NVIDIA DOCA FlexIO SDK

Programming Guide

MLNX-15-060574 _v1.5.1 | November 2022

Table of Contents

Chapter 1. Introduction

The datapath accelerator (DPA) processor is an auxiliary processor designed to accelerate packet processing and other datapath operations. The FlexIO SDK exposes an API for managing the device and executing native code over it.

The DPA processor is supported on NVIDIA® BlueField®-3 DPUs and later generations.

Chapter 2. Prerequisites

DOCA FlexIO applications can run either on the host machine or on the target DPU.

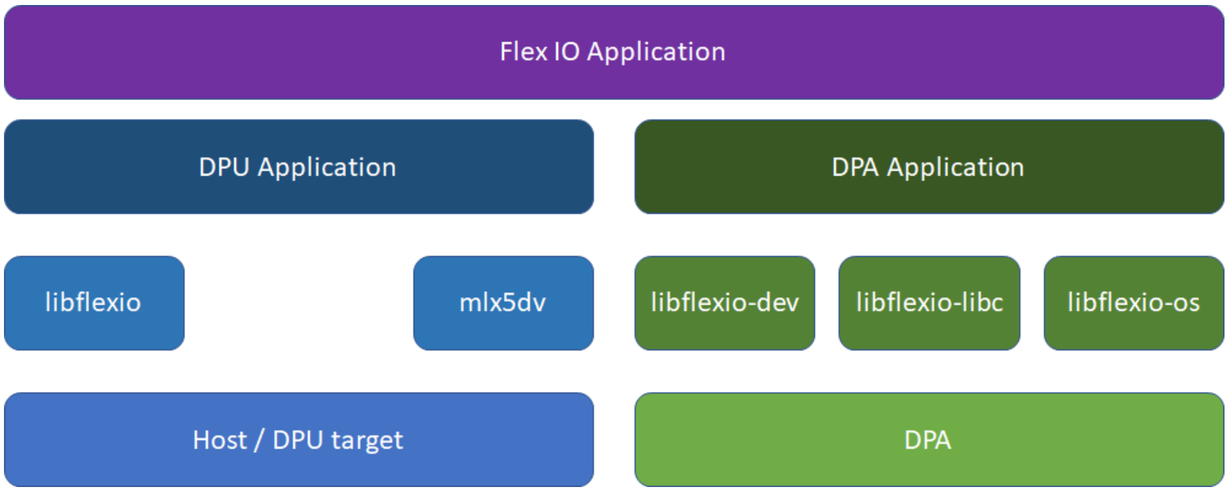
Developing programs over FlexIO SDK requires knowledge of DPU networking queue usage and management.

Chapter 3. Architecture

FlexIO SDK library exposes a few layers of functionality:

- `libflexio` – library for DPU-side operations. Mainly used for resource management.
- `libflexio_dev` – library for DPA-side operations. Mainly used for data path implementation.
- `libflexio_os` – library for DPA OS-level access
- `libflexio_libc` – custom LibC for DPA

A typical application is composed of two parts: One running on the host machine or the DPU target and another running directly over the DPA.



Chapter 4. API

For the driver API reference, refer to the NVIDIA DOCA Driver APIs Reference Manual.

Chapter 5. Resource Management

DPA programs cannot create resources. The responsibility of creating resources, such as FlexIO process, thread, outbox and window, as well as queues for packet processing (completion, receive and send), lies on the DPU program. The relevant information should be communicated (copied) to the DPA side and the address of the copied information should be passed as an argument to the running thread.

5.1. Example

DPU side:

1. Declare a variable to hold the DPA buffer address.
   ```
   flexio_uintptr_t app_data_dpa_daddr;
   ```
2. Allocate a buffer on the DPA side.
   ```
   flexio_buf_dev_alloc(flexio_process, sizeof(struct my_app_data),
    &app_data_dpa_daddr);
   ```
3. Copy application data to the DPA buffer.
   ```
   flexio_host2dev_memcpy(flexio_process, (uintptr_t)app_data, sizeof(struct
    my_app_data), app_data_dpa_daddr);
   ```

   `struct my_app_data` should be common between the DPU and DPA applications so the DPA application can access the struct fields.

   The event handler should get the address to the DPA buffer with the copied data:
   ```
   flexio_event_handler_create(flexio_process, net_entry_point, app_data_dpa_daddr,
    NULL, flexio_outbox, &app_ctx.net_event_handler)
   ```

DPA side:

```
__dpa_rpc__ uint64_t event_handler_init(uint64_t thread_arg)
{
struct my_app_data *app_data;
app_data = (my_app_data *)thread_arg;
...
}
```

Chapter 6. DPA Memory Management

As mentioned previously, the DPU program is responsible for allocating buffers on the DPA side (same as resources). The DPU program should allocate device memory in advance for the DPA program needs (e.g., queues data buffer and rings, buffers for the program functionality, etc).

The DPU program is also responsible for releasing the allocated memory. For this purpose, the FlexIO SDK API exposes the following memory management functions:

```
flexio_status flexio_buf_dev_alloc(struct flexio_process *process, size_t
 buff_bsize, flexio_uintptr_t *dest_daddr_p);
flexio_status flexio_buf_dev_free(flexio_uintptr_t daddr_p);
flexio_status flexio_host2dev_memcpy(struct flexio_process *process, void
 *src_haddr, size_t buff_bsize, flexio_uintptr_t dest_daddr);
flexio_status flexio_buf_dev_memset(struct flexio_process *process, int value,
 size_t buff_bsize, flexio_uintptr_t dest_daddr);
```

6.1. Buffers/Rings For DPA Queues

The FlexIO SDK exposes an API for allocating work queues and completion queues for the DPA. This means that the DPA may have direct access and control over these queues, allowing it to create doorbells and access their memory.

When creating a FlexIO SDK queue, the user must pre-allocate and provide memory buffers for the queue's element buffer/ring. This buffer may be allocated on the DPU or the DPA memory.

To this end, the FlexIO SDK exposes the `flexio_qmem` struct, which allows the user to provide the buffer address and type (DPA or DPU). For addresses on the DPU memory, the user must also provide the relevant DUMEM ID. But for addresses on the DPA memory, this is not needed as the relevant process DUMEM ID is used.

6.2. Memory Allocation Best Practices

To optimize process DUMEM memory allocation, it is recommended to use the following allocation sizes (or closest to it):

- Up to 1 page (4KB)
- 2^6 pages (256KB)
- 2^{11} pages (8MB)

- 2^{16} pages (256MB)

Using these sizes minimizes memory fragmentation over the process DUMEM heap. If other buffer sizes are required, it is recommended to round the allocation up to one of the listed sizes and use it for multiple buffers.

Chapter 7. DPA Window

To use the window functionality, DPU memory must be registered for the device using the `ibv_reg_mr()` call. Two of this call parameters are the host address and size to be registered.

Chapter 8. DPA Side Memory and I/O Barriers

To ensure correctness, certain operations require a barrier. These barriers are exposed in the `libflexio_os_mb.h` header file:

```
flexio_os_dma_to_dev_wr_barrier();
flexio_os_dma_to_dev_wr_order_barrier();
flexio_os_dma_from_dev_rd_barrier();
flexio_os_window_write_barrier();
flexio_os_window_read_barrier();
flexio_os_window_rw_barrier();
flexio_os_all_barrier();
```

Some cases that require a barrier are handled by the SDK such as outbox and window configuration and DPU memory access through the window.

Chapter 9. DPA Event Handler

9.1. Default Window/Outbox

The DPA event handler expects a DPA window and DPA outbox structs on creation. These are used as the default for the event handler thread. The user may choose to set one or both to NULL, in which case there will be no valid default value for one/both of them.

Upon thread invocation on the DPA side, the thread context is set for the provided default IDs. If at any point the outbox/window IDs are changed, then the thread context on the next invocation is restored to the default IDs. This means that the DPA Window MKey must be configured each time the thread is invoked, as it has no default value.

9.2. HART Management

DPA HARTs are the equivalent of a logical core. A DPA program must be assigned a HART in order to execute.

It is possible to set HART affinity for an event handler upon creation. This causes the event handler to execute its DPA program over specific HARTs (or a group of HARTs).

DPA supports three types of affinity:

- None – select a HART from all available HARTs
- Strict – select only the specified HART (by ID)
- Group – select a HART from all the HARTs in the specified group (by ID, must be created in advance)

The affinity type and ID, if applicable, are passed to the event handler upon creation using the `affinity` field of the `flexio_event_handler_attr` struct.

Chapter 10. Application Debugging

Since application execution is divided between the DPU side and the DPA processor services, debugging may be somewhat challenging, especially considering the fact that the DPA side does not have a terminal allowing the use of the C stdio library printf services.

10.1. Using Device Prints API

Another logging option is to use FlexIO SDK infrastructure to write strings from the DPA side to the DPU side console or file. The DPU side's `flexio.h` file provides the `flexio_print_init` API call for initializing the required infrastructures to support this. Once initialized, the DPA side must have the thread context, which can be obtained by calling `flexio_dev_get_thread_ctx`. `flexio_dev_print` can then be called to write a string to the DPA side where it is directed to the console or a file, according to user configuration in the init stage.

It is important to call `flexio_print_destroy()` when exiting the DPU application to ensure proper clean-up of the print mechanism resources.

Note: Device prints use an internal QP for the communication between the DPA and the DPU. When running over an InfiniBand fabric, the user must ensure that the subnet is well-configured and that the relevant device's port is in `active` state.

Note: IMPORTANT! `flexio_print_init` should only be called once per process. Calling it multiple times recreates print resources which may cause a resource leak and other malfunctions.

10.1.1. Printf Support

Only limited functionality is implemented for printf. Not all libc printf is supported.

Please consult the following list for supported modifiers:

- Formats – `%c`, `%s`, `%d`, `%ld`, `%u`, `%lu`, `%i`, `%li`, `%x`, `%hx`, `%hxx`, `%lx`, `%X`, `%lX`, `%o`, `%lo`, `%p`, `%%`
- Flags – `.`, `*`, `-`, `+`, `#`
- General supported modifiers:
    - "0" padding

    - Min/max characters in string
- General unsupported modifiers:
    - Floating point modifiers – `%e`, `%E`, `%f`, `%lf`, `%LF`
    - Precision modifiers

Chapter 11. Samples

Please refer to NVIDIA DOCA FlexIO Sample Guide for more information about the API of this DOCA library.