DOCA UROM
Table of contents

Introduction
Prerequisites
Architecture
  UROM Deployment
  UROM Framework
  Plugin Task Offloading Flow
  UROM Installation
API
  DOCA_UROM_SERVICE_FILE
  doca_urom_service
  doca_urom_service_plugin_info
  doca_urom_service_get_workers_by_gid_task
  doca_urom_service_create
  doca_urom_service_destroy
  doca_urom_service_set_max_comm_msg_size
  doca_urom_service_as_ctx
  doca_urom_service_get_plugins_list
  doca_urom_service_get_cpuset
  doca_urom_service_get_workers_by_gid_task_allocate_init
  doca_urom_service_get_workers_by_gid_task_release
  doca_urom_service_get_workers_by_gid_task_as_task
doca_urom_service_get_workers_by_gid_task_get_workers_count

doca_urom_service_get_workers_by_gid_task_get_worker_ids

doca_urom_worker

doca_urom_worker_cmd_task

doca_urom_worker_cmd_task_completion_cb_t

doca_urom_worker_create

doca_urom_worker_destroy

doca_urom_worker_set_service

doca_urom_worker_set_id

doca_urom_worker_set_gid

doca_urom_worker_set_plugins

doca_urom_worker_set_env

doca_urom_worker_as_ctx

doca_urom_worker_cmd_task_allocate_init

doca_urom_worker_cmd_task_release

doca_urom_worker_cmd_task_set_plugin

doca_urom_worker_cmd_task_set_cb

doca_urom_worker_cmd_task_get_payload

doca_urom_worker_cmd_task_get_response

doca_urom_worker_cmd_task_get_user_data

doca_urom_worker_cmd_task_as_task

doca_urom_domain

doca_urom_domain_allgather_cb_t
doça_urom_domain_req_test_cb_t

donca_urom_domain_req_free_cb_t

doca_urom_domain_oob_coll

doca_urom_domain_create

doca_urom_domain_destroy

doca_urom_domain_set_workers

doca_urom_domain_add_buffer

doca_urom_domain_set_oob

doca_urom_domain_as_ctx

Execution Model

UROM Building Blocks

Program Flow

DPU

Host

Plugin Development

Developing Offload Plugin on DPU

Creating Plugin Host Task

DOCA UROM Samples

Sample Prerequisite

Running the Sample

UROM Plugin Samples

Graph

Sandbox
UROM Program Samples

UROM Multi-worker Bootstrap

UROM Ping Pong
This guide provides an overview and configuration instructions for DOCA Unified Resources and Offload Manager (UROM) API.

Introduction

Note

This library is currently supported at alpha level only.

The DOCA Unified Resource and Offload Manager (UROM) offers a framework for offloading a portion of parallel computing tasks, such as those related to HPC or AI workloads and frameworks, from the host to the NVIDIA DPUs. This framework includes the UROM service which is responsible for resource discovery, coordination between the host and DPU, and the management of UROM workers that execute parallel computing tasks.

When an application utilizes the UROM framework for offloading, it consists of two main components: the host part and the UROM worker on the DPU. The host part is responsible for interacting with the DOCA UROM API and operates as part of the application with the aim of offloading tasks to the DPU. This component establishes a connection with the UROM service and initiates an offload request. In response to the offload request, the UROM service provides network identifiers for the workers, which are spawned by the UROM service. If the UROM service is running as a Kubernetes POD, the workers are spawned within the POD. Each worker is responsible for executing either a single offload or multiple offloads, depending on the requirements of the host application.

Prerequisites

UCX is required for the communication channel between the host and DPU parts of DOCA UROM based on TCP socket transport. This is a mechanism to transfer commands from the host to the UROM service on the DPU and receive responses from the DPU.

By default, UCX scans all available devices on the machine and selects the best ones based on performance characteristics. The environment variable `UCX_NET_DEVICES=<dev1>,`
<dev2>... would restrict UCX to using only the specified devices. For example, UCX_NET_DEVICES=eth2 uses the Ethernet device eth2 for TCP socket transport.

For more information about UCX, refer to [DOCA UCX Programming Guide](#).

## Architecture

### UROM Deployment

The diagram illustrates a standard UROM deployment where each DPU is required to host both a service process instance and a group of worker processes.

The typical usage of UROM services involves the following steps:

1. Every process in the parallel application discovers the UROM service.
2. UROM handles authentication and provides service details.
3. The host application receives the available offloading plugins on the local DPU through UROM service.
4. The host application picks the desired plugin info and triggers UROM worker plugin instances on the DPU through the UROM service.
5. The application delegates specific tasks to the UROM workers.
6. UROM workers execute these tasks and return the results.

### UROM Framework
This diagram shows a high-level overview of the DOCA UROM framework.

A UROM offload plugin is where developers of AI/HPC offloads implement their own offloading logic while using DOCA UROM as the transport layer and resource manager. Each plugin defines commands to execute logic on the DPU and notifications that are returned to the host application. Each type of supported offload corresponds to a distinct type of DOCA UROM plugin. For example, a developer may need a UCC plugin to offload UCC functionality to the DPU. Each plugin implements a DPU-side plugin API and exposes a corresponding host-side interface.
A UROM daemon loads the plugin DPU version (.so file) in runtime as part of the discovery of local plugins.

**Plugin Task Offloading Flow**

**UROM Installation**

DOCA UROM is an integral part of the DOCA SDK installation package. Depending on your system architecture and enabled offload plugins, UROM is comprised by several components, which can be categorized into two main parts: those on the host and those on the DPU.

- **DOCA UROM library components:**
  - libdoca_urom shared object – contains the DOCA UROM API
  - libdoca_urom_components_comm_ucp_am – includes the UROM communication channel interface API
DOCA UROM headers:

- DOCA UROM host interface (*doca_urom.h*) – this header includes three essential components: contexts, tasks, and plugins.
  - Service context (*doca_urom_service*) – this context serves as an abstraction of the UROM service process. Tasks posted within this context include the authentication, spawning, and termination of workers on the DPU.
  - Worker context (*doca_urom_worker*) – this context abstracts the DPU UROM worker, which operates on behalf of host application plugins (offload). Tasks posted within this context involve relaying commands from the host application to the worker on behalf of a specific offload plugin, such as offloaded functionality for communication operations.
  - Domain context (*doca_urom_domain*) – this context encapsulates a group of workers belonging to the same host application. This concept is similar to the MPI (message passing interface) communicator in the MPI programming model or PyTorch's process groups. Plugins are not required to use the UROM Domain.

- DOCA UROM plugin interface (*doca_urom_plugin.h*) – this header includes the main structure and definitions that the user can use to build both the host and DPU components of their own offloading plugins.
  - UROM plugin interface structure (*urom_plugin_iface*) – this interface includes a set of operations to be executed by the UROM worker.
- **UROM worker command structure** (*urom_worker_cmd*) – this structure defines the worker instance command format
  
- **UROM worker notification structure** (*urom_worker_notify*) – this structure defines the worker instance notification format

The following diagram shows various software components of DOCA UROM:

- **DOCA Core** – involves DOCA device discovery, DOCA progress engine, DOCA context, etc.

- **DOCA UROM Core** – includes the UROM library functionality

- **DOCA UROM Host SDK** – UROM API for the host application to use

- **DOCA UROM DPU SDK** – UROM API for the NVIDIA® BlueField® networking platform (DPU or SuperNIC) to use

- **DOCA UROM Host Plugin** – user plugin host version

- **DOCA UROM DPU Plugin** – user plugin DPU version

- **DOCA UROM App** – user UROM host application

- **DOCA UROM Worker** – the offload functionality component that executes the offloading logic

- **DOCA UROM Daemon** – is responsible for resource discovery, coordination between the host and DPU, managing the workers on BlueField
API

Info

More information is available on DOCA UROM API in the NVIDIA DOCA Library APIs.

Note

The pkg-config (*.pc file) for the UROM library is doca-urom.

The following sections provide additional details about the library API.

**DOCA_UROM_SERVICE_FILE**

This environment variable sets the path to the UROM service file. When creating the UROM service object (see doca_urom_service_create), UROM performs a look-up using this file, the hostname where an application is running, and the PCIe address of the associated DOCA device to identify the network address, and network devices associated with the UROM service.

This file contains one entry per line describing the location of each UROM service that may be used by UROM. The format of each line must be as follows:

```
<app_hostname> <service_type> <dev_hostname> <dev_pci_addr> <net,devs>
```

Example:
Fields are described in the following table:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>app_hostname</td>
<td>Network hostname (or IP address) for the node that this line applies to</td>
</tr>
<tr>
<td>service_type</td>
<td>The UROM service type. Valid type is dpu (used for all DOCA devices).</td>
</tr>
<tr>
<td>dev_hostname</td>
<td>Network hostname (or IP address) for the associated DOCA device</td>
</tr>
<tr>
<td>dev_pci_addr</td>
<td>PCIe address of the associated DOCA device. This must match the PCIe address provided by DOCA.</td>
</tr>
<tr>
<td>net_devs</td>
<td>Comma-separated list of network devices shared between the host and DOCA device</td>
</tr>
</tbody>
</table>

**doca_urom_service**

An opaque structure that represents a DOCA UROM service.

```c
struct doca_urom_service;
```

**doca_urom_service_plugin_info**

DOCA UROM plugin info structure. UROM generates this structure for each plugin on the local DPU where the UROM service is running and the service returns an array of available plugins to the host application to pick which plugins to use.

```c
struct doca_urom_service_plugin_info {
    uint64_t id;
    uint64_t version;
};
```
- **id** – Unique ID to send commands to the plugin, UROM generates this ID
- **version** – Plugin DPU version to verify that the plugin host interface has the same version
- **plugin_name** – The .so plugin file name without ".so". The name is used to find the desired plugin.

### `doca_urom_service_get_workers_by_gid_task`

An opaque structure representing a DOCA service gets workers by group ID task.

```c
struct doca_urom_service_get_workers_by_gid_task;
```

### `doca_urom_service_create`

Before performing any UROM service operation (spawn worker, destroy worker, etc.), it is essential to create a `doca_urom_service` object. A service object is created in state `DOCA_CTX_STATE_IDLE`. After creation, the user may configure the service using setter methods (e.g., `doca_urom_service_set_dev()`).

Before use, a service object must be transitioned to state `DOCA_CTX_STATE_RUNNING` using the `doca_ctx_start()` interface. A typical invocation looks like `doca_ctx_start(doca_urom_service_as_ctx(service_ctx))`.

```c
doca_error_t doca_urom_service_create(struct doca_urom_service **service_ctx);
```

- **service_ctx [in/out]** – `doca_urom_service` object to be created
- **Returns** – `DOCA_SUCCESS` on success, error code otherwise
Info

Multiple application processes could create different service objects that represent/connect to the same worker on the DPU.

doca_urom_service_destroy

Destroy a doca_urom_service object.

```c
doca_error_t doca_urom_service_destroy(struct doca_urom_service *service_ctx);
```

- `service_ctx[in]` – doca_urom_service object to be destroyed. It is created by doca_urom_service_create().
- Returns – DOCA_SUCCESS on success, error code otherwise

doca_urom_service_set_max_comm_msg_size

Set the maximum size for a message in the UROM communication channel. The default message size is 4096B.

Note

It is important to ensure that the combined size of the plugins' commands and notifications and the UROM structure's size do not exceed this maximum size.
Once the service state is running, users cannot update the maximum size for the message.

```c
struct doca_urom_service_bridge *docta_urom_service_set_max_comm_msg_size(struct doca_urom_service *service_ctx, size_t msg_size);
```

- `service_ctx[in]` – a pointer to `doca_urom_service` object to set new message size
- `msg_size[in]` – new message size to set
- Returns – `DOCA_SUCCESS` on success, error code otherwise

**docta_urom_service_as_ctx**

Convert a `docta_urom_service` object into a DOCA object.

```c
struct doca_ctx *docta_urom_service_as_ctx(struct doca_urom_service *service_ctx);
```

- `service_ctx[in]` – a pointer to `doca_urom_service` object
- Returns – a pointer to the `doca_ctx` object on success, `NULL` otherwise

**docta_urom_service_get_plugins_list**

Retrieve the list of supported plugins on the UROM service.

```c
doca_error_t docta_urom_service_get_plugins_list(struct doca_urom_service *service_ctx, const struct doca_urom_service_plugin_info **plugins, size_t *plugins_count);
```

- `service_ctx[in]` – a pointer to `doca_urom_service` object
• plugins[out] – an array of pointers to doca_urom_service_plugin_info object

• plugins_count[out] – number of plugins

• Returns – DOCA_SUCCESS on success, error code otherwise

doca_urom_service_get_cpuset

Get the allowed CPU set for the UROM service on BlueField, which can be used when
spawning workers to set processor affinity.

doca_error_t doca_urom_service_get_cpuset(struct doca_urom_service *service_ctx, doca_cpu_set_t *
cpuset);

• service_ctx[in] – a pointer to doca_urom_service object

• cpuset[out] – set of allowed CPUs

• Returns – DOCA_SUCCESS on success, error code otherwise

doca_urom_service_get_workers_by_gid_task_allocate_init

Allocate a get-workers-by-GID service task and set task attributes.


doca_error_t doca_urom_service_get_workers_by_gid_task_allocate_init(struct doca_urom_service
*service_ctx,

uint32_t gid,

doca_urom_service_get_workers_by_gid_task_completion_cb_t cb,

struct
doca_urom_service_get_workers_by_gid_task **task);

• service_ctx[in] – a pointer to doca_urom_service object
• gid[in] – group ID to set
• cb[in] – user task completion callback
• task[out] – a new get-workers-by-GID service task
• Returns – DOCA_SUCCESS on success, error code otherwise

**doca_urom_service_get_workers_by_gid_task_release**

Release a get-workers-by-GID service task and task resources.

```c
doca_error_t doca_urom_service_get_workers_by_gid_task_release(struct
doca_urom_service_get_workers_by_gid_task *task);
```

• task[in] – service task to release
• Returns – DOCA_SUCCESS on success, error code otherwise

**doca_urom_service_get_workers_by_gid_task_as_task**

Convert a doca_urom_service_get_workers_by_gid_task object into a DOCA task object.

After creating a service task and configuring it using setter methods (e.g.,
doca_urom_service_get_workers_by_gid_task_set_gid()) or as part of task allocation, the user should submit the task by calling doca_task_submit.

A typical invocation looks like
doca_task_submit(doca_urom_service_get_workers_by_gid_task_as_task(task)).

```c
struct doca_task *doca_urom_service_get_workers_by_gid_task_as_task(struct
doca_urom_service_get_workers_by_gid_task *task);
```
- task[in] – get-workers-by-GID service task
- Returns – a pointer to the doca_task object on success, NULL otherwise

`doca_urom_service_get_workers_by_gid_task_get_workers_count`:
Get the number of workers returned for the requested GID.

```c
size_t doca_urom_service_get_workers_by_gid_task_get_workers_count(struct doca_urom_service_get_workers_by_gid_task *task);
```

- task[in] – get-workers-by-GID service task
- Returns – workers ID’s array size

`doca_urom_service_get_workers_by_gid_task_get_worker_ids`:
Get service get workers task IDs array.

```c
const uint64_t *doca_urom_service_get_workers_by_gid_task_get_worker_ids(struct doca_urom_service_get_workers_by_gid_task *task);
```

- task[in] – get-workers-by-GID service task
- Returns – workers ID’s array, NULL otherwise

`doca_urom_worker`
An opaque structure representing a DOCA UROM worker context.
struct doca_urom_worker;

**doca_urom_worker_cmd_task**

An opaque structure representing a DOCA UROM worker command task context.

```c
struct doca_urom_worker_cmd_task;
```

**doca_urom_worker_cmd_task_completion_cb_t**

A worker command task completion callback type. It is called once the worker task is completed.

```c
typedef void (*doca_urom_worker_cmd_task_completion_cb_t)(struct doca_urom_worker_cmd_task *task,
                                                          union doca_data task_user_data,
                                                          union doca_data ctx_user_data);
```

- task[in] – a pointer to worker command task
- task_user_data[in] – user task data
- ctx_user_data[in] – user worker context data

**doca_urom_worker_create**

This method creates a UROM worker context.

A worker is created in a DOCA_CTX_STATE_IDLE state. After creation, a user may configure the worker using setter methods (e.g., `doca_urom_worker_set_service()`). Before use, a worker must
be transitioned to state `DOCA_CTX_STATE_RUNNING` using the `doca_ctx_start()` interface. A typical invocation looks like `doca_ctx_start(doca_urom_worker_as_ctx(worker_ctx))`.

```c
doca_error_t doca_urom_worker_create(struct doca_urom_worker **worker_ctx);
```

- `worker_ctx [in/out]` – `doca_urom_worker` object to be created
- Returns – `DOCA_SUCCESS` on success, error code otherwise

**doca_urom_worker_destroy**

Destroys a UROM worker context.

```c
doca_error_t doca_urom_worker_destroy(struct doca_urom_worker *worker_ctx);
```

- `worker_ctx [in]` – `doca_urom_worker` object to be destroyed. It is created by `doca_urom_worker_create()`.
- Returns – `DOCA_SUCCESS` on success, error code otherwise

**doca_urom_worker_set_service**

Attaches a UROM service to the worker context. The worker is launched on the DOCA device managed by the provided service context.

```c
doca_error_t doca_urom_worker_set_service(struct doca_urom_worker *worker_ctx, struct doca_urom_service *service_ctx);
```

- `service_ctx [in]` – service context to set
- Returns – `DOCA_SUCCESS` on success, error code otherwise
**doca_urom_worker_set_id**

This method sets the worker context ID to be used to identify the worker. Worker IDs enable an application to establish multiple connections to the same worker process running on a DOCA device.

Worker ID must be unique to a UROM service.

- If `DOCA_UROM_WORKER_ID_ANY` is specified, the service assigns a unique ID for the newly created worker.

- If a specific ID is used, the service looks for an existing worker with matching ID. If one exists, the service establishes a new connection to the existing worker. If a matching worker does not exist, a new worker is created with the specified worker ID.

```c
doca_error_t doca_urom_worker_set_id(struct doca_urom_worker *worker_ctx, uint64_t worker_id);
```

- `worker_id [in]` – worker ID
- Returns – `DOCA_SUCCESS` on success, error code otherwise

**doca_urom_worker_set_gid**

Set worker group ID. This ID must be set before starting the worker context.

Through service get workers by GID task, the application can have the list of workers' IDs which are running on DOCA device and that belong to the same group ID.

```c
doca_error_t doca_urom_worker_set_gid(struct doca_urom_worker *worker_ctx, uint32_t gid);
```
- gid [in] – worker group ID
- Returns – DOCA_SUCCESS on success, error code otherwise

**doca_urom_worker_set_plugins**

Adds a plugin mask for the supported plugins by the UROM worker on the DPU. The application can use up to 62 plugins.

```c
#define doca_urom_worker_set_plugins(worker_ctx, plugins)
```

- plugins[in] – an ORing set of worker plugin IDs
- Returns – DOCA_SUCCESS on success, error code otherwise

**doca_urom_worker_set_env**

Set worker environment variables when spawning worker on DPU side by DOCA UROM service. They must be set before starting the worker context.

⚠️ **Info**

This call fails if the worker already spawned on the DPU.
worker_ctx [in] – doca_urom_worker object

env [in] – an array of environment variables

count [in] – array size

Returns – DOCA_SUCCESS on success, error code otherwise

doca_urom_worker_as_ctx

Convert a doca_urom_worker object into a DOCA object.

struct doca_ctx *doca_urom_worker_as_ctx(struct doca_urom_worker *worker_ctx);

worker_ctx[in] – a pointer to doca_urom_worker object

Returns – a pointer to the doca_ctx object on success, NULL otherwise

doca_urom_worker_cmd_task_allocate_init

Allocate worker command task and set task attributes.

doca_error_t doca_urom_worker_cmd_task_allocate_init(struct doca_urom_worker *worker_ctx, uint64_t plugin, struct doca_urom_worker_cmd_task **task);

worker_ctx [in] – a pointer to doca_urom_worker object

plugin [in] – task plugin ID

task [out] – set worker command new task
• Returns – DOCA_SUCCESS on success, error code otherwise

**doca_urom_worker_cmd_task_release**

Release worker command task.

```c
doca_error_t doca_urom_worker_cmd_task_release(struct doca_urom_worker_cmd_task *task);
```

• task[in] – worker task to release

• Returns – DOCA_SUCCESS on success, error code otherwise

**doca_urom_worker_cmd_task_set_plugin**

Set worker command task plugin ID. The plugin ID is created by the UROM service and the plugin host interface should hold it to create UROM worker command tasks.

```c
void doca_urom_worker_cmd_task_set_plugin(struct doca_urom_worker_cmd_task *task, uint64_t plugin);
```

• task [in] – worker task

• plugin [in] – task plugin to set

**doca_urom_worker_cmd_task_set_cb**

Set worker command task completion callback.

```c
void doca_urom_worker_cmd_task_set_cb(struct doca_urom_worker_cmd_task *task,
```


- task [in] – worker task
- plugin [in] – task callback to set

**doca_urom_worker_cmd_task_get_payload**

Get worker command task payload. The plugin interface populates this buffer by plugin command structure. The payload size is the maximum message size in the DOCA UROM communication channel (the user can configure the size by calling `doca_urom_service_set_max_comm_msg_size()`). To update the payload buffer, the user should call `doca_buf_set_data()`.

```c
struct doca_buf *doca_urom_worker_cmd_task_get_payload(struct doca_urom_worker_cmd_task *task);
```

- task [in] – worker task
- Returns – a `doca_buf` that represents the task's payload

**doca_urom_worker_cmd_task_get_response**

Get worker command task response. To get the response's buffer, the user should call `doca_buf_get_data()`.

```c
struct doca_buf *doca_urom_worker_cmd_task_get_response(struct doca_urom_worker_cmd_task *task);
```

- task [in] – worker task
- Returns – a `doca_buf` that represents the task's response
**doca_urom_worker_cmd_task_get_user_data**

Get worker command user data to populate. The data refers to the reserved data inside the task that the user can get when calling the completion callback. The maximum data size is 32 bytes.

```c
void *doca_urom_worker_cmd_task_get_user_data(struct doca_urom_worker_cmd_task *task);
```

- task [in] – worker task
- Returns – a pointer to user data memory

**doca_urom_worker_cmd_task_as_task**

Convert a `doca_urom_worker_cmd_task` object into a DOCA task object.

After creating a worker command task and configuring it using setter methods (e.g., `doca_urom_worker_cmd_task_set_plugin()`) or as part of task allocation, the user should submit the task by calling `doca_task_submit`.

A typical invocation looks like `doca_task_submit(doca_urom_worker_cmd_task_as_task(task))`.

```c
struct doca_task *doca_urom_worker_cmd_task_as_task(struct doca_urom_worker_cmd_task *task);
```

- task[in] – worker command task
- Returns – a pointer to the `doca_task` object on success, NULL otherwise

**doca_urom_domain**
An opaque structure representing a DOCA UROM domain context.

```c
struct doca_urom_domain;
```

doca_urom_domain_allgather_cb_t

A callback for a non-blocking all-gather operation.

```c
typedef doca_error_t (*doca_urom_domain_allgather_cb_t)(void *sbuf, void *rbuf, size_t msglen, void *coll_info, void **req);
```

- `sbuf [in]` – local buffer to send to other processes
- `rbuf [in]` – global buffer to include other process's source buffer
- `msglen [in]` – source buffer length
- `coll_info [in]` – collection info
- `req [in]` – allgather request data
- Returns – `DOCA_SUCCESS` on success, error code otherwise

doca_urom_domain_req_test_cb_t

A callback to test the status of a non-blocking allgather request.

```c
typedef doca_error_t (*doca_urom_domain_req_test_cb_t)(void *req);
```

- `req [in]` – allgather request data to check status
- Returns – `DOCA_SUCCESS` on success, `DOCA_ERROR_IN_PROGRESS` otherwise
**doca_urom_domain_req_free_cb_t**

A callback to free a non-blocking allgather request.

```c
typedef doca_error_t(*doca_urom_domain_req_free_cb_t)(void *req);
```

- Returns – `DOCA_SUCCESS` on success, error code otherwise

**doca_urom_domain_oob_coll**

Out-of-band communication descriptor for domain creation.

```c
struct doca_urom_domain_oob_coll {
    doca_urom_domain_allgather_cb_t allgather;
    doca_urom_domain_req_test_cb_t req_test;
    doca_urom_domain_req_free_cb_t req_free;
    void *coll_info;
    uint32_t n_oob_indexes;
    uint32_t oob_index;
};
```

- `allgather` – non-blocking allgather callback
- `req_test` – request test callback
- `req_free` – request free callback
- `coll_info` – context or metadata required by the OOB collective
- `n_oob_indexes` – number of endpoints participating in the OOB operation (e.g., number of client processes representing domain workers)
- oob_index – an integer value that represents the position of the calling processes in the given OOB operation. The data specified by src_buf is placed at the offset "oob_index*size" in the recv_buf.

**Note**

oob_index must be unique at every calling process and should be in the range [0:n_oob_indexes).

### doca_urom_domain_create

Creates a UROM domain context. A domain is created in state DOCA_CTX_STATE_IDLE. After creation, a user may configure the domain using setter methods (e.g., doca_urom_domain_set_workers()). Before use, a domain must be transitioned to state DOCA_CTX_STATE_RUNNING using the doca_ctx_start() interface. A typical invocation looks like doca_ctx_start(doca_urom_domain_as_ctx(worker_ctx)).

```c
doca_error_t doca_urom_domain_create(struct doca_urom_domain **domain_ctx);
```

- domain_ctx [in/out] – doca_urom_domain object to be created
- Returns – DOCA_SUCCESS on success, error code otherwise

### doca_urom_domain_destroy

Destroys a UROM domain context.

```c
doca_error_t doca_urom_domain_destroy(struct doca_urom_domain *domain_ctx);
```
• domain_ctx [in] – doca_urom_domain object to be destroyed; it is created by doca_urom_domain_create()

• Returns – DOCA_SUCCESS on success, error code otherwise

doca_urom_domain_set_workers

Sets the list of workers in the domain.

```c
doca_error_t doca_urom_domain_set_workers(struct doca_urom_domain *domain_ctx, uint64_t *domain_worker_ids, struct doca_urom_worker **workers, size_t workers_cnt);
```

• domain_ctx [in] – doca_urom_domain object

• domain_worker_ids [in] – list of domain worker IDs

• workers [in] – an array of UROM worker contexts that should be part of the domain

• workers_cnt [in] – the number of workers in the given array

• Returns – DOCA_SUCCESS on success, error code otherwise

doca_urom_domain_add_buffer

Attaches local buffer attributes to the domain. It should be called after calling doca_urom_domain_set_buffers_count().

The local buffer will be shared with all workers belonging to the domain.

```c
doca_error_t doca_urom_domain_add_buffer(struct doca_urom_domain *domain_ctx, void *buffer, size_t buf_len, void *memh, size_t memh_len, void *mkey, size_t mkey_len);
```

• domain_ctx [in] – doca_urom_domain object
• buffer [in] – buffer ready for remote access which is given to the domain
• buf_len [in] – buffer length
• memh [in] – memory handle for the exported buffer. (should be packed)
• memh_len [in] – memory handle size
• mkey [in] – memory key for the exported buffer. (should be packed)
• mkey_len [in] – memory key size
• Returns – DOCA_SUCCESS on success, error code otherwise

doca_urom_domain_set_oob

Sets OOB communication info to be used for domain initialization.

doca_error_t doca_urom_domain_set_oob(struct doca_urom_domain *domain_ctx, struct
doca_urom_domain_oob_coll *oob);

• domain_ctx [in] – doca_urom_domain object
• oob [in] – OOB communication info to set
• Returns – DOCA_SUCCESS on success, error code otherwise

doca_urom_domain_as_ctx

Convert a doca_urom_domain object into a DOCA object.

struct doca_ctx *doca_urom_domain_as_ctx(struct doca_urom_domain *domain_ctx);
Execution Model

DOCA UROM uses the DOCA Core Progress Engine as an execution model for service and worker contexts and tasks progress. For more details about it please refer to this guide.

UROM Building Blocks

This section explains the general concepts behind the fundamental building blocks to use when creating a DOCA UROM application and offloading functionality.

Program Flow

DPU

Launching DOCA UROM Service

DOCA UROM service should be run before running the application on the host to offload commands to BlueField. For more information, refer to the NVIDIA DOCA UROM Service Guide.

Host

Initializing UROM Service Context

1. Create service context : Establish a service context within the control plane alongside the progress engine.

2. Set service context attributes : Specific attributes of the service context are configured. The required attribute is `doca_dev`.

- `domain_ctx[in]` – a pointer to `doca_urom_domain` object
- Returns – a pointer to the `doca_ctx` object on success, NULL otherwise
3. Start the service context: The service context is initiated by invoking the `doca_ctx_start` function.

    1. Discover BlueField availability: The UROM library identifies the available BlueField device.

    2. Connect to UROM service: The library establishes a connection to the UROM service. The connection process is synchronized, meaning that the host process and the BlueField service process are blocked until the connection is established.

    3. Perform lookup using UROM service file: A lookup operation is executed using the UROM service file. The path to this file should be specified in the `DOCA_UROM_SERVICE_FILE` environment variable. More information can be found in `doca_urom.h`.

4. Switch context state to `DOCA_CTX_STATE_RUNNING`: The context state transitions to `DOCA_CTX_STATE_RUNNING` at this point.

5. Service context waits for worker bootstrap requests: The service context is now in a state where it awaits and handles worker bootstrap requests.

```c
/* Create DOCA UROM service instance */
doca_urom_service_create(&service);

/* Connect service context to DOCA progress engine */
doca_pe_connect_ctx(pe, doca_urom_service_as_ctx(service));

/* Set service attributes */
doca_urom_service_set_max_workers(service, nb_workers)
doca_urom_service_set_dev(service, dev);

/* Start service context */
doca_ctx_start(doca_urom_service_as_ctx(service));

/* Handling workers bootstrap requests */
do {
    doca_pe_progress(pe);
} while (!are_all_workers_started);
```
Picking UROM Worker Offload Functionality

Once the service context state is running, the application can call `doca_urom_service_get_plugins_list()` to get the available plugins on the local BlueField device where the UROM service is running.

The UROM service generates an identifier for each plugin and the application is responsible for forwarding this ID to the host plugin interface for sending commands and receiving notifications by calling `urom_<plugin_name>_init(<plugin_id>, <plugin_version>).`

```c
const char *plugin_name = "worker_rdmo";
struct doca_urom_service *service;
const struct doca_urom_service_plugin_info *plugins, *rdmo_info;

/* Create and Start UROM service context. */
..

/* Get worker plugins list. */
doca_urom_service_get_plugins_list(*service, &plugins, &plugins_count);

/* Check if RDMO plugin exists. */
for (i = 0; i < plugins_count; i++) {
    if (strcmp(plugin_name, plugins[i].plugin_name) == 0) {
        rdmo_info = &plugins[i];
        break;
    }
}

/* Attach RDMO plugin ID and DPU plugin version for compatibility check */
urom_rdmo_init(rdmo_info->id, rdmo_info->version);
```

Initializing UROM Worker Context

1. Create a service context and connect the worker context to DOCA Progress Engine (PE).

2. Set worker context attributes (in the example below worker plugin is RDMO).
3. Start worker context, submitting internally spawns worker requests on the service context.

4. Worker context state changes to DOCA_CTX_STATE_STARTING (this process is asynchronous).

5. Wait until the worker context state changes to DOCA_CTX_STATE_RUNNING:
   
   1. When calling `doca_pe_progress`, check for a response from the service context that the spawning worker on BlueField is done.
   
   2. If the worker is spawned on BlueField, connect to it and change the status to running.

```c
const struct doca_urom_service_plugin_info *rdmo_info;

/* Create DOCA UROM worker instance */
doca_urom_worker_create(&worker);

/* Connect worker context to DOCA progress engine */
doca_pe_connect_ctx(pe, doca_urom_worker_as_ctx(worker));

/* Set worker attributes */
doca_urom_worker_set_service(worker, service);
doca_urom_worker_set_id(worker, worker_id);
doca_urom_worker_set_max_inflight_tasks(worker, nb_tasks);
doca_urom_worker_set_plugins(worker, rdmo_info->id);
doca_urom_worker_set_cpuset(worker, cpuset);

/* Start UROM worker context */
doca_ctx_start(doca_urom_worker_as_ctx(worker));

/* Progress until worker state changes to running or error happened */
do {
    doca_pe_progress(pe);
    result = doca_ctx_get_state(doca_urom_worker_as_ctx(worker), &state);
} while (state == DOCA_CTX_STATE_STARTING);
```

### Offloading Plugin Task
Once the worker context state is `DOCA_CTX_STATE_RUNNING`, the worker is ready to execute offload tasks. The example below is for offloading an RDMO command.

1. Prepare RDMO task arguments (e.g., completion callback).

2. Call the task function from the plugin host interface.

3. Poll for completion by calling `doca_pe_progress`.

4. Get completion notification through the user callback.

```c
int ret;
struct doca_urom_worker *worker;
struct rdmo_result res = {0};
union doca_data cookie = {0};
size_t server_worker_addr_len;
ucp_address_t *server_worker_addr;

cookie.ptr = &res;
res.result = DOCA_SUCCESS;

ucp_worker_create(*ucp_context, &worker_params, server_ucp_worker);
ucp_worker_get_address(*server_ucp_worker, &server_worker_addr, &server_worker_addr_len);
/* Create and submit RDMO client init task */
urom_rdmo_task_client_init(worker, cookie, 0, server_worker_addr, server_worker_addr_len,
urom_rdmo_client_init_finished);
/* Wait for completion */
do {
    ret = doca_pe_progress(pe);
    ucp_worker_progress(*server_ucp_worker);
} while (ret == 0 && res.result == DOCA_SUCCESS);
/* Check task result */
if (res.result != DOCA_SUCCESS)
    DOCA_LOG_ERR("Client init task finished with error");
```

**Initializing UROM Domain Context**

1. Create a domain context on the control plane PE.

2. Set domain context attributes.
3. Start the domain context by calling `doca_ctx_start`.

1. Exchange memory descriptors between all workers.

4. Wait until the domain context state is running.

```c
/* Create DOCA UROM domain instance */
doca_urom_domain_create(&domain);

/* Connect domain context to DOCA progress engine */
doca_pe_connect_ctx(pe, doca_urom_domain_as_ctx(domain));

/* Set domain attributes */
doca_urom_domain_set_oob(domain, oob);
doca_urom_domain_set_workers(domain, worker_ids, workers, nb_workers);
doca_urom_domain_set_buffers_count(domain, nb_buffers);
for each buffer:
    doca_urom_domain_add_buffer(domain);

/* Start domain context */
doca_ctx_start(doca_urom_domain_as_ctx(domain));

/* Loop till domain state changes to running */
do {
    doca_pe_progress(pe);
    result = doca_ctx_get_state(doca_urom_domain_as_ctx(domain), &state);
} while (state == DOCA_CTX_STATE_STARTING && result == DOCA_SUCCESS);

/* Request domain context stop */
doca_ctx_stop(doca_urom_domain_as_ctx(domain));

/* Destroy domain context */
```

**Destroying UROM Domain Context**

1. Request the domain context to stop by calling `doca_ctx_stop`.

2. Clean up resources by destroying the domain context.

```c
/* Request domain context stop */
doca_ctx_stop(doca_urom_domain_as_ctx(domain));

/* Destroy domain context */
```
Destroying UROM Worker Context

1. Request the worker context stop by calling `doca_ctx_stop` and posting the destroy command on the service context.

2. Wait until a completion for the destroy command is received.
   1. Change worker state to idle.

3. Clean up resources.

   ```c
   /* Stop worker context */
   doca_ctx_stop(doca_urom_worker_as_ctx(worker));

   /* Progress till receiving a completion */
   do {
       doca_pe_progress(pe);
       doca_ctx_get_state(doca_urom_worker_as_ctx(worker), &state);
   } while (state != DOCA_CTX_STATE_IDLE);

   /* Destroy worker context */
   doca_urom_worker_destroy(worker);
   ```

Destroying UROM Service Context

1. Wait for the completion of the UROM worker context commands.

2. Once all UROM workers have been successfully destroyed, initiate service context stop by invoking `doca_ctx_stop`.

3. Disconnect from the UROM service.

4. Perform resource cleanup.
Plugin Development

Developing Offload Plugin on DPU

1. Implement `struct urom_plugin_iface` methods.

   1. The `open()` method initializes the plugin connection state and may create an endpoint to perform communication with other processes/workers.

```c
static doca_error_t urom_worker_rdmo_open(struct urom_worker_ctx *ctx)
{
    ucp_context_h ucp_context;
    ucp_worker_h ucp_worker;
    struct urom_worker_rdmo *rdmo_worker;
    rdmo_worker = calloc(1, sizeof(*rdmo_worker));
    if (rdmo_worker == NULL)
        return DOCA_ERROR_NO_MEMORY;
    ctx->plugin_ctx = rdmo_worker;
    /* UCX transport layer initialization */
    .
    .
    .
```
2. The `addr()` method returns the address of the plugin endpoint generated during `open()` if it exists (e.g., UCX endpoint to communicate with other UROM workers).

3. The `worker_cmd()` method is used to parse and start work on incoming commands to the plugin.

```c
/* Create UCX worker Endpoint */
ucp_worker_create(ucp_context, &worker_params, &ucp_worker);
ucp_worker_get_address(ucp_worker, &rdmo_worker->ucp_data.worker_address,
&rdmo_worker->ucp_data.ucp_addrlen);

/* Resources initialization */
rdmo_worker->clients = kh_init(client);
rdmo_worker->eps = kh_init(ep);
/* Init completions list, UROM worker checks completed requests by calling progress() method */
ucp_list_head_init(&rdmo_worker->completed_reqs);

return DOCA_SUCCESS;
}

static doca_error_t urom_worker_rdmo_worker_cmd(struct urom_worker_ctx *ctx,
ucs_list_link_t *cmd_list)
{
    struct urom_worker_rdmo_cmd *rdmo_cmd;
    struct urom_worker_cmd_desc *cmd_desc;
    struct urom_worker_rdmo *rdmo_worker = (struct urom_worker_rdmo *)ctx->plugin_ctx;

    while (!ucs_list_is_empty(cmd_list)) {
        /* Get new RDMO command from the list */
        cmd_desc = ucs_list_extract_head(cmd_list, struct urom_worker_cmd_desc, entry);

        /* Unpack and deserialize RDMO command */
        urom_worker_rdmo_cmd_unpack(&cmd_desc->worker_cmd, cmd_desc->worker_cmd.len, &cmd);

        rdmo_cmd = (struct urom_worker_rdmo_cmd *)cmd->plugin_cmd;
        /* Handle command according to it's type */
        switch (rdmo_cmd->type) {
            case UROM_WORKER_CMD_RDMO_CLIENT_INIT:
```
4. The `progress()` method is used to give CPU time to the plugin code to advance asynchronous tasks.
5. The `notif_pack()` method is used to serialize notifications before they are sent back to the host.

2. Implement and expose the following symbols:

   1. `doca_error_t urom_plugin_get_version(uint64_t *version);`

      Returns a compile-time constant value stored within the `.so` file and is used to verify that the host and DPU plugin versions are compatible.

   2. `doca_error_t urom_plugin_get_iface(struct urom_plugin_iface *iface);`

      Get the `urom_plugin_iface` struct with methods implemented by the plugin.

3. Compile the user plugin as an `.so` file and place it where the UROM service can access it. For more details, refer to section "Plugin Discovery and Reporting" under the NVIDIA DOCA UROM Service Guide.

**Creating Plugin Host Task**

1. Allocate and init worker command task.

2. Populate payload buffer by task command.

3. Pack and serialize the command.

4. Set user data.

5. Submit the task.
struct doca_buf *payload;
struct doca_urom_worker_cmd_task *task;
struct doca_rdmo_task_data *task_data;
struct urom_worker_rdmo_cmd *rdmo_cmd;

/* Allocate task */
doca_urom_worker_cmd_task_allocate_init(worker_ctx, rdmo_id, &task);
/* Get payload buffer */
payload = doca_urom_worker_cmd_task_get_payload(task);
doca_buf_get_data(payload, (void **)&rdmo_cmd);
doca_buf_get_data_len(payload, &pack_len);
/* Populate commands attributes */
rdmo_cmd->type = UROM_WORKER_CMD_RDMO_CLIENT_INIT;
rdmo_cmd->client_init.id = id;
rdmo_cmd->client_init.addr = addr;
rdmo_cmd->client_init.addr_len = addr_len;
/* Pack and serialize the command */
urom_worker_rdmo_cmd_pack(rdmo_cmd, &pack_len, (void *)rdmo_cmd);
/* Update payload data size */
doca_buf_set_data(payload, rdmo_cmd, pack_len);
/* Set user data */
task_data = (struct doca_rdmo_task_data *)doca_urom_worker_cmd_task_get_user_data(task);
task_data->client_init_cb = cb;
task_data->cookie = cookie;
/* Set task plugin callback */
doca_urom_worker_cmd_task_set_cb(task, urom_rdmo_client_init_completed);

/* Submit task */
doca_task_submit(doca_urom_worker_cmd_task_as_task(task));

return DOCA_SUCCESS;
}

DOCA UROM Samples

This section provides DOCA UROM library sample implementations on top of BlueField.

The samples illustrate how to use the DOCA UROM API to do the following:
- Define and create a UROM plugin host and DPU versions for offloading HPC/AI tasks
- Build host applications that use the plugin to execute jobs on BlueField by the DOCA UROM service and workers

**Sample Prerequisite**

<table>
<thead>
<tr>
<th>Sample</th>
<th>Type</th>
<th>Prerequisite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sandbox</td>
<td>Plugin</td>
<td>A plugin which offloads the UCX tagged send/receive API</td>
</tr>
<tr>
<td>Graph</td>
<td>Plugin</td>
<td>The plugin uses UCX data structures and UCX endpoint</td>
</tr>
<tr>
<td>UROM Ping Pong</td>
<td>Program</td>
<td>The sample uses the Open MPI package as a launcher framework to launch two processes in parallel</td>
</tr>
</tbody>
</table>

**Info**

All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

**Running the Sample**

1. Refer to the following documents:
   - [NVIDIA DOCA Installation Guide for Linux](#) for details on how to install BlueField-related software
   - [NVIDIA DOCA Troubleshooting Guide](#) for any issue you may encounter with the installation, compilation, or execution of DOCA samples

2. To build a given sample:

   ```bash
   cd /opt/mellanox/doca/samples/doca_urom/<sample_name>
   meson /tmp/build
   ```
3. UROM Sample arguments:

<table>
<thead>
<tr>
<th>Sample</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UROM multi-workers bootstrap</td>
<td>-d, --device &lt;IB device name&gt;</td>
<td>IB device name</td>
</tr>
<tr>
<td>UROM Ping Pong</td>
<td>-d, --device &lt;IB device name&gt;</td>
<td>IB device name</td>
</tr>
<tr>
<td></td>
<td>-m, --message</td>
<td>Specify ping pong message</td>
</tr>
</tbody>
</table>

4. For additional information per sample, use the -h option:

```
/tmp/build/doca_<sample_name> -h
```

**UROM Plugin Samples**

DOCA UROM plugin samples have two components. The first one is the host component which is linked with UROM host programs. The second is the DPU component which is compiled as an .so file and is loaded at runtime by the DOCA UROM service (daemon, workers).

To build a given plugin:

```
cd /opt/mellanox/doca/samples/doca_urom/plugins/worker_<plugin_name>
```
Graph

This plugin provides a simple example for creating a UROM plugin interface. It exposes only a single command loopback, sending a specific value in the command, and expects to receive the same value in the notification from UROM worker.

References:

- /opt/mellanox/doca/samples/doca_urom/plugins/worker_graph/meson.build
- /opt/mellanox/doca/samples/doca_urom/plugins/worker_graph/worker_graph.h
- /opt/mellanox/doca/samples/doca_urom/plugins/worker_graph/worker_graph.c
- /opt/mellanox/doca/samples/doca_urom/plugins/worker_graph/worker_graph.h

Sandbox

This plugin provides a set of commands for using the offloaded ping pong communication operation.

References:

- /opt/mellanox/doca/samples/doca_urom/plugins/worker_sandbox/meson.build
- /opt/mellanox/doca/samples/doca_urom/plugins/worker_sandbox/urom_sandbox.h
- /opt/mellanox/doca/samples/doca_urom/plugins/worker_sandbox/worker_sandbox.c
- /opt/mellanox/doca/samples/doca_urom/plugins/worker_sandbox/worker_sandbox.c

Info

The binary worker_<sample_name>.so file is created under /tmp/build/.

meson /tmp/build
ninja -C /tmp/build
UROM Program Samples

DOCA UROM program samples can run only on the host side and require at least one DOCA UROM service instance to be running on BlueField.

The environment variable should be set `DOCA_UROM_SERVICE_FILE` to the path to the UROM service file.

UROM Multi-worker Bootstrap

This sample illustrates how to properly initialize DOCA UROM interfaces and use the API to spawn multiple workers on the same application process.

The sample initiates four threads as UROM workers to execute concurrently, alongside the main thread operating as a UROM service. It divides the workers into two groups based on their IDs, with odd-numbered workers in one group and even-numbered workers in the other.

Each worker executes the data loopback command by using the Graph plugin, sends a specific value, and expects to receive the same value in the notification.

The sample logic includes:

1. Opening DOCA IB device.
2. Initializing needed DOCA core structures.
3. Creating and starting UROM service context.
4. Initiating the Graph plugin host interface by attaching the generated plugin ID.
5. Launching 4 threads and for each of them:
   1. Creating and starting UROM worker context.
   2. Once the worker context switches to running, sending the loopback graph command to wait until receiving a notification.
3. Verifying the received data.

4. Waiting until an interrupt signal is received.

6. The main thread checking for pending jobs of spawning workers (4 jobs, one per thread).

7. Waiting until an interrupt signal is received.

8. The main thread checking for pending jobs of destroying workers (4 jobs, one per thread) for exiting.

9. Cleaning up and exiting.

References:

- /opt/mellanox/doca/samples/doca_urom/urom_multi_workers_bootstrap/urom_multi_workers_bootstrap
- /opt/mellanox/doca/samples/doca_urom/urom_common.c
- /opt/mellanox/doca/samples/doca_urom/urom_common.h

**UROM Ping Pong**

This sample illustrates how to properly initialize the DOCA UROM interfaces and use its API to create two different workers and run ping pong between them by using Sandbox plugin-based UCX.

The sample is using Open MPI to launch two different processes, one process as server and the second one as client, the flow is decided according to process rank.

The sample logic per process includes:

1. Initializing MPI.

2. Opening DOCA IB device.
3. Creating and starting UROM service context.

4. Initiating the Sandbox plugin host interface by attaching the generated plugin id.

5. Creating and starting UROM worker context.

6. Creating and starting domain context.

7. Through domain context, the sample processes exchange the worker's details to communicate between them on the BlueField side for ping pong flow.

8. Starting ping pong flow between the processes, each process offloading the commands to its worker on the BlueField side.

9. Verifying that ping pong is finished successfully.

10. Destroying the domain context.

11. Destroying the worker context.

12. Destroying the service context.

References:

- /opt/mellanox/doca/samples/doca_urom/urom_ping_pong/urom_ping_pong_sample.c
- /opt/mellanox/doca/samples/doca_urom/urom_ping_pong/urom_ping_pong_main.c
- /opt/mellanox/doca/samples/doca_urom/urom_ping_pong/meson.build
- /opt/mellanox/doca/samples/doca_urom/urom_common.c
- /opt/mellanox/doca/samples/doca_urom/urom_common.h

© Copyright 2024, NVIDIA. PDF Generated on 08/15/2024