DOCA DMA
Table of contents

Introduction
Prerequisites
Library Changes From Previous Releases
Changes in 2.8.0
API Additions
Environment
Architecture
Objects
Device and Device Representor
Memory Buffers
Configuration Phase
Configurations
Mandatory Configurations
Device Support
Buffer Support
Execution Phase
Tasks
Memory Copy Task
Events
State Machine
Idle
Starting
Running
Stopping

Alternative Datapath Options

DOCA DMA Samples

Running the Samples

Samples

DMA Local Copy

DMA Copy DPU

DMA Copy Host
This guide provides instructions on building and developing applications that require copying memory using Direct Memory Access (DMA).

**Introduction**

DOCA DMA provides an API to copy data between DOCA buffers using hardware acceleration, supporting both local and remote memory regions.

The library provides an API for executing DMA operations on DOCA buffers, where these buffers reside either in local memory (i.e., within the same host) or host memory accessible by the DPU. See **DOCA Core** for more information about the memory subsystem.

Using DOCA DMA, complex memory copy operations can be easily executed in an optimized, hardware-accelerated manner.

This document is intended for software developers wishing to accelerate their application's memory I/O operations and access memory that is not local to the host.

**Prerequisites**

This library follows the architecture of a DOCA Core Context, it is recommended read the following sections before:

- **DOCA Core Execution Model**
- **DOCA Core Device**
- **DOCA Core Memory Subsystem**

**Library Changes From Previous Releases**

**Changes in 2.8.0**

The following subsection(s) detail the `doca_comch` library updates in version 2.8.0.

**API Additions**

- `doxa_error_t doxa_dma_get_gpu_handle(struct doxa_dma *dma, struct doxa_gpu_dma **gpu_dma)`
- Provides the option to export DMA to GPU and use GPUNetIO for DMA datapath on the GPU

**Environment**

DOCA DMA-based applications can run either on the host machine or on the NVIDIA® BlueField® DPU target.

Copying from Host to DPU and vice versa only works with a DPU configured running in DPU mode as described in NVIDIA BlueField Modes of Operation.

**Architecture**

DOCA DMA is a DOCA Context as defined by DOCA Core. See DOCA Core Context for more information.

DOCA DMA leverages DOCA Core architecture to expose asynchronous tasks/events that are offloaded to hardware.

DMA can be used to copy data as follows:

- Copying from local memory to local memory:
• Using DPU to copy memory between host and DPU:

• Using host to copy memory between host and DPU:
Objects

Device and Device Representor

The DMA library needs a DOCA device to operate. The device is used to access memory and perform the actual copy. See DOCA Core Device Discovery.

For same BlueField DPU, it does not matter which device is used (PF/VF/SF), as all these devices utilize the same hardware component. If there are multiple DPUs, then it is possible to create a DMA instance per DPU, providing each instance with a device from a different DPU.

To access memory that is not local (from the host to the DPU or vice versa), the DPU side of the application must select a device with an appropriate representor. See DOCA Core Device Representor Discovery.

The device must stay valid for as long as the DMA instance is not destroyed.

Memory Buffers

The memory copy task requires two DOCA buffers containing the destination and the source. Depending on the allocation pattern of the buffers, refer to the table in the "Inventory Types" section. To find what kind of memory is supported, refer to the table in section "Buffer Support".

Buffers must not be modified or read during the memory copy operation.
Configuration Phase

To start using the library, users must go through a configuration phase as described in DOCA Core Context Configuration Phase.

This section describes how to configure and start the context, to allow execution of tasks and retrieval of events.

Configurations

The context can be configured to match the application use case.

To find if a configuration is supported, or what the min/max value for it is, refer to section "Device Support".

Mandatory Configurations

These configurations are mandatory and must be set by the application before attempting to start the context:

- At least one task/event type must be configured. See configuration of tasks and/or events in sections "Tasks" and "Events" respectively for information.
- A device with appropriate support must be provided upon creation

Device Support

DOCA DMA requires a device to operate. To picking a device, refer to "DOCA Core Device Discovery".

As device capabilities may change (see DOCA Core Device Support), it is recommended to select your device using the following method:

- `doca_dma_cap_task_memcpy_is_supported`
Some devices can allow different capabilities as follows:

- The maximum number of tasks
- The maximum buffer size

**Buffer Support**

Tasks support buffers with the following features:

<table>
<thead>
<tr>
<th>Buffer Type</th>
<th>Source Buffer</th>
<th>Destination Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local mmap buffer</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>mmap from PCIe export buffer</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>mmap From RDMA export buffer</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Linked list buffer</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**Execution Phase**

This section describes execution on CPU using DOCA Core Progress Engine.

**Tasks**

DOCA DMA exposes asynchronous tasks that leverage the DPU hardware according to the DOCA Core architecture. See DOCA Core Task.

**Memory Copy Task**

The memory copy task allows copying memory from one location to another. Using buffers as described in Buffer Support.

**Task Configuration**
<table>
<thead>
<tr>
<th>Description</th>
<th>API to set the configuration</th>
<th>API to query support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td>doca_dma_task_memcpy_set_conf</td>
<td>doca_dma_cap_task_memcpy_is_supported</td>
</tr>
<tr>
<td>Number of tasks</td>
<td>doca_dma_task_memcpy_set_conf</td>
<td>doca_dma_cap_get_max_num_tasks</td>
</tr>
<tr>
<td>Maximal buffer size</td>
<td>–</td>
<td>doca_dma_cap_task_memcpy_get_max_buf_size</td>
</tr>
<tr>
<td>Maximum buffer list size</td>
<td>–</td>
<td>doca_dma_cap_task_memcpy_get_max_buf_list_len</td>
</tr>
</tbody>
</table>

**Task Input**

Common input as described in [DOCA Core Task](#).

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source buffer</td>
<td>Buffer that points to the memory to be copied</td>
<td>Only the data residing in the data segment is copied</td>
</tr>
<tr>
<td>Destination buffer</td>
<td>Buffer that points to where memory is copied</td>
<td>The data is copied to the tail segment extending the data segment</td>
</tr>
</tbody>
</table>

**Task Output**

Common output as described in [DOCA Core Task](#).

**Task Completion Success**

After the task is completed successfully:

- The data is copied from source to destination
- The destination buffer data segment is extended to include the copied data
Task Completion Failure

If the task fails midway:

- The context may enter stopping state, if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task has been submitted, then the source and destination should not be read/written to
- Source and destination must not overlap
- Other limitations are described in `DOCA Core Task`

Events

DOCA DMA exposes asynchronous events to notify on changes that happen unexpectedly, according to `DOCA Core architecture`.

The only event DMA exposes is common events as described in `DOCA Core Event`.

State Machine
The DOCA DMA library follows the Context state machine as described in DOCA Core Context State Machine.

The following section describes how to move states and what is allowed in each state.

**Idle**

In this state it is expected that application:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to section "Configurations"
- Starting the context

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Create the context</td>
</tr>
<tr>
<td>Running</td>
<td>Call stop after making sure all tasks have been freed</td>
</tr>
<tr>
<td>Stopping</td>
<td>Call progress until all tasks are completed and freed</td>
</tr>
</tbody>
</table>

**Starting**

This state cannot be reached.

**Running**

In this state it is expected that application:

- Allocates and submits tasks
- Calls progress to complete tasks and/or receive events

Allowed operations:

- Allocating a previously configured task
- Submitting a task
- Calling stop

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>Call start after configuration</td>
</tr>
</tbody>
</table>

### Stopping

In this state it is expected that application:

- Calls progress to complete all inflight tasks (tasks complete with failure)
- Frees any completed tasks

Allowed operations:

- Call progress

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running</td>
<td>Call progress and fatal error occurs</td>
</tr>
<tr>
<td>Running</td>
<td>Call stop without freeing all tasks</td>
</tr>
</tbody>
</table>

### Alternative Datapath Options
DOCA DMA allows data path to be run on the CPU or GPU.

### Info

For the CPU data path, see Execution Phase.

---

## GPU Datapath

DOCA offers the DOCA GPUNetIO library which provides a programming model for offloading the orchestration of the communication to a GPU CUDA kernel.

The user may run a DMA operation on the GPU data path by configuring the DOCA DMA context used by the application in the following manner:

1. Obtain DOCA CTX by calling `doca_dma_as_ctx()`.

2. Set the datapath of the context to GPU by calling `doca_ctx_set_datapath_on_gpu()`. For additional information, refer to DOCA Core Alternative Data Path.

3. Finish context configuration and start the context by calling `doca_ctx_start()`. For additional information, refer to DOCA Core Context.

After configuring the datapath, the user can obtain a GPU handle for the DOCA RDMA context by calling `doca_dma_get_gpu_handle()`. The GPU handle must be passed to a GPU CUDA kernel so the DOCA GPUNetIO CUDA device functions can execute datapath operations. For additional information, refer to section "GPU Functions – RDMA" under DOCA GPUNetIO library documentation.

---

## DOCA DMA Samples

This section describes DOCA DMA samples based on the DOCA DMA library.

The samples illustrate how to use the DOCA DMA API to do the following:

- Copy contents of a local buffer to another buffer
Use DPU to copy contents of buffer on the host to a local buffer

Info

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

Running the Samples

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:

   ```
   cd /opt/mellanox/doca/samples/doca_dma/dma_local_copy
   meson /tmp/build
   ninja -C /tmp/build
   ```

   The binary `doca_dma_local_copy` is created under `/tmp/build/`.

3. Sample (e.g., `doca_dma_local_copy`) usage:

   ```
   Usage: doca_<sample_name> [DOCA Flags] [Program Flags]
   
   DOCA Flags:
   -h, --help Print a help synopsis
   -v, --version Print program version information
   -l, --log-level Set the (numeric) log level for the program
   <10=DISABLE, 20=Critical, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
   -j, --json <path> Parse all command flags from an input json file
   ```
4. For additional information per sample, use the \texttt{-h} option:

\begin{verbatim}
/tmp/build/<sample_name> -h
\end{verbatim}

**Samples**

**DMA Local Copy**

This sample illustrates how to locally copy memory with DMA from one buffer to another on the DPU. This sample should be run on the DPU.

The sample logic includes:

1. Locating DOCA device.
2. Initializing needed DOCA core structures.
3. Populating DOCA memory map with two relevant buffers.
4. Allocating element in DOCA buffer inventory for each buffer.
5. Initializing DOCA DMA memory copy task object.
6. Submitting DMA task.
7. Handling task completion once it is done.
8. Checking task result.
9. Destroying all DMA and DOCA core structures.

Reference:
This sample illustrates how to copy memory (which contains user defined text) with DMA from the x86 host into the DPU. This sample should be run on the DPU.

The sample logic includes:

1. Locating DOCA device.
2. Initializing needed DOCA core structures.
3. Reading configuration files and saving their content into local buffers.
4. Allocating the local destination buffer in which the host text is to be saved.
5. Populating DOCA memory map with destination buffer.
6. Creating the remote memory map with the export descriptor file.
7. Creating memory map to the remote buffer.
8. Allocating element in DOCA buffer inventory for each buffer.
9. Initializing DOCA DMA memory copy task object.

Note

This sample should run only after DMA Copy Host is run and the required configuration files (descriptor and buffer) have been copied to the DPU.
10. Submitting DMA task.

11. Handling task completion once it is done.

12. Checking DMA task result.

13. If the DMA task ends successfully, printing the text that has been copied to log.

14. Printing to log that the host-side sample can be closed.

15. Destroying all DMA and DOCA core structures.

Reference:

- /opt/mellanox/doca/samples/doca_dma/dma_copy_dpu/dma_copy_dpu_sample.c
- /opt/mellanox/doca/samples/doca_dma/dma_copy_dpu/dma_copy_dpu_main.c
- /opt/mellanox/doca/samples/doca_dma/dma_copy_dpu/meson.build

**DMA Copy Host**

⚠️ **Note**

This sample should be run first. It is user responsibility to transfer the two configuration files (descriptor and buffer) to the DPU and provide their path to the DMA Copy DPU sample.

This sample illustrates how to allow memory copy with DMA from the x86 host into the DPU. This sample should be run on the host.

The sample logic includes:

1. Locating DOCA device.

2. Initializing needed DOCA core structures.
3. Populating DOCA memory map with source buffer.

4. Exporting memory map.

5. Saving export descriptor and local DMA buffer information into files. These files should be transferred to the DPU before running the DPU sample.

6. Waiting until DPU DMA sample has finished.

7. Destroying all DMA and DOCA core structures.

Reference:

- /opt/mellanox/doca/samples/doca_dma/dma_copy_host/dma_copy_host_sample.c
- /opt/mellanox/doca/samples/doca_dma/dma_copy_host/dma_copy_host_main.c
- /opt/mellanox/doca/samples/doca_dma/dma_copy_host/meson.build