NVIDIA DOCA Compress

Programming Guide

MLNX-15-060564 _v2.2.0 | October 2023

Table of Contents

Chapter 1. Introduction

DOCA Compress library provides an API to compress and decompress data using hardware acceleration, supporting both host and DPU memory regions.

The library provides an API for executing compress operations on DOCA buffers, where these buffers reside in either the DPU memory or host memory.

Using DOCA Compress, compress and decompress memory operations can be easily executed in an optimized, hardware-accelerated manner.

For BlueField-2 devices, this library supports:

- Compress operation using the deflate algorithm
- Decompress operation using the deflate algorithm

For BlueField-3 devices, this library supports:

- Decompress operation using the deflate algorithm
- Decompress operation using the LZ4 algorithm

This document is intended for software developers wishing to accelerate their application's compress memory operations.

Chapter 2. Prerequisites

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

Chapter 3. Architecture

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

After initialization, a compress operation is requested by submitting a compress job on the relevant work queue. The DOCA Compress 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 Compress library for general initialization, setup, and clean-up. See later sections for local and remote DOCA Compress operations.

The API for DOCA Compress consists of two DOCA Compress job structures.

- The first one is `struct doca_compress_deflate_job` for operations that use the deflate algorithm

```
struct doca_compress_deflate_job {
 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 *output_chksum;               /**< Output checksum. If it is a compress
 job the
                                             *  checksum calculated is of the
 src_buf.
                                             *  If it is a decompress job the
 checksum result
                                             *  calculated is of the dst_buf.
                                             *  When the job processing will end,
 the output_chksum will
                                             *  contain the CRC checksum result in
 the lower 32bit
                                             *  and the Adler checksum result in the
 upper 32bit. */
};
```

- The second one is `struct doca_compress_lz4_job` for operations that use the LZ4 algorithm

```
struct doca_compress_lz4_job {
 struct doca_job base;                  /**< Common job data. */
 struct doca_buf *dst_buff;             /**< Destination data buffer. */
 struct doca_buf const *src_buff;       /**< Source data buffer.
                                             *  The source buffer must be from local
 memory.
                                             *  Note: when using doca_buf linked
 list, the length of the
                                             *  first data element in the source
 buffer
                                             *  must be at least 4B. */
    uint64_t *output_chksum;               /**< Output checksum. If it is a compress
 job the
                                             *  checksum calculated is of the
 src_buf.
                                             *  If it is a decompress job the
 checksum result
                                             *  calculated is of the dst_buf.
                                             *  When the job processing will end,
 the output_chksum will
```

```
                                               *  contain the CRC checksum result in
 the lower 32bit
                                               *  and the Adler checksum result in the
 upper 32bit. */
};
```

These structures are passed to the work queue to instruct the library on the source, destination, and checksum output.

The source and destination buffers should not overlap, while the `data_len` field of the source `doca_buf` defines the number of bytes to compress/decompress and the `data` field of the source `doca_buf` defines the location in the source buffer to compress/decompress from, to the destination buffer.

DOCA Compress library calculates the checksum and stores the result inside the `output_chksum` field. The field length is 64 bits, where the lower 32 bits contain the CRC checksum result and the upper 32 bits contain the Adler checksum result.

As with other libraries, the compress job contains the standard `doca_job` base field that must be set as follows:

- For a deflate compress job:
  ```
  /* Construct Compress job */
  doca_job.type = DOCA_COMPRESS_DEFLATE_JOB;
  doca_job.flags = DOCA_JOB_FLAGS_NONE;
  doca_job.ctx = doca_compress_as_ctx(doca_compress_inst);
  ```
- For a deflate decompress job:
  ```
  /* Construct Deflate Decompress job */
  doca_job.type = DOCA_DECOMPRESS_DEFLATE_JOB;
  doca_job.flags = DOCA_JOB_FLAGS_NONE;
  doca_job.ctx = doca_compress_as_ctx(doca_compress_inst);
  ```
- For LZ4 decompress job:
  ```
  /* Construct LZ4 Decompress job */
  doca_job.type = DOCA_DECOMPRESS_LZ4_JOB;
  doca_job.flags = DOCA_JOB_FLAGS_NONE;
  doca_job.ctx = doca_compress_as_ctx(doca_compress_inst);
  ```

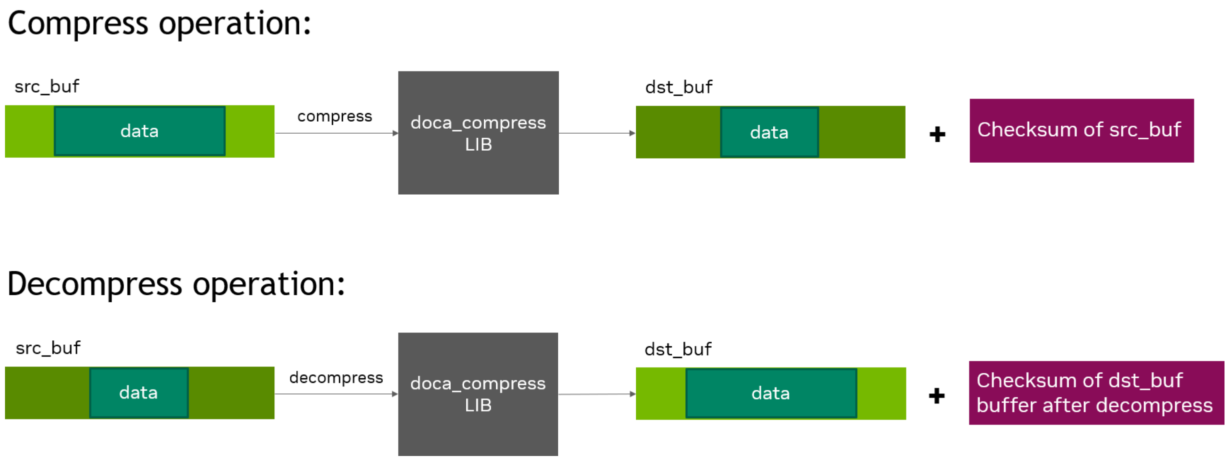
Compress job-specific fields should be set based on the required source and destination buffers. The user can provide output parameter so the library can store the checksum result in it or NULL.

```
compress_job.base = doca_job;
compress_job.dst_buff = dst_doca_buf;
compress_job.src_buff = src_doca_buf;
compress_job.output_chksum = output_chksum;
```

To get the job result from the WorkQ, depending on the WorkQ working mode, the application can either periodically poll the work queue or wait for event on the work queue (via the `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
```


Compress operation:
src_buf
data
compress
doca_compress LIB
dst_buf
data
+
Checksum of src_buf
Decompress operation:
src_buf
data
decompress
doca_compress LIB
dst_buf
data
+
Checksum of dst_buf buffer after decompress

Chapter 5. Local Memory Programming Guide

These sections discuss the usage of the DOCA Compress library in real-world situations. Most of this section utilizes code which is available through the DOCA Compress sample projects located under `/samples/doca_compress/` and application projects located under `/applications/file_compression`.

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 compress/ decompress operation.

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

5.1. Initialization Process

The DOCA Compress API uses the DOCA core library to create the required objects (memory map, inventory, buffers, etc.) for the DOCA Compress library 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 (e.g., PCIe address, etc). You can also use the function `doca_compress_job_get_supported` to check if the device is suitable for the compress job type you want to perform. After this, the device should be opened using `doca_dev_open`.

5.1.2. Creating DOCA Core Objects

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

Once a DOCA Compress instance has been created, it can be used as a context using the `doca_ctx` APIs this can be achieved by getting a context representation using `doca_compress_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

Prior to starting the mmap (`doca_mmap_start`), make sure that you set the memory range correctly (via `doca_mmap_set_memrange`). 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. WorkQ Initialization

There are two options for the WorkQ working mode, the default polling mode or event-driven mode.

To set the WorkQ to work in event-driven mode, use `doca_workq_set_event_driven_enable` and then `doca_workq_get_event_handle` to get the event handle of the WorkQ so you can wait on events using epoll or other Linux wait for event interfaces.

5.1.3.4. DOCA Compress Context Initialization

The context created previously (via `doca_compress_create()`) and acquired using (`doca_compress_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 compress 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.

Finally, you must set the data address and length of the source DOCA buffer using the function `doca_buf_set_data`. This field determines the data address and the data length perform de/compress on.

To know the maximum `data_len` of the DOCA buffers that can be used to perform a de/compress operation on, users must call the function `doca_compress_get_max_buffer_size`.

5.2. Compress Execution

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

5.2.1. Constructing and Executing DOCA Compress Operation

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

Within the compress job structure, the context field must point to your DOCA Compress context, and the `type` field must be set to:

- `DOCA_COMPRESS_DEFLATE_JOB` for a deflate compress operation
- `DOCA_DECOMPRESS_DEFLATE_JOB` for a deflate decompress operation
- `DOCA_DECOMPRESS_LZ4_JOB` for LZ4 decompress operation

The DOCA Compress specific elements of the job point to your DOCA buffers for the source and destination and to a checksum field that uses to store the checksum result from the hardware.

Note that if it is a compress job, the checksum result calculated is of the source buffer. If it is a decompress job, the checksum result calculated is of the destination buffer.

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

5.2.2. Waiting for Completion

According to the WorkQ mode, you can detect when the compress operation has completed (via `doca_workq_progress_retrieve`):

- WorkQ operates in polling mode – periodically poll the work queue until the API call indicates that a valid event has been received
- WorkQ operates in event mode – while `doca_workq_progress_retrieve` does not return a success result, perform the following loop:
    1. Arm the WorkQ `doca_workq_event_handle_arm`.
    2. Wait for an event using the event handle (e.g., using `epoll_wait()`).
    3. Once the thread wakes up, call `doca_workq_event_handle_clear`.

Regardless of the operating mode, you should be able to detect the success of the compress operation if the `event.result.u64` field is equal to `DOCA_SUCCESS`. It should be noted that other work queue operations (i.e., non-compress operations) present their events differently. Refer to their respective guides for more information.

If there is already data inside the destination buffer, the DOCA Compress library appends the compress operation result after the existing data. Otherwise, the DOCA Compress library stores the new data in the data address of the destination buffer. Either way the library keeps the data address unchanged and increases the `data_len` field of the destination buffer by the number of bytes produced by the de/compress operation.

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 are 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 happen in any order, but destruction must occur on the work queue (`doca_workq_destroy`), compress context (`doca_compress_destroy`), buf inventory (`doca_buf_inventory_destroy`), mmap (`doca_mmap_destroy`), and device closure (`doca_dev_close`).

Chapter 6. Remote Memory Programming Guide

This section covers the creation of a remote memory DOCA Compress operation. This operation allows memory from the host, accessible by DOCA Compress on the DPU, to be used as a source or destination.

6.1. Sender

The sender holds the source memory to perform the compress operation on and sends it to the DPU. The developer decides the method of how the source memory address is transmitted to the DPU. For example, it can be a socket that is connected from a "local" host sender to a "remote" BlueField DPU receiver. The address is passed using 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 exactly memory region (`doca_mmap_set_memrange`) and call a special mmap function (`doca_mmap_export`).

This function generates a descriptor object that can be transmitted to the DPU. The information in the descriptor object refers to the exported "remote" host 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 DOCA buffer (via `doca_buf_inventory_buf_by_addr`) to represent the host memory, you should call the special mmap function that retrieves the remote mmap from the host (`doca_mmap_create_from_export`). The DOCA buffer can then be created using this remote mmap and used as source/destination buffer in the DOCA Compress job structures.

> Note: When using an LZ4 operation the source buffer must be from local memory.

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 Compress SG Support

DOCA compress library supports scatter-gather (SG) DOCA buffers. You can use a `doca_buf` with a linked list extension as the source buffer in the `doca_compress` job. The library then compresses/decompresses all the content of the DOCA buffers to a single destination buffer.

> Note: The length of the linked list for the source buffer must not exceed the return value from the function `doca_compress_get_max_list_buf_num_elem`. The destination buffer must be a single buffer.

> Note: When using SG for a LZ4 decompress operation, the first data element in the source buffer must be at least 4B.

Chapter 8. DOCA Compress Samples

This document describes compress samples based on the DOCA Compress library. These samples illustrate how to use the DOCA Compress API to compress and decompress files.

8.1. 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_compress/<sample_name>
meson build
ninja -C build
```

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

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

```
Usage: doca_compress_deflate [DOCA Flags] [Program Flags]
DOCA Flags:
  -h, --help                      Print a help synopsis
  -v, --version                   Print program version information
  -l, --log-level                 Set the log level for the program <CRITICAL=20,
 ERROR=30, WARNING=40, INFO=50, DEBUG=60>

Program Flags:
  -p, --pci-addr                  PCI device address
  -f, --file                      input file to compress/decompress
  -m, --mode                      mode - {compress, decompress}
  -o, --output                    output file
```

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

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

8.2. Samples

8.2.1. Compress Deflate

This sample illustrates how to use DOCA Compress library to compress and decompress a file.

The sample logic includes:

1. Locating a DOCA device.
2. Initializing the required DOCA core structures.
3. Populating DOCA memory map with two relevant buffers; one for the source data and one for the result.
4. Allocating elements in DOCA buffer inventory for each buffer.
5. Initializing DOCA Compress job object.
6. Submitting a compress or decompress job into the work queue.
7. Retrieving the job from the queue once it is done.
8. Writing the result into an output file, `out.txt`.
9. Destroying all compress and DOCA core structures.

References:

- `/opt/mellanox/doca/samples/doca_compress/compress_deflate/compress_deflate_sample.c`
- `/opt/mellanox/doca/samples/doca_compress/compress_deflate/compress_deflate_main.c`
- `/opt/mellanox/doca/samples/doca_compress/compress_deflate/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 NONINFRINGEMENT, 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.

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