



NVIDIA DOCA DMA

Programming Guide

Table of Contents

Chapter 1. Introduction.....	1
Chapter 2. Prerequisites.....	2
Chapter 3. Architecture.....	3
Chapter 4. API.....	4
4.1. DMA Memory Copy Job.....	4
4.1.1. Job and Result Structures.....	4
4.1.2. Buffer Support.....	4
4.1.2.1. Source Buffer.....	5
4.1.2.2. Destination Buffer.....	5
4.1.3. Device Support.....	6
4.1.4. Context Configurations.....	6
4.1.5. WorkQ Support.....	6
4.2. Completion Event Retrieval.....	6
Chapter 5. Programming Local Memory.....	8
5.1. Initialization Process.....	8
5.1.1. DOCA Device Open.....	8
5.1.2. Creating DOCA Core Objects.....	8
5.1.3. Initializing DOCA Core Objects.....	9
5.1.3.1. Memory Map Initialization.....	9
5.1.3.2. Buffer Inventory Initialization.....	9
5.1.3.3. DMA Context and WorkQ Initialization.....	9
5.1.4. Constructing DOCA Buffers.....	10
5.2. DMA Memory Copy Job Execution.....	10
5.2.1. Constructing and Executing DOCA DMA Operation.....	10
5.2.2. Considerations.....	11
5.2.3. Waiting for Completion.....	11
5.2.4. Clean Up.....	11
Chapter 6. Programming Remote Memory.....	12
6.1. Host.....	12
6.2. DPU.....	13
Chapter 7. DOCA DMA Samples.....	15
7.1. Running the Sample.....	15
7.2. Samples.....	16
7.2.1. DMA Local Copy.....	16

7.2.2. DMA Copy DPU.....	17
7.2.3. DMA Copy Host.....	17

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 host memory accessible by the DPU. 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()`.



Note: Once a job is submitted, the ownership of the buffers is moved to the library, and they should not be modified until the job is complete. Modifying the buffer or the data of the buffer can cause anomalous behavior.

4.1.2.1. Source Buffer

- ▶ Source buffer cannot be an RDMA buffer
- ▶ Source buffer can be local or DPU-accessible memory on host
- ▶ Source buffer can be a linked list
- ▶ The segment to copy from is determined by the data pointer of the buffer. If a linked list is provided, then the data pointer of each buffer is taken into consideration.
- ▶ 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 data length sum of each buffer.
- ▶ Source buffer must not be freed, nor the data invalidated, before the job is finished, and the completion event retrieved.
- ▶ Source buffer is never modified by the job

4.1.2.2. Destination Buffer

- ▶ Destination buffer cannot be an RDMA buffer
- ▶ Destination buffer cannot be a linked list of buffers



Note: If a linked list is provided, anomalous behavior occurs.

- ▶ Destination buffer can be local or DPU-accessible memory on host
- ▶ Once a successful completion event is received, the destination buffer would contain an exact copy of the source buffer content, appended to the end of the data section, as it was prior to the job submission
- ▶ The destination buffer would also have its data length increased by the amount of bytes copied. The updated data length can be inspected using `doca_buf_get_data_len()`.
- ▶ The destination buffer must not be freed and the data is considered undefined until the job is finished and the completion event is retrieved

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_dpu_supported()` – if accessing memory on the host from the DPU, 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 [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. Programming Local Memory

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 region where data is copied to or from. See [NVIDIA DOCA Core Programming Guide](#) for more details about memory subsystem.

Consider the case where the source data and destination data reside in two different memory ranges which are not necessarily continuous: For that purpose, two `doca_mmaps` must be created, a source mmap and a destination mmap.

The initialization of both mmaps is similar:

1. Set the source or destination memory range using `doca_mmap_set_memrange()`.
2. Add the `doca_device` that has been opened earlier (must be same device used for the DMA context initialization later) using `doca_mmap_dev_add()`.
3. Set permissions of the mmap. In this case, set the minimum viable permissions as follows:
 - ▶ The source mmap is only be used for reading data (`DOCA_ACCESS_LOCAL_READ_ONLY`)
 - ▶ The destination mmap is used for writing data (`DOCA_ACCESS_LOCAL_READ_WRITE`)
4. Start the mmap using `doca_mmap_start()`. Once this is done, the mmap cannot be configured any further.

5.1.3.2. Buffer Inventory Initialization

The inventory is used to allocate two `doca_bufs`; one for the source and another for the destination. Unlike with [mmap](#), it is enough to allocate a single inventory to hold them both.

The initialization of buffer inventory:

1. Specify that buffer inventory must accommodate two buffers during creation using `doca_buf_inventory_create()`.
2. Start the inventory using `doca_buf_inventory_start()`.

5.1.3.3. DMA Context and WorkQ Initialization

The DMA context must be created and prepared to start receiving jobs:

1. Create the DMA context (`doca_dma_create()`).
2. (Optional) Verify that DMA is supported (`doca_dma_job_get_supported()`).
3. Get a `doca_ctx` representation of the DMA context (`doca_dma_as_ctx()`).

4. Add a device to the context (`doca_ctx_dev_add()`). Must be the same device added to the mmap.
5. Start the context (`doca_ctx_start()`). After this step, the context can no longer be configured.
6. Add a WorkQ to the context (`doca_ctx_workq_add()`). This allows submission of DMA jobs to that WorkQ.

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.

At this stage, there are two initialized mmaps and an inventory. It is now possible to use `doca_buf_inventory_buf_by_data()` to allocate the source and destination buffers for the job. Considerations for using this API:

- ▶ The inventory holds the `doca_buf` descriptors, so that no memory is allocated in this stage
- ▶ The mmap holds necessary information for mapping memory to device. The caller must provide the matching source/destination mmap.
- ▶ As a result, `doca_buf_inventory_buf_by_data()` is considered non-resource intensive and can be done in data path

5.2. DMA Memory Copy Job Execution

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_memcopy`) that is a composite of specific DMA fields.

Within the DMA job structure, the `type` field should be set to `DOCA_DMA_JOB_MEMCOPY` 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. Considerations

- ▶ The DMA operation is asynchronous in nature. Therefore, you must enqueue the operation and then, later, poll for completion.
- ▶ The DMA operation is not atomic. Therefore, it is imperative for the application to handle synchronization appropriately.



Note: A DMA operation is not atomic because it is possible for the host side to read a memory which is accessed in parallel by the DPU. Therefore, the application must add a synchronization mechanism so data is not corrupted. For more details, please refer to section "DOCA Sync Event" in the [NVIDIA DOCA Core Programming Guide](#).

5.2.3. 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.

5.2.4. 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()`).

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. Programming Remote Memory

These sections discuss the creation of a DMA operation that copies memory from the host to the DPU. 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 [samples](#) 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, DPU to host, or even host to host, is always done from the DPU. From the host, it is only possible to copy local memory.

6.1. Host

The host is responsible for allocating local memory and then granting the DPU access to that memory. From there, the DPU can read or write based on the granted permissions.

To allow the copy of the data, the user must first grant the DPU access to the memory where the data resides.

To achieve this:

1. Create the mmap (`doca_mmap_create()`).
2. Add a device (`doca_mmap_dev_add()`).
3. (Optional) Verify that the device supports this operation (`doca_devinfo_get_is_mmap_export_dpu_supported()`).
4. Set the memory range which holds the data to be copied (`doca_mmap_set_memrange()`).
5. Set permissions that allow the DPU access to the memory (`doca_mmap_set_permissions()`).



Note: Set either `DOCA_ACCESS_DPU_READ_ONLY` or `DOCA_ACCESS_DPU_READ_WRITE`.

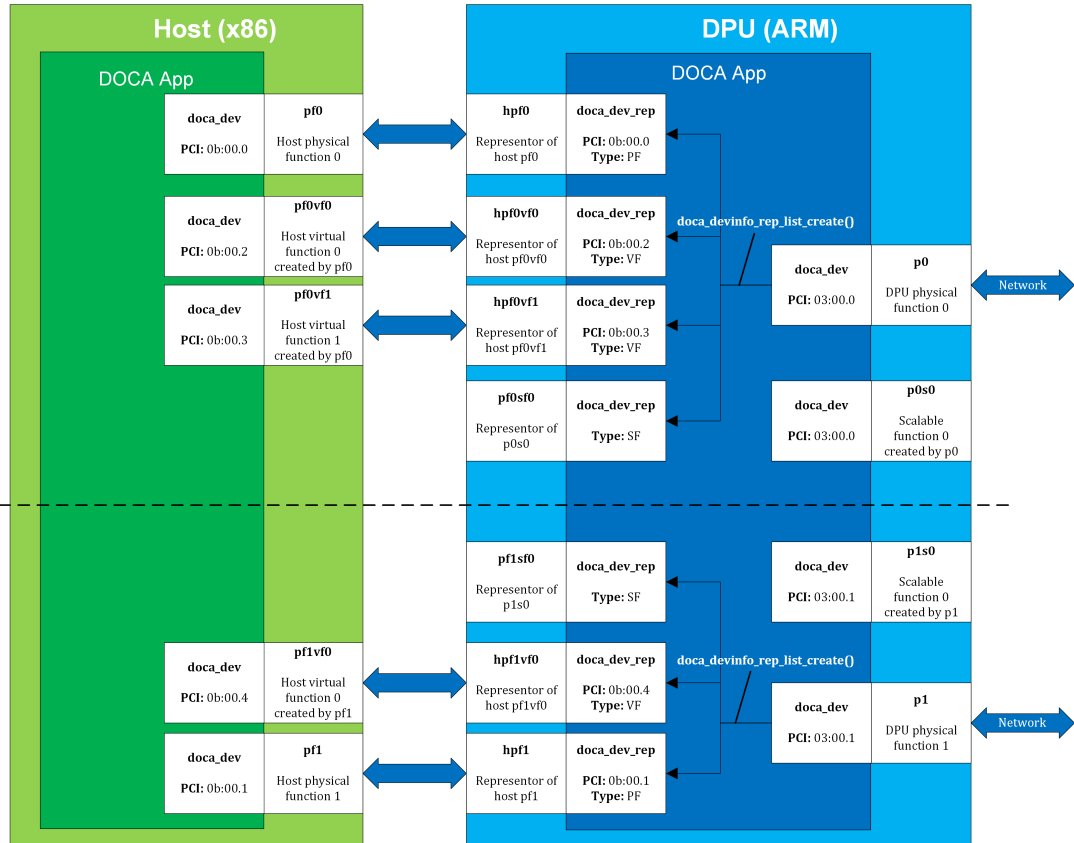
6. Start the mmap (`doca_mmap_start()`).
7. Export the mmap (`doca_mmap_export_dpu()`). This provides a blob that can be used on the DPU side to gain access to the memory.
8. Send the blob to the DPU side. This can be done using a socket, RDMA, or any other transport method. The recommended method is using DOCA Comm Channel.

6.2. DPU

The DPU is responsible for performing the copy operation in either direction (i.e., host to DPU or vice versa).

As with [local memory programming](#), the user must define two mmaps here also: One for the host memory and another for DPU memory.

- ▶ The DPU's mmap holds local memory and can be constructed the same as before.
- ▶ The host's mmap, however, points to non-local, or remote memory (on the host). To obtain the host's mmap:
 1. Receive the mmap blob from host (according to user-defined transport).
 2. (Optional) Verify that the device supports the upcoming operation (`doca_devinfo_get_is_mmap_from_export_dpu_supported()`).
 3. Create the host's mmap (`doca_mmap_create_from_export()`).
 - ▶ This creates and starts an mmap that is ready for use
 - ▶ When allocating buffers from this mmap, the address used is an address known by host
 - ▶ To know what device to use, see the following diagram:



4. (Optional) Find what memory range has been used by the host when creating the mmap (`doca_mmap_get_memrange()`).
5. Allocate a source or destination buffer from the host's mmap (`doca_buf_inventory_buf_by_data()`).
 - a). To copy from the host to the DPU, allocate a source mmap from the host's mmap.
 - b). To copy from the DPU to the host, allocate a destination mmap from the host's mmap.
6. Continue as with [local memory programming](#).

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	<code>-p, --pci-addr</code>	DOCA DMA device PCIe address
	<code>-t, --text</code>	Text to DMA copy from one local buffer to another
DMA Copy Host	<code>-p, --pci-addr</code>	DOCA DMA device PCIe address

Sample	Argument	Description
DMA Copy DPU	-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
	-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`

Notice

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

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

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

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

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

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

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

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

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

Trademarks

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

Copyright

© 2023 NVIDIA Corporation & affiliates. All rights reserved.