize different use-cases.

4.12. ucp_worker_progress

```
unsigned ucp_worker_progress(ucp_worker_h worker)
```

Where:

- `worker [in]` – an existing UCP worker

The function returns a non-zero value if any communication has been progressed. Otherwise, it returns zero.

This function progresses outstanding communications on the worker. This includes polling hardware and shared memory queues, calling callbacks, pushing pending operations to the network devices, advancing the state of complex protocols, progressing connection establishment process, and more.

Though some transports, such as RDMA, offload do much of the heavy lifting, the initiation and completion of communication operations still must be performed explicitly by the process. UCX does not spawn additional progress threads. Instead, it is expected that the upper-layer application spawns its own progress thread, as needed, to call `ucp_worker_progress()`.

4.13. ucp_am_send_nbx

```
ucs_status_ptr_t ucp_am_send_nbx(ucp_ep_h ep, unsigned id, const void *header,
           size_t header_length, const void *buffer,
           size_t count, const ucp_request_param_t *param)
```

Where:

- `ep [in]` – connection to send the active message on. Previously returned from `ucp_ep_create()`.
- `id [in]` – active message identifier. This is an arbitrary 16-bit integer value defined by the application and used to select the active message callback to call on the

receiver side. This allows handling different types of messages by different callback functions.

- `header [in]` – pointer to a user-defined header for an active message
- `header_length [in]` – length of the header to send. Usually, the header is small and, in any case, it should be no larger than the `max_am_header` worker attribute, as returned from `ucp_worker_query()`. The header size could vary depending on the available transports and is usually expected to be at least 256 bytes.
- `buffer [in]` – pointer to the active message payload
- `count [in]` – number of elements in the payload buffer. By default, each element is a single byte, so this is the byte-length of the buffer. Other data layouts, such as IO vector (IOV) list, could be specified by `param->datatype`.
- `param [in]` – additional parameters controlling request completion semantics. The relevant field is only `flags` and it can be set to a combination of the following flags:
  - `UCP_AM_SEND_FLAG_REPLY` – force passing `reply_ep` to the callback on the receiver side. This can increase the internal header size and add some overhead.
  - `UCP_AM_SEND_FLAG_EAGER` – force using eager protocol (details below)
  - `UCP_AM_SEND_FLAG_RNDV` – force using rendezvous protocol (details below)

    The active message can be sent either by the eager or rendezvous protocol. Eager protocol means the data buffer is available on the receiver immediately during the callback, while the rendezvous protocol requires fetching the data using an additional call to `ucp_am_recv_data_nbx()`, allowing it to be placed directly to an application-selected buffer. By default, smaller messages are sent via eager protocol, and larger messages use rendezvous protocol. This can be overridden using `UCP_AM_SEND_FLAG_EAGER` or `UCP_AM_SEND_FLAG_RNDV`.

Note: `UCP_AM_SEND_FLAG_EAGER` and `UCP_AM_SEND_FLAG_RNDV` are mutually exclusive.

The function returns a status pointer to check the operation's status. NULL means success.

This function initiates sending of an active message from the initiator side. As a result, a designated callback (registered by `ucp_worker_set_am_recv_handler`) is called on the receiver side to handle this message. The function is non-blocking, so if the send operation is not completed immediately, a request handle is retuned.

4.14. ucp_worker_set_am_recv_handler

```
ucs_status_t ucp_worker_set_am_recv_handler(ucp_worker_h worker, const
 ucp_am_handler_param_t *param)
```

Where:

- `worker [in]` – an existing UCP worker
- `param [in]` – set callback configurations. See more below.

The function returns a non-zero value if any communication has been progressed. Otherwise, it returns zero.

This function registers a callback for processing active messages on the given worker.

The following are the mandatory fields to set in `param`:

- `id` – active message identifier to bind with the registered callback. Callback is invoked when receiving incoming messages with the same ID.
- `arg` – a user-defined argument to pass to the active message callback
- `cb` – a user-defined callback to invoke when an active message arrives. The callback is defined as:

```
ucs_status_t (*ucp_am_recv_callback_t)(void *arg, const void *header,
               size_t header_length, void *data,
               size_t length,
               const ucp_am_recv_param_t *param)
```

The following are the parameters passed from UCX to the callback:

- `arg` – the same user-defined argument passed to `ucp_worker_set_am_recv_handler`
- `header` – points to the active message header as defined by the sender side while sending the active message. The header should be consumed by the callback since it is not valid after the callback returns.
- `header_length` – valid size of the buffer pointer by `header`
- `data` – pointer to the data or an opaque handle that can be used to fetch the data according to the `UCP_AM_RECV_ATTR_FLAG_RNDV` flag in the field `param->recv_attr`. When flag is on, this is an opaque handle.
- `length` – length of the active message data (even if the data argument is an opaque handle and not the actual data)
- `param` – pointer to additional parameters of the incoming message. The relevant fields are:
  - `recv_attr` – flags providing more information about the incoming message
  - `reply_ep` – if `UCP_AM_RECV_ATTR_FIELD_REPLY_EP` is set in `recv_attr`, then this field holds a handle to an endpoint that can be used to send replies to the active message sender

The callback is expected to return UCS_OK if the message data has been consumed or if `UCP_AM_RECV_ATTR_FLAG_RNDV` is set in `recv_attr`. Otherwise, the if `UCP_AM_RECV_ATTR_FLAG_DATA` is set in `recv_attr`, the callback is allowed to keep the data for later processing (by adding it to an internal application queue, for example). In this case, the callback should return `UCS_INPROGRESS` as indication that the data should persist.

When a message arrives with `UCP_AM_RECV_ATTR_FLAG_RNDV` flag, the function ucp_am_recv_data_nbx must be used to fetch the data from the sender.

4.15. ucp_am_recv_data_nbx

```
ucs_status_ptr_t ucp_am_recv_data_nbx(ucp_worker_h worker, void *data_desc,
```

```
                          void *buffer, size_t count,
                const ucp_request_param_t *param)
```

Where:

- `worker [in]` – UCP worker object to use for initiating the receive operation.

  Note: The connection handle (endpoint) is not needed.

- `data_desc [in]` – handle for the data to receive. Obtained from the `data` argument for the active message callback.
- `buffer [in]` – receive buffer for the incoming data
- `count [in]` – number of elements in the payload buffer. By default, each element is a single byte, so this is the byte-length of the buffer. Other data layouts, such as the IOV list, may be specified by `param->datatype`.
- `param [in]` – additional parameters that control request allocation and completion reporting. No specific flags are needed for this function.

The function returns a status pointer to check the operation's status. NULL means success.

This function is used for rendezvous active messages. The function initiates the process of fetching data from the sender side into an application-defined receive buffer. It is expected to be used when an active message callback is called with the `UCP_AM_RECV_ATTR_FLAG_RNDV` flag set in `params->recv_attr` field.

Chapter 5. UCX Best Practices

5.1. Initialization

An application using UCX will usually create one global context (`ucp_context_h`) then create one or more workers (`ucp_worker_h`). Each worker consumes some memory for send/receive buffers, so it is not recommended to create too many workers. The rule of thumb is that the number of workers should be roughly tied to the number of CPU cores/threads.

The mapping of workers to threads is defined by the application's use case, for example:

- A single-threaded application does not need more than one worker
- A simple implementation of a multi-threaded application can create one or more workers in multi-threaded mode. These workers can be used by any thread.
- A multi-threaded application with a strong affinity between the thread and CPU core can create a dedicated worker per thread. These workers can be created in a single-threaded mode.
- Applications with many threads can implement a pool of workers and use one randomly or assign some to threads temporarily.

Note: If there are multiple workers, each of them needs to create its own set of endpoints, since every endpoint connects a specific pair of workers.

To initiate communications, the application should create endpoints (`ucp_ep_h`) connected to the remote peers. There are two main methods to create an endpoint: Either by connecting directly to a remote worker's address, or by creating a listener object (`ucp_listener_h`) and connecting to remote IP address and port. These methods are described in more detail in the ucp_ep_create section.

5.2. Communications

After initializing the UCP context, worker, and endpoints, the application can start using the endpoint for communications. Usually, endpoints are associated with application-level object that represents a connection.

Most communication operations follow a similar pattern: A non-blocking function (with _nbx suffix) receives a pointer to the `ucp_request_param_t` structure and returns `ucs_status_ptr_t`. Using a struct pointer allows extending the operations and while maintaining backward compatibility.

There are several types of communication methods supported by UCP intended for different kinds of applications. The recommended method for most applications is active messages which mean that the initiator can send arbitrary data to the responder, and the responder invokes a callback that can access this data.

Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation nor any of its direct or indirect subsidiaries and affiliates (collectively: “NVIDIA”) make no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assume no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer (“Terms of Sale”). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer’s own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, “MATERIALS”) ARE BEING PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA’s aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

Trademarks

NVIDIA, the NVIDIA logo, and Mellanox are trademarks and/or registered trademarks of Mellanox Technologies Ltd. and/or NVIDIA Corporation in the U.S. and in other countries. The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a world¬wide basis. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
http://www.nvidia.com