NVIDIA DOCA DMA

Programming Guide

MLNX-15-060544 _v1.4 | September 2022

Table of Contents

Chapter 1. Introduction... 1
Chapter 2. Prerequisites... 2
Chapter 3. Architecture...3
Chapter 4. API...4
Chapter 5. Local Memory Programming Guide..5
5.1. Initialization Process...5
5.1.1. DOCA Device Open.. 5
5.1.2. Creating DOCA Core Objects...5
5.1.3. Initializing DOCA Core Objects.. 5
5.1.3.1. Memory Map Initialization...6
5.1.3.2. Buffer Inventory... 6
5.1.3.3. DMA Context Initialization... 6
5.1.4. Populating Memory Map...6
5.1.5. Constructing DOCA Buffers...6
5.2. DMA Execution.. 6
5.2.1. Constructing and Executing DOCA DMA Operation...6
5.2.2. Waiting for Completion.. 7
5.2.3. Clean Up.. 7
Chapter 6. Remote Memory Programming Guide..8
6.1. Sender...8
6.2. Receiver... 8

Chapter 1. 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 in either local memory (i.e., within the same host) or remote memory (i.e., on another host).

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.

Chapter 2. Prerequisites

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

Chapter 3. Architecture

DOCA DMA relies heavily on the underlying DOCA core architecture for its operation, utilizing the existing memory map and buffer objects.

After initialization, a DMA operation is requested by submitting a DMA job on the relevant work queue. The DMA library then executes that operation asynchronously before posting a completion event on the work queue.

Chapter 4. API

This chapter details the specific structures and operations related to the DOCA DMA library for general initialization, setup, and clean-up. Please see later sections on local and remote memory DMA operations.

The API for DOCA DMA consists of the main DMA job structure that is passed to the work queue to instruct the DMA library on source, destination, and operation length.

As per most memory copy operations, the source and destination buffers should not overlap while the `num_bytes_to_copy` field defines the number of bytes to copy from the start of the source buffer to the destination buffer.

```
struct doca_dma_job_memcpy {
 struct doca_job base;               /**< Common job data */
 struct doca_buf *dst_buff;          /**< Destination data buffer */
 struct doca_buf const *src_buff;    /**< Source data buffer */
 uint64_t num_bytes_to_copy;         /**< Number of bytes to copy */
};
```

As with other libraries, the DMA job contains the standard doca_job base field that should be set as follows:

```
/* Construct DMA job */
doca_job.type = DOCA_DMA_JOB_MEMCPY;
doca_job.flags = DOCA_JOB_FLAGS_NONE;
doca_job.ctx = doca_dma_as_ctx(doca_dma_inst);
```

The DMA job-specific fields should be set based on the required source and destination buffers. They must provide the number of bytes you wish to copy.

```
dma_job.base = doca_job;
dma_job.dst_buff = dst_doca_buf;
dma_job.src_buff = src_doca_buf;
dma_job.num_bytes_to_copy = data_to_copy_len;
```

As with all WorkQ operations, the application must periodically poll the work queue (via `doca_workq_progress_retrieve` API call). When the retrieve call returns with a `DOCA_SUCCESS` value (to indicate the work queues event is valid), you can then test that received event for success:

```
event.result.u64 == DOCA_SUCCESS
```

Chapter 5. Local Memory Programming Guide

These sections discuss the usage of the DOCA DMA library in real-world situations. Most of this section utilizes code which is available through the DOCA DMA sample projects located under /samples/doca_dma/.

When memory is local to your DOCA application (i.e., you can directly access the memory space of both source and destination buffers) this is referred to as a local DMA operation.

The following step-by-step guide goes through the various stages required to initialize, execute, and clean-up a local memory DMA operation.

5.1. Initialization Process

The DMA API uses the DOCA core library to create the required objects (memory map, inventory, buffers, etc.) for the DMA operations. This section runs through this process in a logical order. If you already have some of these operations in your DOCA application, you may skip or modify them as needed.

5.1.1. DOCA Device Open

The first requirement is to open a DOCA device, normally your BlueField controller. You should iterate all DOCA devices (via doca_devinfo_list_create) and select one using some criteria (PCIe address, etc.). After this, the device should be opened using doca_dev_open.

5.1.2. Creating DOCA Core Objects

DOCA DMA requires several DOCA objects to be created. This includes the memory map (doca_mmap_create), buffer inventory (doca_buf_inventory_create), work queue (doca_workq_create). DOCA DMA also requires the actual DOCA DMA context to be created (doca_dma_create).

5.1.3. Initializing DOCA Core Objects

In this phase of initialization, the core objects are ready to be set up and started.

5.1.3.1. Memory Map Initialization

Prior to starting the mmap (`doca_mmap_start`), make sure that you set the maximum chunks correctly (via `doca_mmap_property_set`). After starting mmap, add the DOCA device to the mmap (`doca_mmap_dev_add`).

5.1.3.2. Buffer Inventory

This can be started using the `doca_buf_inventory_start call`.

5.1.3.3. DMA Context Initialization

Finally, the context created previously can have the device added (`doca_ctx_dev_add`), started (`doca_ctx_start`), and work queue added (`doca_ctx_workq_add`).

5.1.4. Populating Memory Map

Register the memory regions you require for DMA operations with the memory map using the `doca_mmap_populate` call. These regions may be one large region, or many smaller regions depending on your use case.

5.1.5. Constructing DOCA Buffers

Prior to building and submitting a DOCA DMA operation, you must construct two DOCA buffers for the source and destination addresses (the addresses used must exist within the memory region registered with the memory map). The `doca_buf_inventory_buf_by_addr` returns a `doca_buffer` when provided with a memory address.

These are the buffers supplied to the DMA operation and both must contain at least `num_bytes_to_copy`. If they are bigger, then any bytes beyond the range [0, `num_bytes_to_copy`) remain unmodified.

5.2. DMA Execution

The DMA operation is asynchronous in nature. Therefore, you must enqueue the operation and then, later, poll for completion.

5.2.1. Constructing and Executing DOCA DMA Operation

To begin the DMA operation, you must enqueue a DMA job on the previously created work queue object. This involves creating the DMA job (struct `doca_dma_job_memcpy`) that is a composite of specific DMA fields.

Within the DMA job structure, the `type` field should be set to `DOCA_DMA_JOB_MEMCPY` with the context field pointing to your DMA context.

The DMA specific elements of the job point to your DOCA buffers for source and destination, with a length field providing the number of bytes to be copied.

Finally, the `doca_workq_submit` API call is used to submit the DMA operation to the hardware.

5.2.2. Waiting for Completion

To detect when the DMA operation has completed, you should periodically poll the work queue (via `doca_workq_progress_retrieve`).

When the API call indicates that a valid event has been received, you should then detect the success of the DMA operation through the `event.result.u64` field equal to `DOCA_SUCCESS`. It should be noted that other work queue operations (i.e., non-DMA operations) present their events differently. Refer to their respective guides for more information.

To clean up the `doca_buffers`, you should deference them using the `doca_buf_refcount_rm` call. This call should be made on all buffers when you have finished with them (regardless of whether the operation is successful or not).

5.2.3. Clean Up

The main cleanup process is to remove the worker queue from the context (`doca_ctx_workq_rm`), stop the context itself (`doca_ctx_stop`), remove the device from the context (`doca_ctx_dev_rm`), and remove the device from the memory map (`doca_mmap_dev_rm`).

The final destruction of the objects can now occur. This can occur in any order, but destruction must occur on the work queue (`doca_workq_destroy`), dma context (`doca_dma_destroy`), buf inventory (`doca_buf_inventory_destroy`), mmap (`doca_mmap_destroy`), and device closure (`doca_dev_close`).

Chapter 6. Remote Memory Programming Guide

These sections discuss the creation of a remote memory DMA operation. This operation allows memory from a remote host, accessible by DOCA DMA, to be used as a source or destination.

There are two sample applications that show you how this operation may work in scanning a remote memories location for a particular piece of data. They are located at `/samples/doca_dma` as `dma_remote_copy_receiver` and `dma_remote_copy_sender`.

6.1. Sender

The sender holds the source memory is copied to the remote receiver. The method of how the source memory address is transmitted to the remote receiver is for the developer to decide. In the sample application, a socket is connected from a "host" sender to a "remote" BlueField DPU. The address passed via this method.

The sender application should open the device, as per a normal local memory operation, but initialize only a memory map (`doca_mmap_create`, `doca_mmap_start`, `doca_mmap_dev_add`).

It should then populate the mmap with the source buffer (`doca_mmap_populate`) and call a special mmap function (`doca_mmap_export`). This function generates a JSON structure that can be transmitted to the remote device. The information in the JSON structure refers to the exported "remote" memory (from the perspective of the receiver).

6.2. Receiver

For reception, the standard initiation described for the local memory process should be followed.

Prior to constructing the source DOCA buffer (via `doca_buf_inventory_buf_by_addr`), you should call the special mmap function that retrieves the remote mmap (`doca_mmap_create_from_export`).

The source DOCA buffer can then be created using this remote memory map.

All other aspects of the application (executing, waiting on results, and cleanup) should be the same as the process described for local memory operations.

Notice

Trademarks

Copyright

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