NVIDIA DOCA DPA Subsystem

Programming Guide

MLNX-15-060584 _v2.0.2 | May 2023

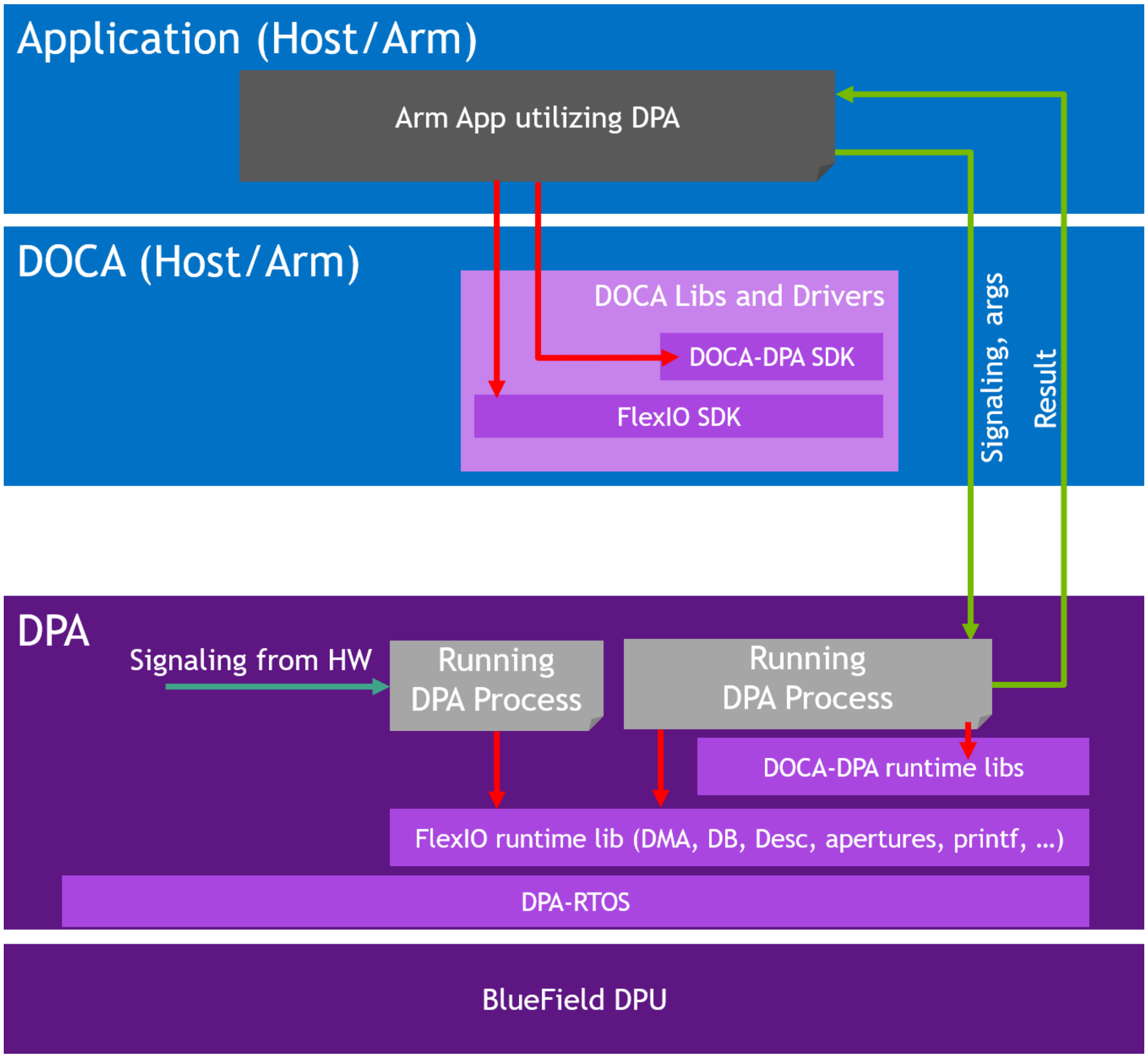
Table of Contents

Chapter 1. DPA Subsystem Overview

The NVIDIA® BlueField®-3 data-path accelerator (DPA) is an embedded subsystem designed to accelerate workloads that require high-performance access to the NIC engines in certain packet and I/O processing workloads. Applications leveraging DPA capabilities run faster on the DPA than on host. Unlike other programmable embedded technologies, such as FPGAs, the DPA enables a high degree of programmability using the C programming model, multi-process support, tools chains like compilers and debuggers, SDKs, dynamic application loading, and management.

The DPA architecture is optimized for executing packet and I/O processing workloads. As such, the DPA subsystem is characterized by having many execution units that can work in parallel to overcome latency issues (such as access to host memory) and provide an overall higher throughput.

The following diagram illustrates the DPA subsystem. The application accesses the DPA through the DOCA library (DOCA DPA) or the DOCA driver layer (FlexIO SDK). On the host or DPU side, the application loads its code into the DPA (shown as "Running DPA Process") as well as allocates memory, NIC queues, and more resources for the DPA process to access. The DPA process can use device side libraries to access the resources. The provided APIs support signaling of the DPA process from the host or DPU to explicitly pass control or to obtain results from the DPA.



The threads on the DPA can react independently to incoming messages via interrupts from the hardware, thereby providing full bypass of DPU or Arm CPU for datapath operations.

1.1. DPA Platform Design

1.1.1. Multiple Processes on Multiple Execution Units

The DPA platform supports multiple processes with each process having multiple threads. Each thread can be mapped to a different execution unit to achieve parallel execution. The processes operate within their own address spaces and their execution contexts are isolated. Processes are loaded and unloaded dynamically per the user's request. This is achieved by the platform's hardware design (i.e., privilege layers, memory translation units, DMA engines) and a light-weight real-time operating system (RTOS). The RTOS enforces the privileges and isolation among the different processes.

1.1.2. DPA RTOS

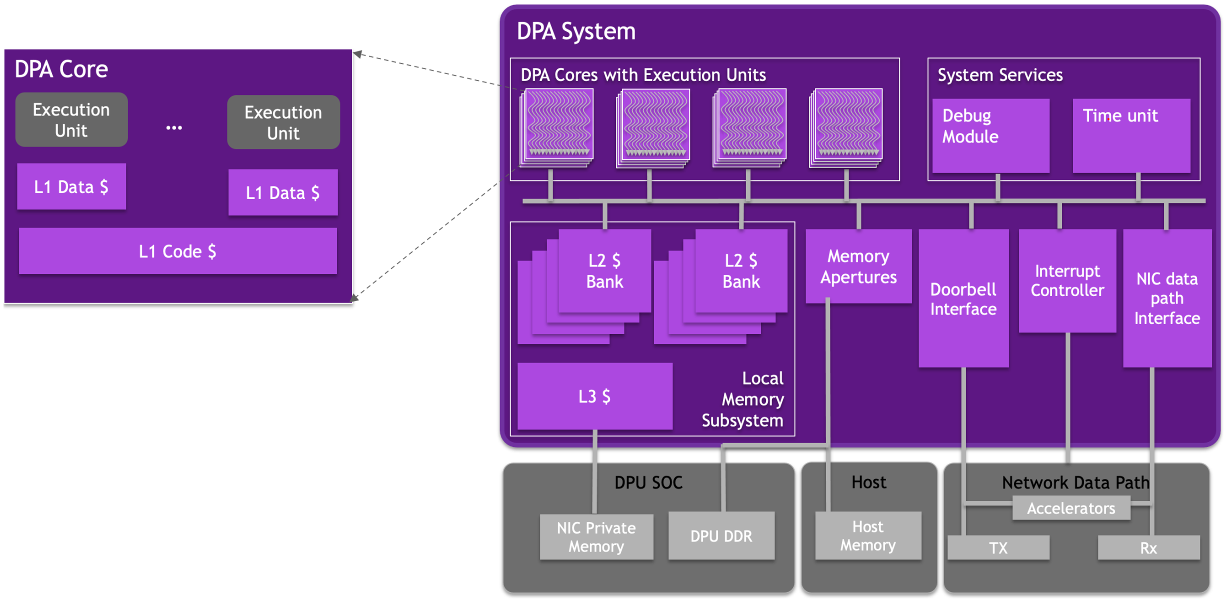
The RTOS is designed to rely on hardware-based scheduling to enable low activation latency for the execution handlers. The RTOS works in a cooperative run-to-completion scheduling model.

Under cooperative scheduling, an execution handler can use the execution unit without interrupts until it relinquishes it. Once relinquished, the execution unit is handed back to the RTOS to schedule the next handler. The RTOS sets a watchdog for the handlers to prevent any handler from unduly monopolizing the execution units.

1.1.3. DPA Memory and Caches

The following diagram illustrates the DPA memory hierarchy. Memory accessed by the DPA can be cached at three levels (L1, L2, and L3). Each execution unit has a private L1 data cache. The L1 code cache is shared among all the execution units in a DPA core. The L2 cache is shared among all the DPA cores. The DPA execution units can access external memory via load/store operations through the Memory Apertures.

The external memory that is fetched can be cached directly in L1. The DPA caches are backed by NIC private memory, which is located in the DPU's DDR memory banks. Therefore, the address spaces are scalable and bound only by the size of the NIC's private memory, which in turn is limited only by the DPU's DDR capacity.



See section Memory Model for more details.

1.1.4. DPA Access to NIC Accelerators

The DPA can send and receive any kind of packet toward the NIC and utilize all the accelerators that reside on the BlueField DPU (e.g., encryption/decryption, hash computation, compression/decompression).

The DPA platform has efficient DMA accelerators that enable the different execution units to access any memory location accessible by the NIC in parallel and without contention. This includes both synchronous and asynchronous DMA operations triggered by the execution units. In addition, the NIC can DMA data to the DPA caches to enable low-latency access and fast processing. For example, a packet received from the wire may be "DMA-gathered" directly to the DPA's last level caches.

1.1.5. Compilation with DPACC

DPACC is a compiler for the DPA processor. It compiles code targeted for the DPA processor into an executable and generates a DPA program. A DPA program is a host library with interfaces encapsulating the DPA executable.

This DPA program is linked with the host application to generate a host executable. The host executable can invoke the DPA code through the DPA SDK's runtime.

Please refer to NVIDIA DOCA DPACC Compiler User Guide for more details.

1.2. DOCA Libs and Drivers

The NVIDIA DOCA framework is the key for unlocking the potential of the BlueField DPU.

DOCA's software environment allows developers to program the DPA to accelerate workloads. Specifically, DOCA includes:

- DOCA DPA SDK – a high-level SDK for application-level protocol acceleration
- DOCA FlexIO SDK – a low-level SDK to load DPA programs into the DPA, manage the DPA memory, create the execution handlers and the needed hardware rings and contexts
- DPACC – DPA toolchain for compiling and ELF file manipulation of the DPA code

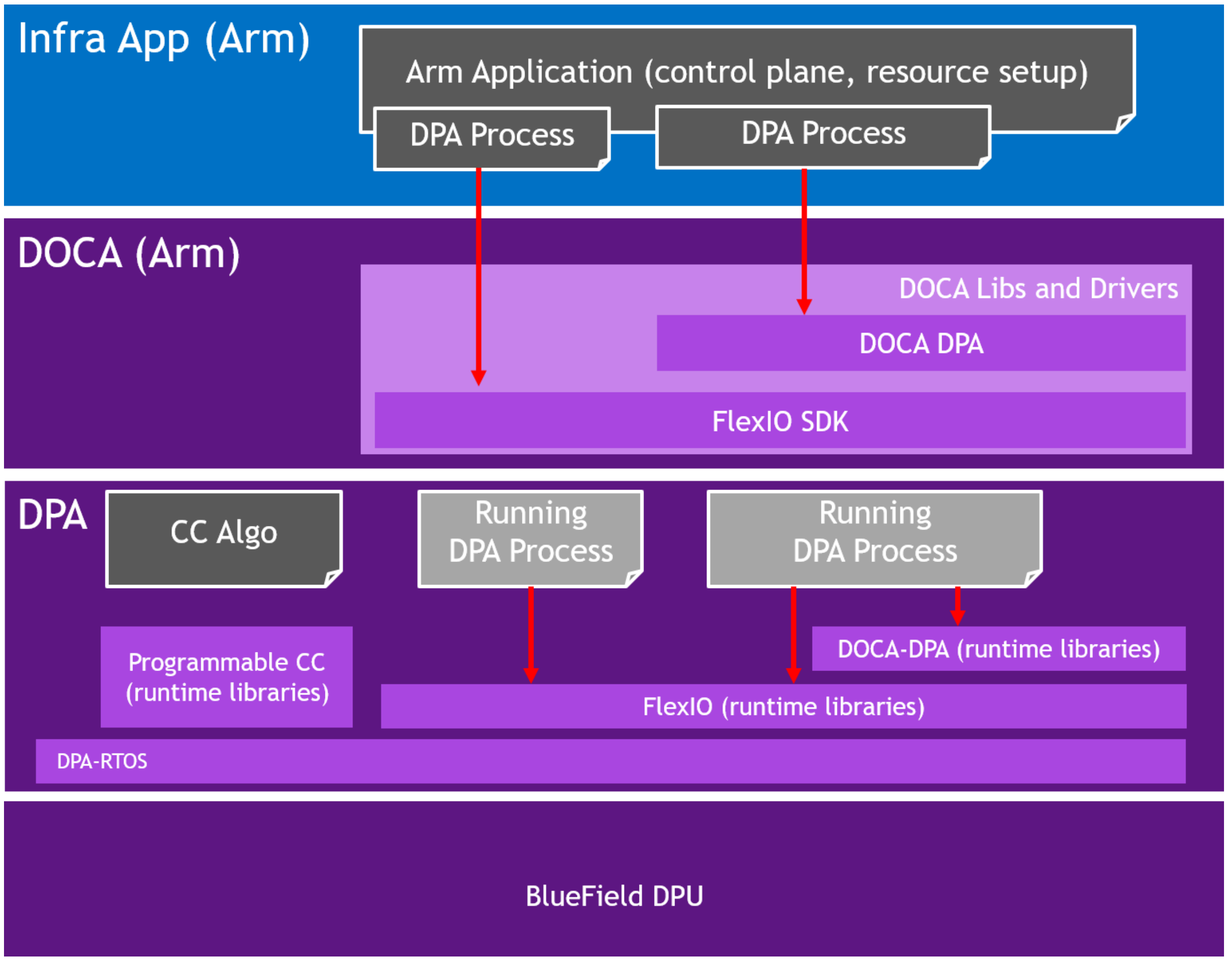
1.3. Programming Model

The DPA is intended to accelerate datapath operations for the DPU and host CPU. The accelerated portion of the application using DPA is presented as a library for the host application. The code within the library is invoked in an event-driven manner in the context of a process that is running on the DPA. One or many DPA execution units may work to handle the work associated with network events. The programmer specifies different conditions when each function should be called using the appropriate SDK APIs on the host or DPU.

Note: The DPA cannot be used as a standalone CPU.

Management of the DPA, such as loading processes and allocating memory, is performed from a host or DPU process. The host process discovers the DPA capabilities on the device and drives the control plane to set up the different DPA objects. The DPA objects exist as long as the host process exists. When the host process is destroyed, the DPA objects are freed. The host process decides which functions it wants to accelerate using the DPA: Either its entire data plane or only a part of it.

The following diagram illustrates the different processes that exist in the system:



1.3.1. FlexIO

Note: Supported at beta level.

FlexIO is a low-level event-driven library to program and accelerate functions on the DPA.

1.3.1.1. FlexIO Execution Model

To load an application onto the DPA, the user must create a process on the DPA, called a FlexIO process. FlexIO processes are isolated from each other like standard host OS processes.

FlexIO supports the following options for executing a user-defined function on the DPA:

1. FlexIO event hander – the event handler executes its function each time an event occurs. An event on this context is a completion event (CQE) received on the NIC completion queue (CQ) when the CQ was in the armed state. The event triggers an internal DPA interrupt that activates the event handler. When the event handler is activated, it is provided with a user-defined argument. The argument in most cases is a pointer to the software execution context of the event handler.

   The following pseudo-code example describes how to create an event handler and attach it to a CQ:

```
// Device code
__dpa_global__ myFunc(void *myArg){
      struct my_db *db = (struct my_db *)myArg;
      get_completion(db->myCq)
      work();
      arm_cq(myCq);
      return;
}

// Host code
main() {

      /* Load the application code into the DPA */
      flexio_process_create(device, application, &myProcess);


      /* create event handler to run my_func with my_arg */
      flexio_event_handler_create(myProcess, myFunc, myArg, &myEventHandler);

      /* Associate the event hanlder with a specific CQ */
      create_cq(&myCQ,… , myEventHandler)


      /* start the event handler */
      flexio_event_handler_run(myEventHandler)
      …

}
```

2. RPC – remote, synchronous, one-time call of a specific function. RPC is mainly used for the control path to update DPA memory contexts of a process. The RPC's return value is reported back to the host application.

   The following pseudo-code example describes how to use the RPC:

```
// Device code
__dpa_rpc__ myFunc(myArg) {
      struct my_db *db = (struct my_db *)myArg;
      if (db->flag) return 1;
      db->flag = 1;
      return 0;
}

// Host code
main() {
      …

      /* Load the application code into the DPA */
      flexio_process_create(device, application, &myProcess);

      /* run the function */
      flexio_process_call(myProcess, myFunc, myArg, &returnValue);
      …
}
```

1.3.1.2. FlexIO Memory Management

The DPA process can access several memory locations:

- Global variables defined in the DPA process
- Stack memory – local to the DPA execution unit. Stack memory is not guaranteed to be preserved between different execution of the same handler.
- Heap memory – this is the process' main memory. The heap memory contents are preserved if the DPA process is active.
- External registered memory – remote to the DPA but local to the server. The DPA can access any memory location that can be registered to the local NIC using the provided API. This includes BlueField DRAM, external host DRAM, GPU memory, and more.

The heap and external registered memory locations are managed from the host process. The DPA execution units can load/store from stack/heap and external memory locations. Note that for external memory locations, the window should be configured appropriately using FlexIO Window APIs.

FlexIO allows the user to allocate and populate heap memory on the DPA. The memory can later be used by in the DPA application as an argument to the execution context (RPC and event handler):

```
/* Load the application code into the DPA */
flexio_process_create(device, application, &myProcess);

/* allocate some memory */
flexio_buf_dev_alloc(process, size, ptr)

/* populate it with user defined data */
flexio_host2dev_memcpy(process, src, size, ptr)

/* run the function */
flexio_process_call(myProcess, function, ptr, &return value);
```

FlexIO allows accessing external registered memory from the DPA execution units using FlexIO Window. FlexIO Window maps a memory region from the DPA process address space to an external registered memory. A memory key for the external memory region is required to be associated with the window. The memory key is used for address translation and protection. FlexIO window is created by the host process and is configured and used by the DPA handler during execution. Once configured, LD/ST from the DPA execution units access the external memory directly.

The access for external memory is not coherent. As such, an explicit memory fencing is required to flush the cached data to maintain consistency. See section Memory Fences for more information.

The following example code demonstrates the window management:

```
// Device code
__dpa_rpc__ myFunc(arg1, arg2, arg3)
{
    struct flexio_dev_thread_ctx *dtctx;
    flexio_dev_get_thread_ctx(&dtctx);
    uint32_t windowId = arg1;
    uint32_t mkey = arg2;
    uint64_t *dev_ptr;
    flexio_dev_window_config(dtctx, windowId, mkey );
    /* get ptr to the external memory (arg3) from the DPA process address space */
    flexio_dev_status status = flexio_dev_window_ptr_acquire (dtctx, arg3, dev_ptr);
    /* will set the external memory */
    *dev_ptr = 0xff;
    /* flush the data out */
    __dpa_thread_window_writeback();
    return 0;
}

// Host code
main() {
      /* Load the application code into the DPA */
      flexio_process_create(device, application, &myProcess);
      /* define an array on host */
      uint64_t var= {0};
      /* register host buffer  */
      mkey =ibv_reg_mr(&var, …)
      /* create the window */
      flexio_window_create(process, doca_device->pd, mkey, &window_ctx);
      /* run the function */
      flexio_process_call(myProcess, myFunc, flexio_window_get_id(window_ctx), mkey,
 &var, &returnValue);
}
```

1.3.1.3. Send and Receive Operation

A DPA process can initiate send and receive operations using the FlexIO outbox object. The FlexIO outbox contains memory-mapped IO registers that enable the DPA application to issue device doorbells to manage the send and receive planes. The DPA outbox can be configured during run time to perform send and receive from a specific NIC function exposed by the DPU. This capability is not available for host CPUs that can only access their assigned NIC function.

Each DPA execution engine has its own outbox. As such, each handler can efficiently use the outbox without needing to lock to protect against accesses from other handlers. To enforce the required security and isolation, the DPA outbox enables the DPA application to send and receive only for queues created by the DPA host process and only for NIC functions the process is allowed to access.

Like the FlexIO window, the FlexIO outbox is created by the host process and configured and used at run time by the DPA process.

```
// Device code
__dpa_rpc__ myFunc(arg1,arg2,arg3) {

    struct flexio_dev_thread_ctx *dtctx;

    flexio_dev_get_thread_ctx(&dtctx);

    uint32_t outbox = arg1;
    flexio_dev_outbox_config (dtctx, outbox);
    /* create some wqe and post it on sq */

    /* send DB on sq*/

    flexio_dev_qp_sq_ring_db(dtctx, sq_pi,arg3);

    /* poll CQ (cq number is in arg2) */

}

// Host code
main() {

      /* Load the application code into the DPA */
      flexio_process_create(device, application, &myProcess);

      /* allocate uar */
      uar = ibv_alloc_uar(ibv_ctx);

      /* create queues*/
      flexio_cq_create(myProcess, ibv_ctx, uar, cq_attr, &myCQ);
      my_hwcq = flexio_cq_get_hw_cq (myCQ);

      flexio_sq_create(myProcess, ibv_ctx, myCQ, uar, sq_attr, &mySQ);
      my_hwsq = flexio_sq_get_hw_sq(mySQ);

      /* Outbox will allow access only for queues created with the same UAR*/
      flexio_outbox_create(process, ibv_ctx, uar, &myOutbox);

      /* run the function */
      flexio_process_call(myProcess, myFunc, myOutbox, my_hwcq->cq_num, my_hwsq-
>sq_num,  &return_value);
}
```

1.3.1.4. Synchronization Primitives

The DPA execution units support atomic instructions to protect from concurrent access to the DPA process heap memory. Using those instructions, multiple synchronization primitives can be designed.

FlexIO currently supports basic spin lock primitives. More advanced thread pipelining can be achieved using DOCA DPA events.

1.3.2. DOCA DPA

Note: Supported at alpha level.

The DOCA DPA SDK eases DPA code management by providing high-level primitives for DPA work offloading, synchronization, and communication. This leads to simpler code but lacks the low-level control that FlexIO SDK provides.

User-level applications and libraries wishing to utilize the DPA to offload their code may choose DOCA DPA. Use-cases closer to the driver level and requiring access to low-level NIC features would be better served using FlexIO.

The implementation of DOCA DPA is based on the FlexIO API. The higher level of abstraction enables the user to focus on their program logic and not the low-level mechanics.

1.3.2.1. Host-to-DPA Work Submission

The work submission APIs enable a host application to invoke a function on the DPA and supply it with arguments. The work is executed in an asynchronous manner only when the work's dependencies are satisfied. Using this model, the user can define an arbitrary sequence of self-triggering work on the DPA. To borrow common CUDA terminology, these functions are called "kernels". This frees up the host's CPU to focus on its tasks after submitting the list of work to the DPA.

The following is an example where the host submits three functions, `func1`, `func2`, and `func3` that execute one after the other. The functions are chained using a DPA event, which is an abstract data type that contains a 64-bit counter.

```
/* func1 -> func2 -> func3 */
/* wait on event, until threshold of 0 (satisfied immediately),
 * add 1 to the same event when func1 is complete */
doca_dpa_kernel_launch(dpa, event, 0, event, 1, …, func1, nthreads, <args>);

/* wait on event, until threshold of 1 (i.e., wait for func1),
 * add 1 to event when func2 is complete */
doca_dpa_kernel_launch(dpa, event, 1, event, 1, …, func2, nthreads, <args>);

/* wait on event, until threshold of 2 (i.e., wait for func2),
 * add 1 to event when func3 is complete */
doca_dpa_kernel_launch(dpa, event, 2, event, 1, …, func3, nthreads, <args>);
```

1.3.2.2. Events

The previous example demonstrates how events can be used to chain functions together for execution on the DPA. In addition to triggered scheduling, events can be directly signaled by either the CPU, GPU, DPA, or by remote nodes. This provides flexibility of coordinating work on the DPA.

The following are some use cases for DPA events:

- Signaling and waiting from CPU (host or DPU's Arm) – the CPU thread signals the event while using its counter. The event can control the execution flow on the DPA. Using the wait operation, the CPU thread can wait in either polling or blocking mode until the corresponding event is signaled.
  - CPU signals an event:
    ```
    doca_dpa_event_update(event, val)
    ```
  - CPU waits for an event:
    ```
    doca_dpa_event_wait(event, threshold)
    ```
- Signaling from the DPA from within a kernel – the event is written to in the user's kernel during its execution. When waiting, the DPA kernel thread waits until the event value satisfies the Boolean operator defined by comparator (e.g., equal, not equal, greater than).
  - DPA kernel signals an event:
    ```
    doca_dpa_dev_event_update(event, val)
    ```
  - DPA kernel waits for an event:
    ```
    doca_dpa_dev_event_wait_until(event, val, comparator)
    ```

- Signaling from remote nodes – the event is written by the remote side after its write operation (`put`) completes. This means that a remote node writes some data in the target and updates a signal at the target when the contents of its write are visible. This allows the target to schedule work which depends on the incoming remote write.
  - Remote node signals an event:

    ```
    doca_dpa_dev_put_signal(ep, <send buffer>, <recv buffer>, event, count);
    ```

1.3.2.2.1. Event Usage Example

The following example demonstrates how to construct a pipeline of functions on the DPA using events:

```
/* Host */
main()
{
      // create event for usage on DPA
      doca_dpa_event_create(ctx, &event, {setter=DPA,
                                         waiter=DPA});
      // export a handle representing the event for DPA to use
      doca_dpa_event_handle_export(event, &event_handle);
}

/* DPA: func1 -> func2 */
__dpa_global__ func1(args)
{
      work1();
      // signal next thread by adding to event counter
      doca_dpa_dev_event_update(event_handle, 1, ADD);
}

__dpa_global__ func2(args)
{
     // wait for event counter to reach 1
     doca_dpa_dev_event_wait_until(event_handle, 1,
                                  CMP_EQ);
     work2();
     // signal next thread by adding to event counter
     doca_dpa_dev_event_udpate(event, 1, ADD);
}
```

1.3.2.3. Memory Management

The DPA program can access several memory spaces using the provided APIs. The following presents models to access DPA process heap, host memory, GPU memory, and NIC device memory:

- DPA process heap – this is the DPA process' main memory. The memory may either be in the stack or on the heap. Heap allocations must be obtained using the `doca_dpa_mem_alloc()` API. The low-level memory model in this space is determined by the processor architecture.
- Host memory – this is the address space of the host program. Any memory accessed by the DPA must be registered using `doca_dpa_mem_host_register()`. DMA access to this space is provided from the DPA using the `doca_dpa_dev_memcpy()` routine.

1.3.2.4. Communication APIs

Note: Communication APIs are currently implemented for InfiniBand only.

The communication APIs enable the application to initiate and be a target of remote memory operations. The communication APIs are modeled on the UCX's UCP APIs and are implemented over RDMA transport on InfiniBand.

All communications are initiated on an endpoint (EP). An EP is an opaque representation of a queue pair (QP). EPs can be either Reliable Connected or Reliable Dynamic Connected Transport. EPs are created on the host-side of the application and then a handle to the EP can be passed to the DPA-side of the application.

The following code demonstrates a ping-pong operation between two processes, each with their code offloaded to a DPA. The program uses remote memory access semantic to transfer host memory from one node to the other. The DPA initiates the data transfer and detects its completion.

```
/* Host */
main()
{
   // Create a worker instance
   // A worker is a container for endpoints, completion queue
   // and thread required to poll the completion queue
   doca_dpa_worker_create(ctx, &worker, …);
   // Get the worker's address to pass to the remote side
   doca_dpa_worker_address(worker, &address);
   // Application does out-of-band passing of address to remote side
   // Assume remote worker's address is now in `rem_work_addr`.
   // Create an endpoint for use
   doca_dpa_ep_create(worker, &endpoint, type=CONNECTED);
   // Connect my endpoint to the remote worker
   doca_dpa_ep_connect(endpoint, rem_work_addr);

   // Pass the endpoint handle to DPA
   doca_dpa_ep_handle_export(endpoint, &ep_handle);
   // Allocate local buffer in host memory
   malloc(&local_buf, size);
   // Register buffer for remote access and
   // obtain an object representing the memory
   doca_dpa_mem_register(local_buf, access_perms, &local_mem);
   // Get local key / handle for local DPA
   doca_dpa_mem_handle_export(local_mem, &local_handle);
   // Get rkey to send to remote side
   doca_dpa_mem_rkey(local_mem, &rkey);
   // Allocate an event that can be signaled by the remote side
   // to indicate that message is ready to be read
   doca_dpa_event_create(&event, {setter = REMOTE, waiter = DPA});
   // Obtain a handle to event that can be passed to the remote side
   doca_dpa_event_remote_handle_export(event, &remote_event_handle);
   // OOB Pass the remote event handle to the other side
   // OOB Pass buffer address to remote side
 …
}
/* DPA */
func()
{
    // Obtain lkey from local handle, and rkey via OOB passing
    // Write contents of local_buf to remote_buf and
    // add `1' atomically to remote_event_handle
    doca_dpa_dev_put_signal(ep_handle, local_buf, lkey,
                                  remote_buf, rkey,
                                  remote_event_handle, 1,
                                  DOCA_DPA_DEV_SIGNAL_ADD);
    // Wait for my partner (remote node) write to complete
    doca_dpa_dev_event_wait_until(local_event, 1,
                                 DOCA_DPA_DEV_SIGNAL_CMP_EQ);
    [...]
}
```

1.3.3. Memory Model

The DPA offers a coherent but weakly ordered memory model. The application is required to use fences to impose the desired memory ordering. Additionally, where applicable, the application is required to write back data for the data to be visible to NIC engines (see the coherency table later in this section).

The memory model offers "same address ordering" within a thread. This means that, if a thread writes to a memory location and subsequently reads that memory location, the read returns the contents that have previously been written.

The memory model offers 8-byte atomicity for aligned accesses to atomic datatypes. This means that all eight bytes of read and write are performed in one indivisible transaction.

Note: The DPA does not support unaligned accesses, such as accessing `N` bytes of data from an address not evenly divisible by `N`.

The DPA processes memory can be divided into the following memory spaces:

Memory Space	Definition
Heap	Memory locations within the DPA process heap. Referenced as `__DPA_HEAP` in the code.
Memory	Memory locations belonging to the DPA process (including stack, heap, BSS and data segment) except the memory-mapped IO. Referenced as `__DPA_MEMORY` in the code.
MMIO (memory-mapped I/O)	External memory outside the DPA process accessed via memory-mapped IO. Window and Outbox accesses are considered MMIO. Referenced as `__DPA_MMIO` in the code.
System	All memory locations accessible to the thread within Memory and MMIO spaces as described previously. Referenced as `__DPA_SYSTEM` in the code.

The coherency between the DPA threads and NIC engines is described in the following table:

Producer	Observer	Coherency	Comments
DPA thread	NIC engine	Not coherent	Data to be read by the NIC must be written back using the appropriate intrinsic (see below).
NIC engine	DPA thread	Coherent	Data written by the NIC is eventually visible to the DPA threads. The order in which the writes are visible to the DPA threads is influenced by the ordering configuration of the memory region (see `IBV_ACCESS_RELAXED_ORDERING`). In a typical example of the NIC writing data and generating a completion entry (CQE), it is guaranteed that when the write to the CQE is visible, the DPA thread can read the data without additional fences.
DPA thread	DPA thread	Coherent	Data written by a DPA thread is eventually visible to the other DPA threads. However, the order in which writes made by a thread are visible to other threads is not defined.

1.3.3.1. Memory Fences

Fence APIs are intended to impose memory access ordering. The fence operations are defined on the different memory spaces. See information on memory spaces under Memory Model.

The fence APIs apply ordering between the operations issued by the calling thread. As a performance note, the fence APIs also have a side effect of writing back data to the memory space used in the fence operation. However, programmers should not rely on this side effect. See the section on Cache Control for explicit cache control operations.

The fence APIs have an effect of a compiler-barrier, which means that memory accesses are not reordered around the fence API invocation by the compiler.

A fence applies between the "predecessor" and the "successor" operations. The predecessor and successor ops can be refenced using `__DPA_R`, `__DPA_W`, and `__DPA_RW` in the code.

The generic memory fence operation can operate on any memory space and any set of predecessor and successor operations. The other fence operations are provided as convenient shortcuts that are specific to the use case. It is preferable for programmers to use the shortcuts when possible.

1.3.3.1.1. Generic Fence

```
void __dpa_thread_fence(memory_space, pred_op, succ_op);
```

This fence can apply to any DPA thread memory space. Memory spaces are defined under Memory Model. The fence ensures that all operations (`pred_op`) performed by the calling thread, before the call to `__dpa_thread_fence()`, are performed and made visible to all threads in the DPA, host, NIC engines, and peer devices as occurring before all operations (`succ_op`) to the memory space after the call to `__dpa_thread_fence()`.

1.3.3.1.2. System Fence

```
void __dpa_thread_system_fence();
```

This is equivalent to calling `__dpa_thread_fence(__DPA_SYSTEM, __DPA_RW, __DPA_RW)`.

1.3.3.1.3. Outbox Fence

```
void __dpa_thread_outbox_fence(pred_op, succ_op);
```

This is equivalent to calling `__dpa_thread_fence(__DPA_MMIO, pred_op, succ_op)`.

1.3.3.1.4. Window Fence

```
void __dpa_thread_window_fence(pred_op, succ_op);
```

This is equivalent to calling `__dpa_thread_fence(__DPA_MMIO, pred_op, succ_op)`.

1.3.3.1.5. Memory Fence

```
void __dpa_thread_memory_fence(pred_op, succ_op);
```

This is equivalent to calling `__dpa_thread_fence(__DPA_MEMORY, pred_op, succ_op)`.

1.3.3.2. Cache Control

The cache control operations allow the programmer to exercise fine-grained control over data resident in the DPA's caches. The fence APIs have an effect of a compiler-barrier.

1.3.3.2.1. Window Read Contents Invalidation

```
void __dpa_thread_window_read_inv();
```

The DPA can cache data that was fetched from external memory using a window. Subsequent memory accesses to the window memory location may return the data that is already cached. In some cases, it is required by the programmer to force a read of external memory (see example under Polling Externally Set Flag). In such a situation, the window read contents cached must be dropped.

This function ensures that contents in the window memory space of the thread before the call to `__dpa_thread_window_read_inv()` are invalidated before read operations made by the calling thread after the call to `__dpa_thread_window_read_inv()`.

1.3.3.2.2. Window Writeback

```
void __dpa_thread_window_writeback();
```

Writes to external memory must be explicitly written back to be visible to external entities.

This function ensures that contents in the window space of the thread before the call to `__dpa_thread_window_writeback()` are performed and made visible to all threads in the DPA, host, NIC engines, and peer devices as occurring before any write operation after the call to `__dpa_thread_window_writeback()`.

1.3.3.2.3. Memory Writeback

```
void __dpa_thread_memory_writeback();
```

Writes to DPA memory space may need to be written back. For example, the data must be written back before the NIC engines can read it. Refer to the coherency table under Memory Fences.

This function ensures that the contents in the memory space of the thread before the call to `__dpa_thread_writeback_memory()` are performed and made visible to all threads in the DPA, host, NIC engines, and peer devices as occurring before any write operation after the call to `__dpa_thread_writeback_memory()`.

1.3.3.3. Memory Fence and Cache Control Usage Examples

These examples illustrate situations in which programmers must use fences and cache control operations.

Note: In most situations, such direct usage of fences is not required by the application using FlexIO or DOCA DPA SDKs as fences are used within the APIs.

1.3.3.3.1. Issuing Send Operation

In this example, a thread on the DPA prepares a work queue element (WQE) that is read by the NIC to perform the desired operation.The ordering requirement is to ensure the WQE data contents are visible to the NIC engines read it. The NIC only reads the WQE after the doorbell (MMIO operation) is performed. Refer to coherency table.

#	User Code – WQE Present in DPA Memory	Comment
1	Write WQE	Write to memory locations in the DPA (memory space = `__DPA_MEMORY`)

#	User Code – WQE Present in DPA Memory	Comment
2	`__dpa_thread_memory_writeback();`	Cache control operation
3	Write doorbell	MMIO operation via Outbox

In some cases, the WQE may be present in external memory. See the description of `flexio_qmem` above. The table of operations in such a case is below.

#	User Code – WQE Present in External Memory	Comment
1	Write WQE	Write to memory locations in the DPA (memory space = `__DPA_MMIO`)
2	`__dpa_thread_window_writeback();`	Cache control operation
3	Write doorbell	MMIO operation via Outbox

1.3.3.3.2. Posting Receive Operation

In this example, a thread on the DPA is writing a WQE for a receive queue and advancing the queue's producer index. The DPA thread will have to order its writes and writeback the doorbell record contents so that the NIC engine can read the contents.

#	User Code – WQE Present in DPA Memory	Comment
1	Write WQE	Write to memory locations in the DPA (memory space = `__DPA_MEMORY`)
2	`__dpa_thread_memory_fence(__DPA_W, __DPA_W);`	Order the write to the doorbell record with respect to WQE
3	Write doorbell record	Write to memory locations in the DPA (memory space = `__DPA_MEMORY`)
4	`__dpa_thread_memory_writeback();`	Ensure that contents of doorbell record are visible to the NIC engine

1.3.3.3.3. Polling Externally Set Flag

In this example, a thread on the DPA is polling on a flag that will be updated by the host or other peer device. The memory is accessed by the DPA thread via a window. The DPA thread must invalidate the contents so that the underlying hardware performs a read.

User Code – WQE Present in DPA Memory	Comment

`while (!flag) {`
`  __dpa_thread_window_read_inv();`
`}`
	`flag` is a memory location read using a window

1.3.3.3.4. Thread-to-thread Communication

In this example, a thread on the DPA is writing a data value and communicating that the data is written to another thread via a flag write. The data and flag are both in DPA memory.

User Code – Thread 1	User Code – Thread 2	Comment
		Initial condition, `flag = 0`
`var1 = x;`	`while(*((volatile int *)&flag) !=1);`	▶ Thread 1 – write to `var1` ▶ Thread 2 – `flag` is accessed as a volatile variable, so the compiler preserves the intended program order of reads
`__dpa_thread_memory_fence(__DPA_W, __DPA_W);`		Write to flag cannot bypass write to `var1`
	`var_t2 = var1;`	
`flag = 1;`	`assert(var_t2 == x);`	`var_t2` must be equal to `x`

1.3.3.3.5. Setting Flag to be Read Externally

In this example, a thread on the DPA sets a flag that is observed by a peer device. The flag is written using a window.

User Code – Thread 1	Comment
`flag = data;`	`flag` is updated in local DPA memory
`__dpa_thread_window_writeback();`	Contents from DPA memory for the window are written to external memory

1.3.4. DPA-specific Operations

The DPA supports some platform-specific operations. These can be accessed using the functions described in the following subsections.

1.3.4.1. Clock Cycles

```
uint64_t __dpa_thread_cycles();
```

Returns a counter containing the number of cycles from an arbitrary start point in the past on the execution unit the thread is currently scheduled on.

Note that the value returned by this function in the thread is meaningful only for the duration of when the thread remains associated with this execution unit.

This function also acts as a compiler barrier, preventing the compiler from moving instructions around the location where it is used.

1.3.4.2. Timer Ticks

```
uint64_t __dpa_thread_time();
```

Returns the number of timer ticks from an arbitrary start point in the past on the execution unit the thread is currently scheduled on.

Note that the value returned by this function in the thread is meaningful only for the duration of when the thread remains associated with this execution unit.

This intrinsic also acts as a compiler barrier, preventing the compiler from moving instructions around the location where the intrinsic is used.

1.3.4.3. Instructions Retired

```
uint64_t __dpa_thread_inst_ret();
```

Returns a counter containing the number of instructions retired from an arbitrary start point in the past by the execution unit the thread is currently scheduled on.

Note that the value returned by this function in the software thread is meaningful only for the duration of when the thread remains associated with this execution unit.

This intrinsic also acts as a compiler barrier, preventing the compiler from moving instructions around the location where the intrinsic is used.

1.3.4.4. Fixed Point Log2

```
int __dpa_fxp_log2(unsigned int);
```

This function evaluates the fixed point Q16.16 base 2 logarithm. The input is an unsigned integer.

1.3.4.5. Fixed Point Reciprocal

```
int __dpa_fxp_rcp(int);
```

This function evaluates the fixed point Q16.16 reciprocal (1/x) of the value provided.

1.3.4.6. Fixed Point Pow2

```
int __dpa_fxp_pow2(int);
```

This function evaluates the fixed point Q16.16 power of 2 of the provided value.

Chapter 2. Programming FlexIO SDK

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.

After DOCA installation, FlexIO SDK headers may be found under `/opt/mellanox/flexio/include` and libraries may be found under `/opt/mellanox/flexio/lib/`.

2.1. 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.

2.2. Architecture

FlexIO SDK library exposes a few layers of functionality:

- `libflexio` – library for DPU-side operations. It is used for resource management.
- `libflexio_dev` – library for DPA-side operations. It is used for data path implementation.
- `libflexio_os` – library for DPA OS-level access
- `libflexio_libc` – a lightweight C library for DPA device code

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.

2.3. API

Please refer to the NVIDIA DOCA Driver APIs Reference Manual.

2.4. 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.

2.4.1. 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;
      ...
}
```

2.5. 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);
```

2.5.1. Allocating NIC Queues for Use by DPA

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 work queue elements (WQEs). 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).

2.5.2. Memory Allocation Best Practices

To optimize process device 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 device memory 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.

2.6. DPA Window

DPA windows are used to access external memory, such as on the DPU's DDR or host's memory. DPA windows are the software mechanism to use the Memory Apertures mentioned in section DPA Memory and Caches. To use the window functionality, DPU or host memory must be registered for the device using the `ibv_reg_mr()` call.

Note: Both the address and size provided to this call must be 64 bytes aligned for the window to operate. This alignment may be obtained using the `posix_memalign()` allocation call.

2.7. DPA Event Handler

2.7.1. 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.

2.7.2. HART Management

DPA HARTs are the equivalent to logical cores. For a DPA program to execute, it must be assigned a HART.

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`, `strict`, and `group`.

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.

Note: For more information, please refer to NVIDIA DOCA DPA HART Management Tool.

2.7.3. HART Partitions

To work over DPA, a HART partition must be created for the used device. A partition is a selection of HARTs marked as available for a device. For the DPU ECPF, a default partition is created upon boot with all HARTs available in it. For any other device (i.e., function), the user must create a partition. This means that running an application on a non-ECPF function without creating a partition would result in failure.

WARNING: FlexIO SDK uses `strict` and `none` affinity for internal threads, which require a partition with at least one HART for the participating devices. Failing to comply with this assumption may cause failures.

2.7.4. Virtual HARTs

Users should be aware that beside the default HART partition, which is exposed to the real HART numbers, all other partitions created use virtual HARTs.

For example, if a user creates a partition with the range of HARTs 20-40, querying the partition info from one of its virtual HCAs it would display HARTs from 0-20. So the real HART number 39 in this example would correspond to the virtual HART number 19.

2.8. 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.

2.8.1. 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.

WARNING: `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.

2.8.1.1. 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`, `%lo`, `%p`, `%%`
- Flags – `.`, `*`, `-`, `+`, `#`
- General supported modifiers:
  - "0" padding
  - Min/max characters in string
- General unsupported modifiers:
  - Floating point modifiers – `%e`, `%E`, `%f`, `%lf`, `%LF`
  - Octal modifier `%o` is partially supported
  - Precision modifiers

2.8.2. Core Dump

If the DPA process encounters a fatal error, the user can create a core dump file to review the application's status at that point using a GDB app.

Creating a core dump file can be done after the process has crashed (as indicated by the `flexio_err_status` API) and before the process is destroyed by calling the `flexio_coredump_create` API.

Recommendations for opening DPA core dump file using GDB:

- Use the `gdb-multiarch` application
- The `Program` parameter for GDB should be the device-side ELF file
  - Use the `dpacc-extract` tool (provided with the DPACC package) to extract the device-side ELF file from the application's ELF file

2.9. FlexIO Samples

This section describes samples based on the FlexIO SDK. These samples illustrate how to use the FlexIO API to configure and execute code on the DPA.

2.9.1. Running FlexIO 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/<library_name>/<sample_name>
   meson build
   ninja -C build
   ```
   Note: The binary `flexio_<sample_name>` will be created under `./build/host/`.
3. Sample (e.g., `flexio_rpc`) usage:
   ```
   Usage: flexio_rpc [DOCA 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>
   ```
   For additional information per sample, use the `-h` option:
   ```
   ./build/host/flexio_<sample_name> -h
   ```

2.9.2. Samples

2.9.2.1. FlexIO RPC

This sample illustrates how to invoke a function on the DPA.

The sample logic includes:

1. Creating FlexIO process.
2. Calling the remote function `flexio_rpc_calculate_sum` on the DPA.
3. Printing return value to the standard output.

References:

- `/opt/mellanox/doca/samples/flexio/flexio_rpc/device/flexio_rpc_device.c`
- `/opt/mellanox/doca/samples/flexio/flexio_rpc/host/flexio_rpc_sample.c`
- `/opt/mellanox/doca/samples/flexio/flexio_rpc/host/meson.build`
- `/opt/mellanox/doca/samples/flexio/flexio_rpc/flexio_rpc_main.c`
- `/opt/mellanox/doca/samples/flexio/flexio_rpc/meson.build`

2.9.2.2. FlexIO Window

This sample illustrates how to use the FlexIO Window API to access host memory from the DPA device.

The sample logic includes:

1. Creating FlexIO process and FlexIO Window.
2. Registering host buffer with FlexIO Window.
3. Calling remote function on the device which overwrites the host buffer.
4. Printing the altered host buffer.

References:

- `/opt/mellanox/doca/samples/flexio/flexio_window/device/flexio_window_device.c`
- `/opt/mellanox/doca/samples/flexio/flexio_window/host/flexio_window_sample.c`
- `/opt/mellanox/doca/samples/flexio/flexio_window/host/meson.build`
- `/opt/mellanox/doca/samples/flexio/flexio_window/flexio_window_common.h`
- `/opt/mellanox/doca/samples/flexio/flexio_window/flexio_window_main.c`
- `/opt/mellanox/doca/samples/flexio/flexio_window/meson.build`

2.9.2.3. FlexIO Multithread

This samples illustrates how to use the FlexIO command queue (`flexio_cmdq_*`) API to run a multithread on the DPA device.

The sample logic includes:

1. Creating FlexIO Process, Window, cmdq, and other FlexIO resources.
2. Copying two matrixes to be multiplied to the device (each matrix is 5*5).
3. Allocating result matrix on the host for the results. Each DPA thread writes to this matrix.
4. Generating 5*5 jobs and submitting in the command queue. Each job is responsible for one cell calculation.
5. Starting the command queue.
6. Printing the result matrix.

References:

- `/opt/mellanox/doca/samples/flexio/flexio_multithread/device/flexio_multithread_device.c`
- `/opt/mellanox/doca/samples/flexio/flexio_multithread/host/flexio_multithread_sample.c`
- `/opt/mellanox/doca/samples/flexio/flexio_multithread/host/meson.build`
- `/opt/mellanox/doca/samples/flexio/flexio_multithread/flexio_multithread_common.h`
- `/opt/mellanox/doca/samples/flexio/flexio_multithread/flexio_multithread_main.c`
- `/opt/mellanox/doca/samples/flexio/flexio_multithread/meson.build`

Chapter 3. Programming DOCA DPA

Note: Supported at alpha level.

This chapter provides an overview and configuration instructions for DOCA DPA API.

The DOCA DPA library offers a programming model for offloading communication-centric user code to run on the DPA processor on NVIDIA® BlueField®-3 DPU. DOCA DPA provides a high-level programming interface to the DPA processor.

DOCA DPA enables the user to submit a function (called a kernel) that runs on the DPA processor. The user can choose to run the kernel with multiple threads.

DOCA DPA offers:

- Support for the DPU subsystem, an offload engine that executes user-written kernels on a highly multi-threaded processor
- Full control on execution-ordering and notifications/synchronization of the DPA and host/DPU
- Abstractions for memory services
- Abstractions for remote communication primitives (integrated with remote event signaling)
- C API for application developers

DPACC is used to compile and link kernels with the DOCA DPA device library to get DPA applications that can be loaded from the host program to execute on the DPA (similar to CUDA usage with NVCC). For more information on DPACC, refer to NVIDIA DOCA DPACC Compiler User Guide.

3.1. API

Please refer to the NVIDIA DOCA Libraries API Reference Manual.

3.2. Development Flow

DOCA enables developers to program the DPA processor using both DOCA DPA library and a suite of other tools (mainly DPACC).

These are the main steps to start DPA offload programming:

1. Write DPA device code, or kernels, (`.c` files) with `__dpa_global__` macro statement before function (see examples later).
2. Use DPACC to build a DPA program (i.e., a host library which contains an embedded device executable). Input for DPACC are kernels from the previous steps and DOCA DPA device library.
3. Build host executable using a host compiler (inputs for the host compiler are DPA program are generated in the previous step and the user application source files).
4. In your program, create a DPA context which then downloads the DPA executable to the DPA processor to allow invoking (launching) kernels from the host/DPU to run on the DPA.

DPACC is provided by the DOCA SDK installation. For more information, please refer to the NVIDIA DOCA DPACC Compiler User Guide.

3.3. Software Architecture

3.3.1. Deployment View

DOCA DPA is composed of two libraries that come with the DOCA SDK installation:

- Host/DPU library and header file (used by user application)
  - `doca_dpa.h`
  - `libdoca_dpa.a`/`libdoca_dpa.so`
- Device library and header file
  - `doca_dpa_dev.h`
  - `libdoca_dpa_dev.a`

3.3.2. DPA Queries

Before invoking the DPA API, make sure that DPA is indeed supported on the relevant device.

The API that checks whether a device supports DPA is `doca_error_t doca_devinfo_get_is_dpa_supported(const struct doca_devinfo *devinfo)`. Only if this call returns `DOCA_SUCCESS` can the user invoke DOCA DPA API on the device.

There is a limitation on the maximum number of DPA threads that can run a single kernel. This can be retrieved by calling the host API `doca_error_t doca_dpa_get_max_threads_per_kernel(doca_dpa_t dpa, unsigned int *value)`.

Each kernel launched into the DPA has a maximum runtime limit. This can be retrieved by calling the host API `doca_error_t doca_dpa_get_kernel_max_run_time(doca_dpa_t dpa, unsigned long long *value)`.

Important: If the kernel execution time on the DPA exceeds this maximum runtime limit, it may be terminated and cause a fatal error. To recover, the application must destroy the DPA context and create a new one.

3.3.3. Overview of DOCA DPA Software Objects

Term	Definition
DPA context	Software construct for the host process that encapsulates the state associated with a DPA process (on a specific device). An application must obtain a DPA context to begin working with the DPA API (several DPA contexts may be created by the application).
Memory	DOCA DPA provides an API to allocate/manage DPA memory and map host memory to the DPA.
Event	Data structure in either CPU/DPU or DPA-heap. An event contains a counter that can be updated and waited on.
Kernel	User function (and arguments) launched from host and executing on the DPA asynchronously. A kernel may be executed by one or more DPA threads (specified by the application as part of the launch API).
Endpoint	Abstraction around a network transport object (encapsulates RDMA RC QP). Allows sending data to remote EP for example.
Worker	Container for communication resources, such as endpoints, completion queues and threads that collect communication completions.

Important: The DOCA DPA SDK does not use any means of multi-thread synchronization primitives. All DOCA DPA objects are non-thread-safe. Developers should make sure the user program and kernels are written so as to avoid race conditions.

3.3.4. DPA Context

Context creation:

```
doca_error_t doca_dpa_create(struct doca_dev *dev, doca_dpa_app_t app, doca_dpa_t
 *dpa, unsigned int flags)
```

The DPA context encapsulates the DPA device and a DPA process (program). Within this context, the application creates various DPA SDK objects and controls them. After verifying DPA is supported for the chosen device, the DPA context is created. This is a host-side API and it is expected to be the first programming step.

The `app` parameter of this call is the same name used when running DPACC to get the DPA program.

3.3.5. Memory Subsystem

A DPA program can allocate (from the host API) and access (from both the host and device API) several memory locations using the DOCA DPA API. DOCA DPA supports access from the host/DPU to DPA heap memory and also enables device access to host memory (e.g., kernel writes to host heap).

Normal memory usage flow would be to:

1. Allocate memory on the host.
2. Register the memory.
3. Export the registered memory so it can be accessed by DPA kernels.
4. Access/use the memory from the kernel (see device side APIs later).

3.3.5.1. Host-side API

- To free previously-allocated DPA memory (if it exists):

  ```
  doca_dpa_mem_free(doca_dpa_dev_uintptr_t dev_ptr)
  ```

- To allocate DPA heap memory:

  ```
  doca_dpa_mem_alloc(doca_dpa_t dpa, size_t size, doca_dpa_dev_uintptr_t *dev_ptr)
  ```

- To copy previously-allocated memory:

  ```
  doca_dpa_h2d_memcpy(doca_dpa_t dpa, doca_dpa_dev_uintptr_t dev_ptr, void
   *src_ptr, size_t size)
  ```

- To set memory:

  ```
  doca_dpa_memset(doca_dpa_t dpa, doca_dpa_dev_uintptr_t dev_ptr, int value, size_t
   size)
  ```

- To register host-resident memories to the hardware, enabling the hardware to access them later:

  ```
   doca_dpa_mem_host_register(doca_dpa_t dpa, void *addr, size_t
   length, unsigned int access, doca_dpa_mem_t *mem, unsigned int flags)
  ```

- To unregister the memory:

  ```
  doca_dpa_mem_unregister(doca_dpa_mem_t mem)
  ```

- To export the memory to DPA to get a handle to use on your kernels:

  ```
  doca_dpa_mem_dev_export(doca_dpa_mem_tmem, doca_dpa_dev_mem_t*handle)
  ```

Note: Use the output parameter (`handle`) by passing it as kernel parameter in `doca_dpa_kernel_launch(...)` API.

3.3.5.2. Device-side API

Device APIs are used by user-written kernels. Memory APIs supplied by the DOCA DPA SDK are all asynchronous (i.e., non-blocking).

Kernels get the `doca_dpa_dev_mem_t` handle in their kernel and invoke the following API:

- To get rkey of exported memory:
  ```
  doca_dpa_dev_mem_rkey_get(doca_dpa_dev_mem_t mem)
  ```
- To copy memory between two registered memory regions:
  ```
  doca_dpa_dev_memcpy_nb(uint64_t dest_addr, doca_dpa_dev_mem_t dest_mem, uint64_t
   src_addr, doca_dpa_dev_mem_t src_mem, size_t length)
  ```
- To transpose memory between two registered memory regions:
  ```
  doca_dpa_dev_memcpy_transpose2D_nb(uint64_t dest_addr, doca_dpa_dev_mem_t
   dest_mem, uint64_t src_addr, doca_dpa_dev_mem_t src_mem, size_t length, size_t
   element_size, size_t num_columns, size_t num_rows)
  ```
- To get a pointer to external memory registered on the host:
  ```
  doca_dpa_dev_external_ptr_get(uint64_t ext_addr, doca_dpa_dev_mem_t mem,
   doca_dpa_dev_uintptr_t *dev_ptr)
  ```
- Since memory operations in the device are non-blocking, developers are given the following API to drain all previous memory operations:
  ```
  doca_dpa_dev_memcpy_synchronize()
  ```

3.3.6. DPA Events

DPA events fulfill the following roles:

- DOCA DPA execution model is asynchronous and events are used to control various threads running in the system (allowing order and dependency)
- DOCA DPA supports remote events so the programmer is capable of invoking remote nodes by means of DOCA DPA events

Events encapsulate a memory (counter) located in either the DPA or host memory. A DPA event can be waited to reach a certain threshold (waiting a thread does busy-loop or yield on the memory location) or updated to a new value (either be atomically adding a value to the current one or setting it) by another thread.

To optimize event operation, DOCA DPA allocates the event memory in the most optimal location (based on hints from the programmer). For example, DOCA DPA may create the event in a location that allows it direct memory access instead of RDMA MMO.

Event creation:

```
doca_error_t doca_dpa_event_create(doca_dpa_t dpa, unsigned int
 update_location, unsigned int wait_location, unsigned int wait_method,
 doca_dpa_event_t *event, unsigned int flags)
```

A DPA event is created on the host and, as part of its creation, DOCA DPA receives hints to the locations of the updater and waiter. If the event waiter is CPU, then the event memory is in CPU memory. If the event waiter is DPA, then the event memory is in DPA memory.

3.3.6.1. Host-side API

- To create an event:
  ```
  doca_dpa_event_create(doca_dpa_t dpa, unsigned int update_location, unsigned int
   wait_location, unsigned int wait_method, doca_dpa_event_t *event, unsigned int
   flags)
  ```
- To destroy an event:
  ```
  doca_dpa_event_destroy(doca_dpa_event_t event)
  ```

- To export an event to a device or remote device (to get a handle to it which can later be used by kernels to signal to the host, for example):
    - Export event to device:
      ```
      doca_dpa_event_dev_export(doca_dpa_event_t event, doca_dpa_dev_event_t
       *handle)
      ```
    - Export event to remote device:
      ```
      doca_dpa_event_dev_remote_export(doca_dpa_event_t event,
       doca_dpa_dev_event_remote_t *handle)
      ```

3.3.6.2. Device-side API

- To get the current event value:
  ```
  doca_dpa_event_get(doca_dpa_event_t event, uint64_t *count)
  ```
- To set/add to the current event value:
  ```
  doca_dpa_event_update(doca_dpa_event_t event, uint64_t val, enum
   doca_dpa_event_op op)
  ```
- To wait until event reaches threshold:
  ```
  doca_dpa_event_wait_until(doca_dpa_event_t ev, uint64_t threshold, enum
   doca_dpa_event_cmp cmp)
  ```

3.3.7. Communication Model

DOCA DPA communication primitives allow sending data from one node to another. The communication object between two nodes is called an endpoint (EP). EPs represent a unidirectional communication pipe between two nodes. Currently, EP are built on top of RDMA RC QP implementation. EPs can be used by kernels only. That is, there is no communication from one host to another (the way to get around that is to have one kernel send data to another node's kernel).

Another entity involved in DOCA DPA communication scheme is the DPA worker which performs communication progress on EPs associated with it. The worker is an abstraction around the completion queue monitored by an internal DPA thread. Many EPs can be mapped to the same worker. It is recommended to have only one worker created to track all EP progress on that node/process. The worker allows the DPA to track the completion of all communications since they are asynchronous.

3.3.7.1. Host-side API

- To create a DPA worker:
  ```
  doca_dpa_worker_create(doca_dpa_t dpa, doca_dpa_worker_t *worker, unsigned int
   flags)
  ```
- To create a DPA EP:
  ```
  doca_dpa_ep_create(doca_dpa_worker_t worker, unsigned int access, doca_dpa_ep_t
   *ep)
  ```
- To destroy a DPA EP:
  ```
  doca_dpa_ep_destroy(doca_dpa_ep_t ep)
  ```
- To get the address of an EP (so a remote device can connect to it):
  ```
  doca_dpa_ep_addr_get(doca_dpa_ep_t ep, doca_dpa_ep_addr_t *addr, size_t
   *addr_length)
  ```

- To free up the address of a remote EP:

```
doca_dpa_ep_addr_free(doca_dpa_ep_addr_t addr)
```

- To connect to a remote EP address:

```
doca_dpa_ep_connect(doca_dpa_ep_t ep, doca_dpa_ep_addr_t addr)
```

- To export an endpoint to the device:

```
doca_dpa_ep_dev_export(doca_dpa_ep_t ep, doca_dpa_dev_ep_t *handle)
```

WARNING: Endpoints are not thread-safe and, therefore, EP must not be used from different kernels/threads concurrently.

3.3.7.2. Device-side API

The flow in DOCA DPA begins with creation and configuration of an object on the host/ DPU side and then exporting it to a device so the kernels and DPA can use various entities. This is no different for endpoints. After the host application connects two EPs, the local EP is exported to the device and can be used by the device-side API.

- To copy local memory to the remote side:

```
doca_dpa_dev_put_nb(doca_dpa_dev_ep_t ep, uint64_t local_addr, size_t length,
 doca_dpa_dev_mem_t local_mem, uint64_t raddr, uint32_t rkey)
```

- To copy local memory to the remote side and update the remote event:

```
doca_dpa_dev_put_signal_nb(doca_dpa_dev_ep_t ep, uint64_t local_addr,
 doca_dpa_dev_mem_t local_mem, size_t length, uint64_t raddr, uint32_t rkey,
 doca_dpa_dev_event_remote_t event, uint64_t comp_count, enum doca_dpa_event_op
 comp_op)
```

The above API encapsulates two operations in a single call:

- An RDMA write operation between two connected endpoints
- A signal remote event which can, in theory, notify a remove thread waiting on the remote event that the write operation has completed

As all EP operations are non-blocking, the following API is provided to kernel developers to wait until all previous EP operations are done (blocking call) to drain the EP:

```
doca_dpa_dev_ep_synchronize(doca_dpa_dev_ep_t ep)
```

When this call returns, all previous non-blocking operations on the EP have completed (i.e., sent to the remote EP). It is expected that the `doca_dpa_dev_ep_synchronize()` call would use the same thread as the `doca_dpa_dev_put_nb()` or `doca_dpa_dev_put_signal_nb()` calls.

Note: Since EPs are non-thread safe, each EP must be accessed by a single thread at any given time. If user launches a kernel that should be executed by more than one thread and this kernel includes EP communication, it is expected that a user will use array of EPs so that each EP will be accessed by single thread (each thread can access it's EP instance by using `doca_dpa_dev_thread_rank()` as its index in the array of endpoints).

3.3.7.3. Typical Endpoint Flow

1. Create a worker on the local and remote nodes:

```
doca_dpa_worker_create(worker)
```

2. Create an EP on each node:
   `doca_dpa_ep_create(worker, …, ep)`
3. Connect the EPs:
   a). Get the remote EP's address:
   `doca_dpa_ep_addr_get(remote_ep_addr)`
   b). Pass the above address to the local machine (out of DOCA scope) by using sockets, for example.
   c). Connect the two EPs:
   `doca_dpa_ep_connect(ep, remote_ep_addr)`
4. Export the local EP to the local device:
   `doca_dpa_ep_dev_export(ep)`

   User can now use the EP handle in the kernel to send data to the remote EP (`doca_dpa_dev_put_nb()` and `doca_dpa_dev_put_signal_nb()`).

   Note: When using the Remote Event Exchange API, `void doca_dpa_dev_put_signal_nb(..., doca_dpa_dev_event_remote_t event_handle, ...)`, within your kernel, note that `event` is a remote event. That is, an event created on the remote node and exported to a remote node (`doca_dpa_event_dev_remote_export(event_handle)`).

3.3.7.4. Limitations

- DOCA DPA has only been enabled and tested with InfiniBand networking protocol.
- EPs are not thread-safe.

3.3.8. Execution Model

DOCA DPA provides an API which enables full control for launching and monitoring kernels.

Understanding the following terms and concepts is important:

- Thread – just as with modern operating systems, DOCA DPA provides a notion of sequential execution of programmer code in an "isolated" environment (i.e., variables which are thread-local and not global or shared between other threads). Moreover, DOCA DPA provides hardware threads which allow a dedicated execution pipe per thread without having the execution of one thread impact other ones (no preemption, priorities etc).
- Kernel – this is a C function written by the programmer and compiled with DPACC that the programmer can invoke from their host application. Programmers can use the `doca_dpa_kernel_launch(...)` API to run their kernels. One of the parameters for this call is the number of threads to run this kernel (minimum is 1). So, for example, a programmer can launch their kernel and ask to run it with 16 threads.

Note: Since DOCA DPA libraries are not thread-safe, it is up to the programmer to make sure the kernel is written to allow it to run in a multi-threaded environment. For example, to program a kernel that uses EPs with 16 concurrent threads, the user should pass an array of 16 EPs to the kernel so that each thread can access its EP using its rank (`doca_dpa_dev_thread_rank()`) as an index to the array.

3.3.8.1. Host-side API

```
doca_dpa_kernel_launch(doca_dpa_t dpa, doca_dpa_event_t wait_event, uint64_t
 wait_threshold, doca_dpa_event_t comp_event, uint64_t comp_count, enum
 doca_dpa_event_op comp_op, doca_dpa_func_t *func, unsigned int nthreads, ... /*
 args */)
```

- This function asks DOCA DPA to run `func` in DPA by `nthreads` and give it the supplied list of arguments (variadic list of arguments)
- This function is asynchronous so when it returns, it does not mean that `func` started/ended its execution
- To add control or flow/ordering to these asynchronous kernels, two optional parameters for launching kernels are available:
  - `wait_event` – the kernel does not start its execution until the event is signaled (if NULL, the kernel starts once DOCA DPA has resources to run it) which means that DOCA DPA would not run the kernel until the event's counter is bigger than `wait_threshold`
  - `comp_event` – once the last thread running the kernel is done, DOCA DPA updates this event (either sets or adds to its current counter value with `comp_count`). Programmers may choose whether to set the event using `enum doca_dpa_event_op`.
- DOCA DPA takes care of packing (on host/DPU) and unpacking (in DPA) the kernel parameters
- `func` must be prefixed with the `__dpa_global__` macro for DPACC to compile it as a kernel (and add it to DPA executable binary) and not as part of host application binary
- The programmer must declare `func` in their application also by adding the line `extern doca_dpa_func_t func`

3.3.8.2. Device-side API

- `unsigned int doca_dpa_dev_thread_rank()`
  Retrieves the running thread's rank for a given kernel on the DPA. If, for example, a kernel is launched to run with 16 threads, each thread running this kernel is assigned a rank ranging from 0 to 15 within this kernel. This is helpful for making sure each thread in the kernel only accesses data relevant for its execution to avoid data-races.
- `unsigned int doca_dpa_dev_num_threads()`
  Returns the number of threads running current kernel.

3.3.8.3. Examples

3.3.8.3.1. Linear Execution Example

Kernel code:

```
#include "doca_dpa_dev.h"

__dpa_global__ void
```

```
linear_kernel(doca_dpa_dev_event_t wait_ev, doca_dpa_dev_event_t comp_ev, enum
 doca_dpa_event_op comp_op)
{
    if (wait_ev)
        doca_dpa_dev_event_wait_until(wait_ev, wait_th = 1, DOCA_DPA_EVENT_CMP_EQ);

    doca_dpa_dev_event_update(comp_ev, comp_count = 1, comp_op);
}
```

Application pseudo-code:

```
#include <doca_dev.h>
#include <doca_error.h>
#include <doca_dpa.h>

int main(int argc, char **argv)
{
    /*
        A
        |
        B
        |
        C
    */

    /* Open DOCA device */
    open_doca_dev(&doca_dev);
    /* Create doca dpa conext */
    doca_dpa_create(doca_dev, dpa_linear_app, &dpa_ctx, 0);

    /* Create root event A that will signal from the host the rest to start */
    doca_dpa_event_create(updater = CPU, waiter = DPA, ev_a);
    /* Create events B,C */
    doca_dpa_event_create(updater = DPA, waiter = DPA, ev_b);
    doca_dpa_event_create(updater = DPA, waiter = DPA, ev_c);
    /* Create completion event for last kernel */
    doca_dpa_event_create(updater = DPA, waiter = CPU, comp_ev);

    /* Export kernel events and acquire their handles */
    doca_dpa_event_dev_export(&ev_b_handle, &ev_c_handle, &ev_d_handle,
 &ev_e_handle, &comp_ev_handle);

    /* launch linear kernels */
    doca_dpa_kernel_launch(wait_ev = ev_a, wait_threshold = 1, &linear_kernel,
 nthreads = 1, kernel_args:
                          NULL, ev_b_handle, DOCA_DPA_EVENT_OP_ADD);
    doca_dpa_kernel_launch(wait_ev = NULL, &linear_kernel, nthreads = 1,
 kernel_args:
                          ev_b_handle, ev_c_handle, DOCA_DPA_EVENT_OP_ADD);
    doca_dpa_kernel_launch(wait_ev = NULL, &linear_kernel, nthreads = 1,
 kernel_args:
                          ev_c_handle, comp_ev_handle, DOCA_DPA_EVENT_OP_ADD);

    /* Update host event to trigger kernels to start executing in a linear manner */
    doca_dpa_event_update(ev_a, 1, DOCA_DPA_EVENT_OP_SET);
    /* Wait for completion of last kernel */
    doca_dpa_event_wait_until(comp_ev, 1, DOCA_DPA_EVENT_CMP_EQ);

    /* Tear Down... */
    teardown_resources();
}
```

3.3.8.3.2. Diamond Execution Example

Kernel code:

```
#include "doca_dpa_dev.h"
```

```
__dpa_global__ void
diamond_kernel(doca_dpa_dev_event_t wait_ev, uint64_t wait_th, doca_dpa_dev_event_t
 comp_ev1, doca_dpa_dev_event_t comp_ev2, enum doca_dpa_event_op comp_op)
{
    if (wait_ev)
        doca_dpa_dev_event_wait_until(wait_ev, wait_th, DOCA_DPA_EVENT_CMP_EQ);

    doca_dpa_dev_event_update(comp_ev1, comp_count = 1, comp_op);
    if (comp_ev2)    // can be 0 (NULL)
        doca_dpa_dev_event_update(comp_ev2, comp_count = 1, comp_op);
}
```

Application pseudo-code:

```
#include <doca_dev.h>
#include <doca_error.h>
#include <doca_dpa.h>

int main(int argc, char **argv)
{
    /*
         A
        / \
       C   B
      /   /
     D   /
      \ /
       E
    */

    /* Open DOCA device */
    open_doca_dev(&doca_dev);
    /* Create doca dpa conext */
    doca_dpa_create(doca_dev, dpa_diamond_app, &dpa_ctx, 0);

    /* Create root event A that will signal from the host the rest to start */
    doca_dpa_event_create(updater = CPU, waiter = DPA, ev_a);
    /* Create events B,C,D,E */
    doca_dpa_event_create(updater = DPA, waiter = DPA, ev_b);
    doca_dpa_event_create(updater = DPA, waiter = DPA, ev_c);
    doca_dpa_event_create(updater = DPA, waiter = DPA, ev_d);
    doca_dpa_event_create(updater = DPA, waiter = DPA, ev_e);
    /* Create completion event for last kernel */
    doca_dpa_event_create(updater = DPA, waiter = CPU, comp_ev);

    /* Export kernel events and acquire their handles */
    doca_dpa_event_dev_export(&ev_b_handle, &ev_c_handle, &ev_d_handle,
 &ev_e_handle, &comp_ev_handle);

    /* wait threshold for each kernel is the number of parent nodes */
    constexpr uint64_t wait_threshold_one_parent {1};
    constexpr uint64_t wait_threshold_two_parent {2};

    /* launch diamond kernels */
    doca_dpa_kernel_launch(wait_ev = ev_a, wait_threshold = 1, &diamond_kernel,
 nthreads = 1, kernel_args:
                          NULL, 0, ev_b_handle, ev_c_handle,
 DOCA_DPA_EVENT_OP_ADD);
    doca_dpa_kernel_launch(wait_ev = NULL, &diamond_kernel, nthreads = 1,
 kernel_args:
                          ev_b_handle, wait_threshold_one_parent, ev_e_handle,
 NULL, DOCA_DPA_EVENT_OP_ADD);
    doca_dpa_kernel_launch(wait_ev = NULL, &diamond_kernel, nthreads = 1,
 kernel_args:
                          ev_c_handle, wait_threshold_one_parent, ev_d_handle,
 NULL, DOCA_DPA_EVENT_OP_ADD);
    doca_dpa_kernel_launch(wait_ev = NULL, &diamond_kernel, nthreads = 1,
 kernel_args:
```

```
                              ev_d_handle, wait_threshold_one_parent, ev_e_handle,
NULL, DOCA_DPA_EVENT_OP_ADD);
    doca_dpa_kernel_launch(wait_ev = NULL, &diamond_kernel, nthreads = 1,
kernel_args:
                              ev_e_handle, wait_threshold_two_parent, comp_ev_handle,
NULL, DOCA_DPA_EVENT_OP_ADD);

    /* Update host event to trigger kernels to start executing in a diamond manner
*/
    doca_dpa_event_update(ev_a, 1, DOCA_DPA_EVENT_OP_SET);
    /* Wait for completion of last kernel */
    doca_dpa_event_wait_until(comp_ev, 1, DOCA_DPA_EVENT_CMP_EQ);

    /* Tear Down... */
    teardown_resources();
}
```

3.3.8.4. Performance Optimizations

- The time interval between a kernel launch call from the host and the start of its execution on the DPA is significantly optimized when the host application calls `doca_dpa_kernel_launch()` repeatedly to execute with the same number of DPA threads. So, if the application calls `doca_dpa_kernel_launch(..., nthreads = x)`, the next call with `nthreads = x` would have a shorter latency (as low as ~5-7 microseconds) for the start of the kernel's execution.
- Applications calling for kernel launch with a wait event (i.e., the completion event of a previous kernel) also have significantly lower latency in the time between the host launching the kernel and the start of the execution of the kernel on the DPA. So, if the application calls `doca_dpa_kernel_launch( ..., completion event = m_ev, ...)` and then `doca_dpa_kernel_launch( wait event = m_ev, ...)`, the latter kernel launch call would have shorter latency (as low as ~3 microseconds) for the start of the kernel's execution.

3.3.8.5. Limitations

- The order in which kernels are launched is important. If an application launches K1 and then K2, K1 must not depend on K2's completion (e.g., wait on its wait event that K2 should update).

  Note: Not following this guideline leads to unpredictable results (at runtime) for the application and might require restarting the DOCA DPA context (i.e., destroying, reinitializing, and rerunning the workload).

- DPA threads are an actual hardware resource and are, therefore, limited in number to 256 (including internal allocations and allocations explicitly requested by the user as part of the kernel launch API)
  - DOCA DPA does not check these limits. It is up to the application to adhere to this number and track thread allocation across different DPA contexts.
  - Each `doca_dpa_worker` consumes 16 threads

    Tip: It is recommended to have a single worker per node/process.

- The DPA has an internal watchdog timer to make sure threads do not block indefinitely. Kernel execution time must be finite and not exceed the time returned by `doca_dpa_get_kernel_max_run_time`.
- The `nthreads` parameter in the `doca_dpa_kernel_launch` call cannot exceed the maximum allowed number of threads to run a kernel returned by `doca_dpa_get_max_threads_per_kernel`.

3.3.9. Logging

The following device-side API (to be used in kernels) supports logging to stdout:

```
doca_dpa_dev_printf(const char *format, ...)
```

3.3.10. Error Handling

DPA context can enter an error state caused by the device flow. The application can check this error state by calling the following host API call:

```
doca_dpa_peek_at_last_error(const doca_dpa_t dpa)
```

This data path call indicates if an error occurred on device. If so, the DPA context enters a fatal state and can no longer be used by the application. Therefore, the application must destroy the DPA context.

3.4. Hello World Example

3.4.1. Procedure Outline

The following details the development flow of the example provided:

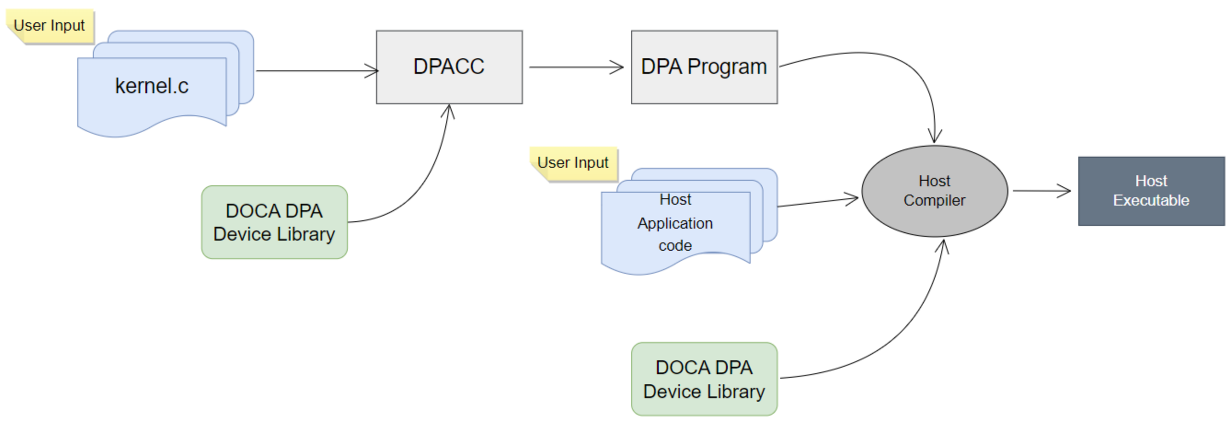
1. Write DPA device code (i.e., kernels or `.c` files).
2. Use DPACC to build a DPA program (i.e., a host library which contains an embedded device executable). Input for DPACC:

   a). Kernels from step 1.

   b). DOCA DPA device library.
3. Build host executable using a host compiler. Input for the host compiler:

   a). DPA program.

   b). User host application `.c`/`.cpp` files.



3.4.2. Procedure Steps

The following code snippets show a very basic DPA code that eventually prints "Hello World" to `stdout`.

1. Prepare kernel code:

```
#include "doca_dpa_dev.h"

__dpa_global__ void hello_world_kernel(void)
{
        doca_dpa_dev_printf("Hello World\n");
}
```

2. Prepare application code:

```
#include <stdio.h>
#include <unistd.h>
#include <doca_dev.h>
#include <doca_error.h>
#include "doca_dpa.h"

/**
* A struct that includes all needed info on registered kernels and is initialized
 during linkage by DPACC.
* Variable name should be the token passed to DPACC with --app-name parameter.
*/
extern doca_dpa_app_t dpa_hello_world_app;

/**
* kernel declaration that the user must declare for each kernel and DPACC is
 responsible to initialize.
* Only then, user can use this variable in the kernel launch API as follows:
* doca_dpa_kernel_launch(..., &hello_world_kernel, ...);
*/
doca_dpa_func_t hello_world_kernel;

int main(int argc, char **argv)
{
        /* local variables... */

        /* Open doca device */
        printf("\n----> Open DOCA device\n");
        /* Open appropriate DOCA device doca_dev... */

        /* Create doca_dpa context */
        printf("\n----> Create DOCA DPA context\n");
        doca_dpa_create(doca_dev, dpa_hello_world_app, &doca_dpa, 0);
        /* Create a CPU event that will be signaled when kernel is finished */
        printf("\n----> Create DOCA DPA event\n");
        doca_dpa_event_create(doca_dpa, DOCA_DPA_EVENT_ACCESS_DPA,
 DOCA_DPA_EVENT_ACCESS_CPU,
        DOCA_DPA_EVENT_WAIT_DEFAULT, &cpu_event, 0);

        /* Kernel launch */
        printf("\n----> Launch hello_world_kernel\n");
        doca_dpa_kernel_launch(doca_dpa, NULL, 0, cpu_event, 1,
 DOCA_DPA_EVENT_OP_SET, &hello_world_kernel, 1);

        /* Waiting for completion of kernel launch */
        printf("\n----> Waiting for hello_world_kernel to finish\n");
        doca_dpa_event_wait_until(cpu_event, 1, DOCA_DPA_EVENT_CMP_GEQ);

        /* Tear down */
        printf("\n----> Destroy DOCA DPA event\n");
        doca_dpa_event_destroy(cpu_event);

        printf("\n----> Destroy DOCA DPA context\n");
        doca_dpa_destroy(doca_dpa);

        printf("\n----> Destroy DOCA device\n");
        doca_dev_close(doca_dev);

        printf("\n----> DONE!\n");
        return 0;
}
```

3. Build DPA program:

```
/opt/mellanox/doca/tools/dpacc \
    libdoca_dpa_dev.a \
    kernel.c \
    -o dpa_program.a \
    -hostcc=gcc \
    -hostcc-options="-Wno-deprecated-declarations" \
    --devicecc-options="-D__linux__ -Wno-deprecated-declarations" \
    --app-name="dpa_hello_world_app" \
    -I/opt/mellanox/doca/include/
```

4. Build host application:

```
gcc hello_world.c -o hello_world \
    dpa_program.a libdoca_dpa.a \
    -I/opt/mellanox/doca/include/ \
    -DDOCA_ALLOW_EXPERIMENTAL_API \
    -L/opt/mellanox/doca/lib/x86_64-linux-gnu/ -ldoca_common \
    -L/opt/mellanox/flexio/lib/ -lflexio \
    -lstdc++ -libverbs -lmlx5
```

5. Execution:

```
$ ./hello_world mlx5_0


***********************************************
**********    Hello World Example    **********
***********************************************


----> Open DOCA device

----> Create DOCA DPA context

----> Create DOCA DPA event

----> Launch hello_world_kernel

----> Waiting for hello_world_kernel to finish
```

```
----> Destroy DOCA DPA event
/  7/Hello World

----> Destroy DOCA DPA context

----> Destroy DOCA device

----> DONE!
```

3.5. DOCA DPA Samples

This section provides DPA sample implementation on top of the BlueField-3 DPU.

3.5.1. DPA Sample Prerequisites

The BlueField-3 DPU's link layer must be configured to InfiniBand and it must also be set to operate in DPU mode.

Note: Running the samples on the host is not currently supported.

3.5.2. Running DPA 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_dpa/<sample_name>
   meson build
   ninja -C build
   ```
   Note: The binary `doca_<sample_name>` will be created under `./build/`.
3. Sample (e.g., `dpa_kernel_launch`) usage:
   ```
   Usage: doca_dpa_kernel_launch [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:
     -d, --device <IB device name>     IB device name that supports DPA (optional).
    If not provided then a random IB device will be chosen
   ```
4. For additional information per sample, use the `-h` option after the `--` separator:
   ```
   ./build/doca_<sample_name> -h
   ```

3.5.3. Samples

3.5.3.1. DPA Kernel Launch

This sample illustrates how to launch a DOCA DPA kernel with a completion event.

The sample logic includes:

1. Allocating DOCA DPA resources.
2. Initializing completion event for the DOCA DPA kernel.
3. Running a "hello_world" DOCA DPA kernel that prints "Hello from kernel".
4. Waiting on completion event of the kernel.
5. Destroying the event and resources.

Reference:

- `/opt/mellanox/doca/samples/doca_dpa/dpa_kernel_launch/dpa_kernel_launch_main.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_kernel_launch/host/dpa_kernel_launch_sample.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_kernel_launch/device/dpa_kernel_launch_kernels_dev.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_kernel_launch/meson.build`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_common.h`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_common.c`
- `/opt/mellanox/doca/samples/doca_dpa/build_dpacc_samples.sh`

3.5.3.2. DPA Wait Kernel Launch

This sample illustrates how to launch a DOCA DPA kernel with wait and completion events.

The sample logic includes:

1. Allocating DOCA DPA resources.
2. Initializing wait and completion events for the DOCA DPA kernel.
3. Running hello_world DOCA DPA kernel that waits on the wait event.
4. Running a separate thread that triggers the wait event.
5. hello_world DOCA DPA kernel prints "Hello from kernel".
6. Waiting for the completion event of the kernel.
7. Destroying the events and resources.

Reference:

- `/opt/mellanox/doca/samples/doca_dpa/dpa_kernel_launch/dpa_kernel_launch_main.c`

- `/opt/mellanox/doca/samples/doca_dpa/dpa_kernel_launch/host/dpa_kernel_launch_sample.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_kernel_launch/device/dpa_kernel_launch_kernels_dev.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_kernel_launch/meson.build`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_common.h`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_common.c`
- `/opt/mellanox/doca/samples/doca_dpa/build_dpacc_samples.sh`

3.5.3.3. DPA Binary Tree

This sample illustrates how to launch multiple DOCA DPA kernels in a binary tree pattern.

The sample logic includes:

1. Allocating DOCA DPA resources.
2. Initializing wait and completion events for the DOCA DPA kernels.
3. Running 7 DOCA DPA kernels, such that each kernel (except the first) waits on the parent kernel in a binary tree-like pattern.
4. Waiting on all 7 completion events (completion event for each kernel).
5. Destroying the events and resources.

Reference:

- `/opt/mellanox/doca/samples/doca_dpa/dpa_binary_tree/dpa_binary_tree_main.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_binary_tree/host/dpa_binary_tree_sample.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_binary_tree/device/dpa_binary_tree_kernels_dev.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_binary_tree/meson.build`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_common.h`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_common.c`
- `/opt/mellanox/doca/samples/doca_dpa/build_dpacc_samples.sh`

3.5.3.4. DPA Diamond Tree

This sample illustrates how to launch multiple DOCA DPA kernels in a diamond tree-like pattern.

The sample logic includes:

1. Allocating DOCA DPA resources.
2. Initializing wait and completion events for the DOCA DPA kernel.
3. Running 4 kernels, such that each kernel (except the first) waits on the parent kernel in a diamond tree-like pattern.

4. Waiting on the completion event of the last kernel.
5. Destroying the events and resources.

Reference:

- `/opt/mellanox/doca/samples/doca_dpa/dpa_diamond_tree/dpa_diamond_tree_main.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_diamond_tree/host/dpa_diamond_tree_sample.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_diamond_tree/device/dpa_diamond_tree_kernels_dev.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_diamond_tree/meson.build`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_common.h`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_common.c`
- `/opt/mellanox/doca/samples/doca_dpa/build_dpacc_samples.sh`

3.5.3.5. DPA Endpoint Copy

Note: This sample does not exist in Ubuntu DOCA BFBs.

This sample illustrates how to perform RDMA copy using DOCA DPA kernels and DOCA DPA endpoints. This sample launches another thread considered to be a "remote thread" to copy to (the thread is not actually remote as the main process spawns it). To avoid confusion, the main process is called the "main thread".

The goal is to illustrate how the main thread can copy a buffer (in this example, an integer) to the remote thread's memory.

The sample logic includes:

1. The main thread allocating DOCA DPA resources.
2. The main thread creating DOCA DPA events for wait and completion of DOCA DPA kernels, and another event for communication with the remote thread.
3. Launching the remote thread.
4. The main thread preparing the DOCA DPA endpoint resources, registering the buffer to copy as a DOCA DPA host memory, and signaling (using the communication event) to the remote thread that it is ready.
5. The remote thread preparing the DOCA DPA endpoint resources, registering a buffer to copy to it as a DOCA DPA host memory, and signaling (using the communication event) to the main thread that it is ready.
6. The main and remote thread connecting to each other's endpoints.
7. The main thread launching a DOCA DPA kernel that copies the main thread's buffer to the remote thread's buffer.
8. The DOCA DPA kernel synchronizing the endpoint and finishing.
9. The main thread waiting on the completion event of the kernel.
10. The main thread signaling to the remote event that the copy has finished.

11. The remote thread destroying its resources and joining the main thread.
12. The main thread destroying the events and resources.

Reference:

- `/opt/mellanox/doca/samples/doca_dpa/dpa_endpoint_copy/dpa_endpoint_copy_main.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_endpoint_copy/host/dpa_endpoint_copy_sample.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_endpoint_copy/device/dpa_endpoint_copy_kernels_dev.c`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_endpoint_copy/meson.build`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_common.h`
- `/opt/mellanox/doca/samples/doca_dpa/dpa_common.c`
- `/opt/mellanox/doca/samples/doca_dpa/build_dpacc_samples.sh`

Chapter 4. Known Limitations

4.1. Supported Devices

- BlueField-3 based DPUs

4.2. Supported Host OS

- Windows is not supported

4.3. Supported SDKs

- DOCA FlexIO at beta level
- DOCA DPA at alpha level
- DOCA DPA is tested only for InfiniBand devices

4.4. Toolchain

- DPA image-signing and signature-verification are not currently supported
- Debugger (GDB) is currently not supported

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