NVIDIA DOCA DMA

Programming Guide

MLNX-15-060544 _v2.0.2 | May 2023

Table of Contents

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, if running on the DPU, remote memory (i.e., host memory). See NVIDIA DOCA Core Programming Guide for more information about the memory sub-system.

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. See the NVIDIA DOCA Core Programming Guide for more information.

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.

The DMA library is a DOCA Core context where it is possible to add multiple WorkQs to the same context. This allows the DMA context to be used in a familiar way by utilizing WorkQs to orchestrate the DMA engine's workload.

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.

4.1. DMA Memory Copy Job

4.1.1. Job and Result Structures

The following is the structure of the DMA memory copy job structure submitted to the work queue to initiate a DMA memory copy operation.

```
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 */
};
```

Where `doca_job` is defined as follows:

```
struct doca_job {
    int type;                       /**< Must hold DOCA_DMA_JOB_MEMCPY */
    int flags;                      /**< At time of writing can only be
 DOCA_JOB_FLAGS_NONE */
    struct doca_ctx *ctx;           /**< Must hold the DMA context - acquired from
 doca_dma_as_ctx */
    union doca_data user_data;  /**< Can hold a user defined value, in order to
 correlate between submitted job and matching completion event.*/
};
```

In addition to the job structure, DMA also defines a result structure which is only useful if a job failure has occurred (indicated by `doca_workq_progress_retrieve() == DOCA_ERROR_IO_FAILED`). In that case, the result can be retrieved from the `doca_event` by casting the result field (`struct doca_dma_memcpy_result *memcpy_result = &doca_event.result.u64`).

```
struct doca_dma_memcpy_result {
    doca_error_t result; /**< Operation result */
};
```

4.1.2. Buffer Support

As per most memory copy operations, the source and destination buffers should not overlap.

The length of the copy is determined by the source buffer data length, and the offset to copy from/to is decided for each buffer by the data pointer. See `doca_buf_set_data()` for more information

The following constraints can be queried during runtime:

- Maximum supported buffer data length in bytes – in case of a linked list, represents maximum data length of each individual buffer, `doca_dma_get_max_buf_size()`.
- Maximum number of allowed elements in a buffer (source buffer only) linked list, `doca_dma_get_max_list_buf_num_elem()`.

4.1.2.1. Source Buffer

The source buffer supports any provided `doca_buf` this can be a local buffer, a remote buffer; or a linked list of buffers. See the NVIDIA DOCA Core Programming Guide for more information.

The buffer data can be set using the `doca_buf_set_data()` API. This sets the data pointer, and the data length. If a linked list is provided, then this API should be called on each buffer individually either before or after the buffers have been chained.

The number of bytes to be copied is determined by the data length of the source buffer. If a linked list is provided, then the size would be the sum of data length of each buffer.

The segment to copy from is determined by the data pointer of the buffer. If a linked list is provided, then data pointer of each buffer is taken into consideration.

Source buffer must not be freed, nor the data invalidated before the job is finished, and the completion event retrieved. This buffer is never modified by the job.

4.1.2.2. Destination Buffer

The destination buffer can only be a local or remote `doca_buf`. If a `doca_buf` with the linked list extension is provided, it may be supported only if the liked list has a size of 1. Otherwise undefined behavior would occur.

As with the source buffer, the data can be set using the `doca_buf_set_data()` API. This sets the data pointer and the data length.

Since the number of bytes to be copied is determined by the source buffer, setting the data length can be skipped.

The data can be copied to a subsegment of the buffer, this is determined by the buffer's data pointer.

The destination buffer must not be freed and the data is considered undefined until the job is finished and the completion event retrieved.

Once a successful completion event is received, the destination buffer would contain an exact copy of the source buffer content. The destination buffer would also have its data length set which can be inspected using `doca_buf_get_data_len()`.

4.1.3. Device Support

To start DMA jobs, a device must be added to the DMA context, and to the mmap where the memory is defined. See section "DOCA Device" in the NVIDIA DOCA Core Programming Guide.

To verify whether a device is capable of executing the desired jobs, the following can be queried during runtime:

- `doca_dma_job_get_supported()` – DMA job support
- `doca_dma_get_max_buf_size()` – maximum supported buffer data length in bytes. If a linked list is used, then it would represent the maximum data length of each individual buffer.
- `doca_dma_get_max_list_buf_num_elem()` – maximum number of allowed elements in a buffer linked list (source buffer only)
- `doca_devinfo_get_is_mmap[_from]_export_supported()` – if a remote memory is used, then these checks can be made. Refer to the NVIDIA DOCA Core Programming Guide for more.

4.1.4. Context Configurations

DMA context does not hold any additional configurations other than the ones described in the the NVIDIA DOCA Core Programming Guide.

4.1.5. WorkQ Support

DOCA DMA conforms to DOCA Core's execution model in that jobs can be asynchronously run using a WorkQ until a completion event is retrieved. More information on the execution model can be found in the NVIDIA DOCA Core Programming Guide.

DOCA DMA in particular supports adding multiple WorkQs to the same DMA context. This can be useful in multithreading cases where an application can add multiple WorkQs (via `doca_ctx_workq_add()`) to the same DMA context, allowing each thread to use a different WorkQ, since WorkQ is not thread-safe.

DMA operations can be retrieved utilizing the WorkQ event-driven mode. The following check `doca_ctx_get_event_driven_supported()` can be used to verify that.

4.2. Completion Event Retrieval

If the polling of the WorkQ (`doca_workq_progress_retrieve()` API call) finishes with `DOCA_SUCCESS`, then a job has been complete since the WorkQ supports sending all types of jobs from all libraries. To identify the type of response held by a `doca_event`, the user can compare the value of `doca_event.type` with the job type they submitted, in this case `DOCA_DMA_JOB_MEMCPY`, to allow them to handle responses appropriately. And the `doca_event.user_data` field can be utilized to correlate an event with an originating job if multiple jobs of the same type have been submitted to the WorkQ.

If the polling of the WorkQ (`doca_workq_progress_retrieve()` API call) fails with `DOCA_ERROR_IO_FAILED`, then this means that the job has failed midway, and a retrieved event can be inspected for more information about the failure:

```
if (event.type == DOCA_DMA_JOB_MEMCPY) {
    struct doca_dma_memcpy_result *memcpy_result = (struct doca_dma_memcpy_result
 *)&event.result.u64;
    DOCA_LOG_ERR("DMA Job failed. user_data: %lu, DMA error: %s",
 event.user_data.u64, doca_get_error_name(memcpy_result->result));
}
```

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/dma_local_copy`.

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()`), select one using some criteria (e.g., PCIe address), then the device should be opened (via `doca_dev_open()`). More information that may help decide on a device can be found in the Device Support section. Once the desired device is opened, the list can be immediately destroyed (via `doca_devinfo_list_destroy()`). This frees the resources of all devices other than the one that was opened.

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()`).

Once a DMA instance is created, it can be used as a context (using `doca_ctx` APIs). This can be achieved by getting a context representation using `doca_dma_as_ctx()`.

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

The memory map is used to define the memory regions (chunks) where data is copied. It is possible to create a single memory map and use it to define all memory regions in this case. See NVIDIA DOCA Core Programming Guide for more details about memory subsystem.

Prior to starting the mmap (`doca_mmap_start()`), make sure that you set the maximum chunks correctly (via `doca_mmap_set_max_num_chunk()`). After starting mmap, add the DOCA device to the mmap (`doca_mmap_dev_add()`). This ensures that device can access all the memory populated within the mmap. After the device has been added, the application should proceed to provide the memory regions to use for DMA operations (via `doca_mmap_populate()`). These regions may be one large region, or many smaller regions.

5.1.3.2. Buffer Inventory

The buffer inventory is used to pre-allocate `doca_buf` descriptors to avoid doing allocations in the middle of application. In addition, the inventory can define extensions for the used buffers (see NVIDIA DOCA Core Programming Guide). DMA operations support buffers with any extension.

It is important to make sure that the inventory can hold as many buffers as the application requires. For instance, if the application runs up to 10 jobs at the same time, then the inventory should hold 20 buffers since every job holds exactly 2 buffers. And the buffers cannot be released until the job is complete.

The inventory can be started using the `doca_buf_inventory_start()` call.

5.1.3.3. DMA Context Initialization

The context created (via `doca_dma_create()`) and acquired using (`doca_dma_as_ctx()`) can have the device added (`doca_ctx_dev_add()`), started (`doca_ctx_start()`), and work queue added (`doca_ctx_workq_add()`). It is also possible to add multiple WorkQs to the same context as well.

5.1.4. 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 any of the memory regions populated in the memory map). The `doca_buf_inventory_buf_by_data()` returns a `doca_buffer` with the data pointer and data length. Alternatively, it is possible to first allocate the buffer `doca_buf_inventory_buf_by_addr()` and then include only a segment within the buffer to be used in the DMA operation by using the `doca_buf_set_data()` API to set the data pointer and length.

These are the buffers supplied to the DMA operation the source buffer is used to determine the length to copy, where the destination buffer must be long enough to hold the data.

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.

Finally, the `doca_workq_submit()` API call is used to submit the DMA operation to the hardware. Some errors may be detected immediately after submitting the job while others are only discovered midway through the job. For such cases, please refer to Completion Event Retrieval.

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()`).

If the call returns a valid event, the `doca_event` type field should be tested before inspecting the result as other WorkQ 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 dereference them using the `doca_buf_refcount_rm()` call. This call should be made on both buffers when you are done with them (regardless of whether the operation is successful or not). If the source buffer is a linked list, then it is enough to only dereference the head. That effectively releases the entire list.

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. For more information about the memory sub-system, refer to the NVIDIA DOCA Core Programming Guide.

There are two sample applications that show how this operation may work in scanning a remote memory's location for a particular piece of data:

- `/samples/doca_dma/dma_copy_dpu`
- `/samples/doca_dma/dma_copy_host`

Please note that copying memory from Host to DPU, or DPU to Host, or even Host to Host, is always done from the DPU. From Host it is only possible to copy local memory.

6.1. Sender

The sender holds the source memory to be 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 DPU. The address is passed via this method.

The sender application should open the device as per a normal local memory operation but only with a memory map initialized (`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 blob that can be transmitted to the remote device. The blob can be used by the receiver to access memory in the sender's mmap.

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.

Chapter 7. DOCA DMA Samples

This guide provides DMA samples implementation on top of the BlueField DPU.

Using DOCA DMA, you can easily execute complex memory copy operations in an optimized, hardware-accelerated way:

- The `dma_local_copy` sample copies content between two local buffers on the DPU.
- The `dma_copy_dpu`/`dma_copy_host` copies user-defined text from the host to the DPU.

Note: DMA Copy Host must be run before DMA Copy DPU.

7.1. 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:

```
cd /opt/mellanox/doca/samples/doca_dma/<sample_name>
meson build
ninja -C build
```

   Note: The binary `doca_<sample_name>` will be created under `./build/`.

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

Sample	Argument	Description
DMA Local Copy	`-p, --pci-addr`	DOCA DMA device PCIe address
	`-t, --text`	Text to DMA copy from one local buffer to another
DMA Copy Host	`-p, --pci-addr`	DOCA DMA device PCIe address

Sample	Argument	Description
	`-t, --text`	Text to DMA copy from the host to the DPU
	`-d, --descriptor-path`	Path on which the exported descriptor file is saved
	`-b, --buffer-path`	Path on which the buffer information file is saved
DMA Copy DPU	`-p, --pci-addr`	DOCA DMA device PCIe address
	`-d, --descriptor-path`	Path from which the exported descriptor file is read
	`-b, --buffer-path`	Path from which the buffer information file is read

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

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

7.2. Samples

7.2.1. 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 job object.
6. Submitting DMA job into work queue.
7. Retrieving DMA job from the queue once it is done.
8. Checking job result.
9. Destroying all DMA and DOCA core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_dma/dma_local_copy/dma_local_copy_sample.c`
- `/opt/mellanox/doca/samples/doca_dma/dma_local_copy/dma_local_copy_main.c`
- `/opt/mellanox/doca/samples/doca_dma/dma_local_copy/dma_local_copy_main.c`

7.2.2. DMA Copy DPU

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.

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 will 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 job object.
10. Submitting DMA job into work queue.
11. Retrieving DMA job from the queue once it is done.
12. Checking DMA job result.
13. If the DMA job 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`

7.2.3. 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`