DOCA Programming Overview

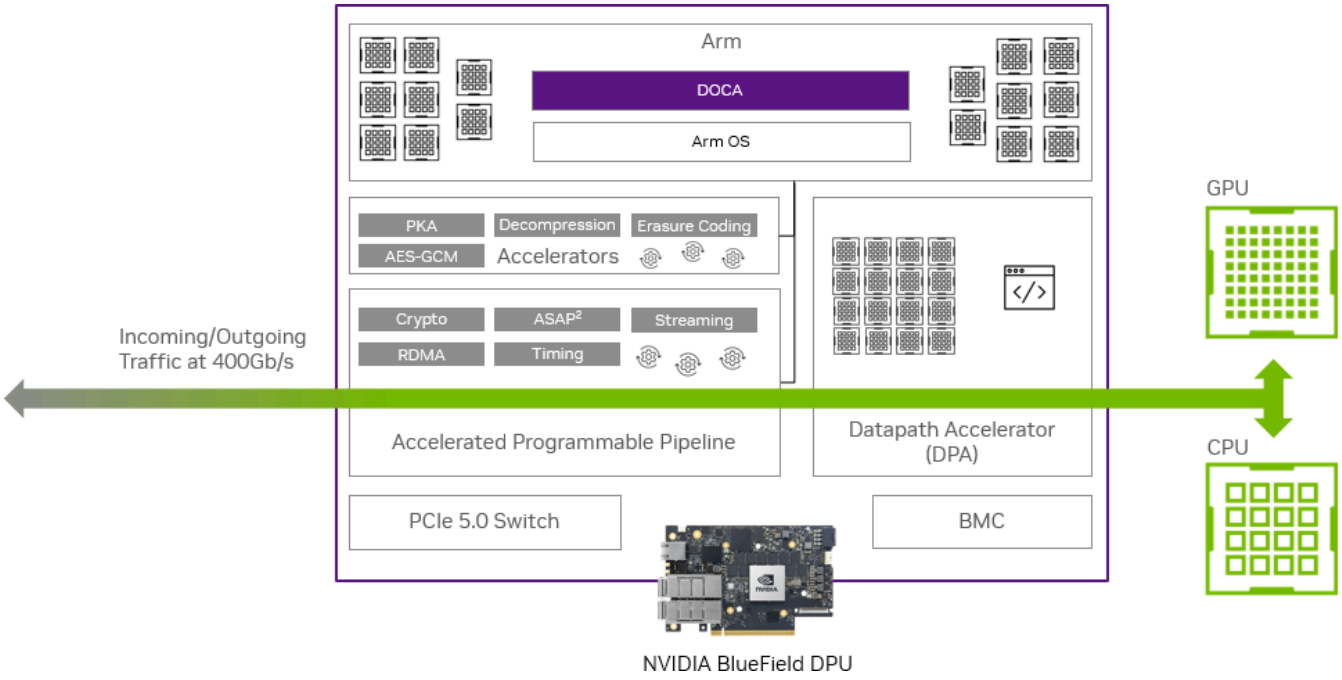
Table of contents

This section contains the following pages:

- Hardware Architecture
- DOCA SDK Architecture

Hardware Architecture

BlueField's main hardware entities, which are optimized for different workloads.



Arm Cores

Optimized for control-path applications, general-purpose computing, and single-flow performance.

- 16 Arm Cortex-A78 cores for general-purpose processing
- Coherent Mesh architecture for efficient interconnectivity
- Last-Level Cache (LLC) for optimized memory performance
- DDR5 memory subsystem for high-speed data access
- Base OS and microservices for managing system resources

Accelerated Programmable Pipeline

Optimized for high-performance packet processing and advanced packet handling.

- Programmable 64-128 packet processor for flexible networking
- Multi-stage, highly parallelized architecture for throughput optimization
- Flow-based classification and action engine for efficient packet processing
- Support for RDMA, cryptographic acceleration, and time-based scheduling

Data-Path Accelerator

Designed for I/O-intensive applications, high insertion rate tasks, network flow processing, device emulation, and DMA operations.

- 16 hyper-threaded I/O and packet processing cores for handling intensive workloads
- Real-time OS for deterministic and low-latency operations

DOCA SDK Architecture

DOCA provides libraries for networking and data processing programmability that leverage NVIDIA® BlueField® networking platform (DPU or SuperNIC) and NVIDIA® ConnectX® NIC hardware accelerators.

DOCA software framework is built on top of DOCA Core, which provides a unified software framework for DOCA libraries, to form a processing pipeline or workflow build of one or many DOCA libraries.

Device Subsystem

The DOCA SDK allows applications to offload resource intensive tasks (e.g., encryption, and compression) to hardware. DOCA also allows applications to offload network related tasks (e.g., packet acquisition, RDMA send). As such, BlueField and ConnectX provide dedicated hardware processing units for executing such tasks.

The DOCA device subsystem provides an abstraction of the hardware processing units referred to as device.

DOCA Device subsystem provides means to:

- Discover available hardware acceleration units provided by DPUs/SuperNICs/NICs
- Query capabilities and properties of available hardware acceleration units
- Open device to enable libraries to allocate and share resources necessary for hardware acceleration

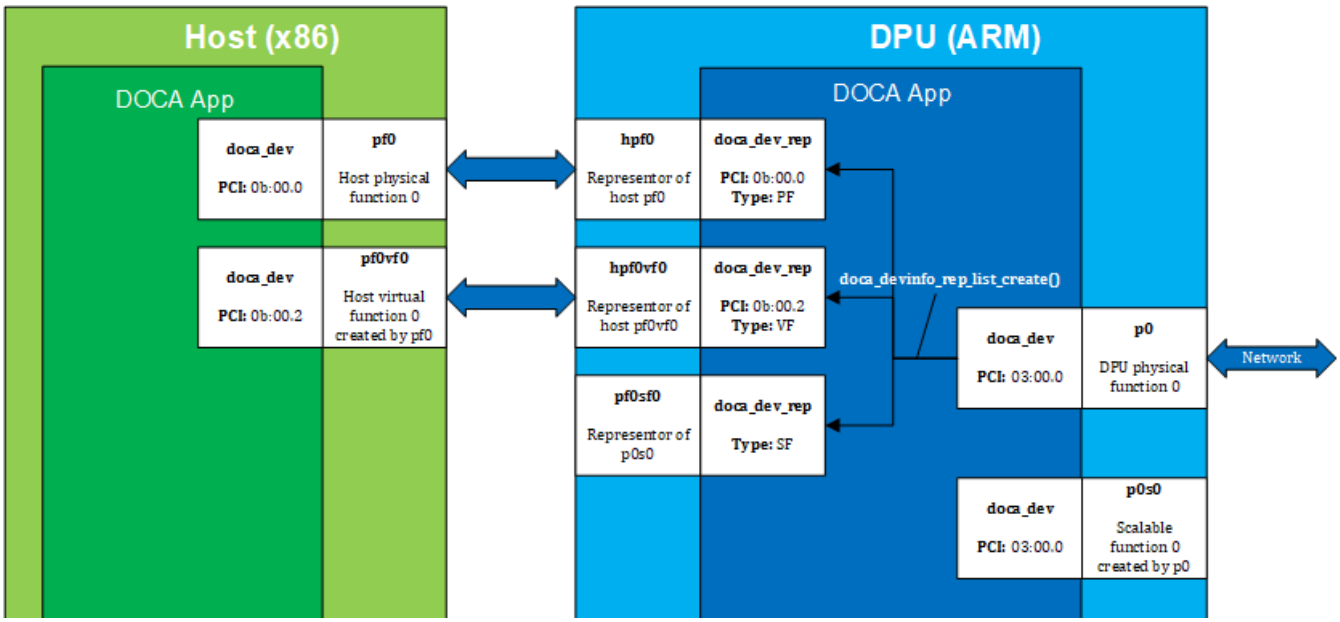
On a given system, there can be multiple available devices. An application can choose a device based on the following characteristics topology (e.g., PCIe address) and/or capabilities (e.g., encryption support).

DOCA Core supports two DOCA Device types:

- Local device – this is an actual device exposed in the local system (BlueField or host) and can perform DOCA library processing jobs. This can be a PCIe physical function (PF), virtual function (VF), or scalable function (SF)

- Representor device – this is a representation of a local device. The represented local device is typically on the host (except for SFs) and the representor is always on the BlueField side (a proxy on the BlueField for the host-side device).

The following figure provides an example of host local devices with representors on BlueField:



> **Note**
>
> The diagram shows typical topology when using BlueField in DPU mode as described in NVIDIA BlueField DPU Modes of Operation .

The diagram shows BlueField (on the right side of the figure) connected to a host (on the left). The host has physical function PF0 with a child virtual function VF0.

The BlueField side has a representor-device per host function in a 1-to-1 ratio (e.g., `hpf0` is the representor device for the host's PF0 device, etc.) as well as a representor for each SF function, such that both the SF and its representor reside in BlueField.

> **Info**

For more details on the DOCA Device subsystem, see section "DOCA Device".

Memory Management Subsystem

Hardware processing tasks require data buffers as inputs and/or outputs to processing operations. The application is responsible to provide the input data and/or read the output data. To achieve maximum performance, the SDK uses zero-copy technology to pass data to hardware. To allow zero-copy, the application must register the memory that would hold data buffers beforehand. The memory management subsystem provides a means to register memory and manage allocation of data buffers on registered memory.

Memory registration:

- Defines user application memory range to use to hold data buffers
- Allows one or more devices to access the memory range
- Defines the access permission (e.g., read only)

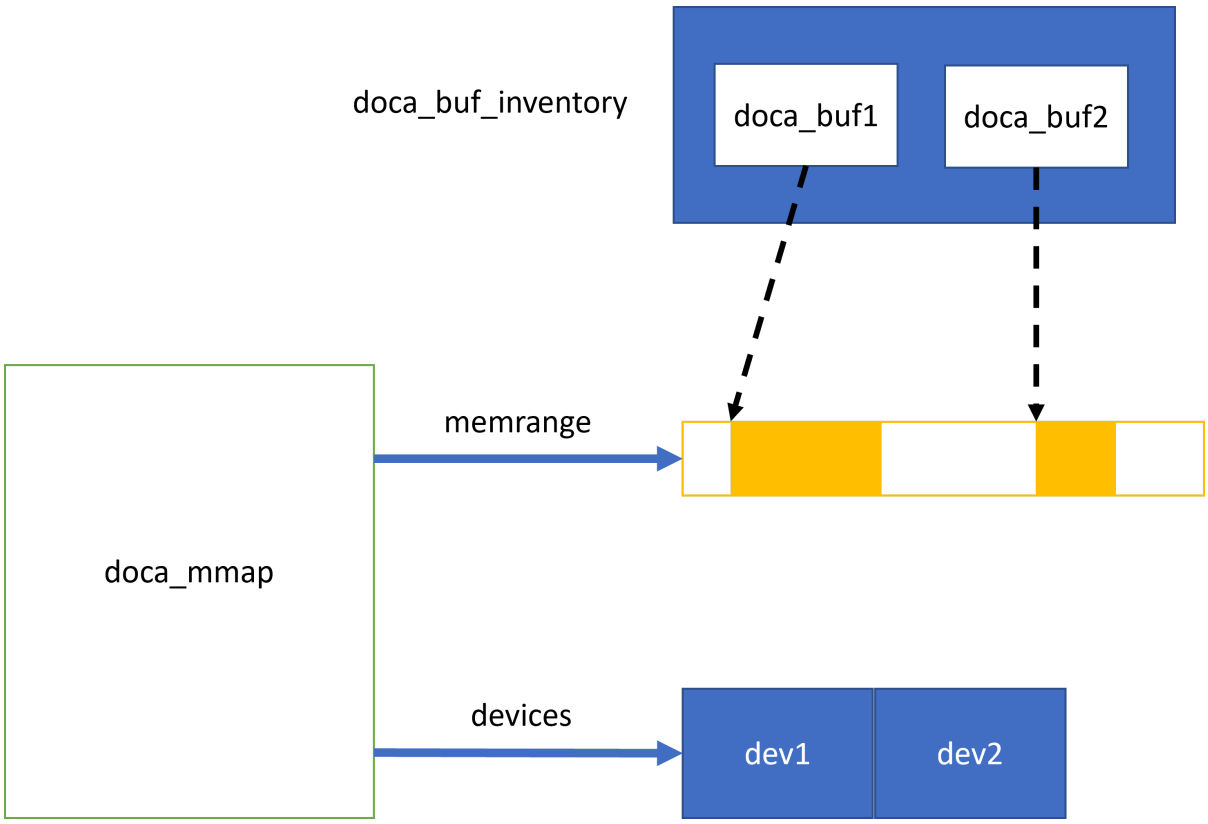
Data buffer allocation management:

- Allows allocating data buffers that cover subranges within the registered memory
- Allows memory pool semantics over registered memory

DOCA memory has the following main components:

- `doca_buf` – describes a data buffer, and is used as input/output to various hardware processing tasks within DOCA libraries
- `doca_mmap` – describes registered memory, which is accessible by devices, with a set of permissions. `doca_buf` is a segment in the memory range represented by `doca_mmap`.
- `doca_buf_inventory` – pool of `doca_buf` with the same characteristics (see more in sections "DOCA Core Buffers" and "DOCA Core Inventories")

The following diagram shows the various modules within the DOCA memory subsystem:



The diagram shows a `doca_buf_inventory` containing 2 `doca_buf`s. Each `doca_buf` points to a portion of the memory buffer which is part of a `doca_mmap`. The mmap is populated with one continuous memory range and is registered with 2 DOCA Devices, `dev1` and `dev2`.

> **Info**
>
> For more details about DOCA Memory management subsystem, see section "DOCA Memory Subsystem".

Execution Model

DOCA SDK introduces libraries that utilize hardware processing units. Each library defines dedicated APIs for achieving a specific processing task (e.g., encryption). The library abstracts all the low-level details related to operation of the hardware, allowing the application focus on what matters. This type of library is referred to as a context. Since a context utilizes a hardware processing unit, it requires a device to operate. This device also

determines which buffers are accessible by that context. Contexts provide hardware processing operation APIs in the form of tasks and events.

Task:

- Application prepares the task arguments
- Application submits the task; this issues a request to the relevant hardware processing unit
- Application receives a completion in the form of a callback once hardware processing completes

Event:

- Application registers to the event. This informs hardware to report whenever the event occurs.
- Application receives a completion in the form of a callback every time hardware identifies that the event has occurred

Since hardware processing is asynchronous in nature. DOCA provides an object that allows waiting on processing operations (tasks and events). This object is referred to as a Progress Engine (PE). The PE allows waiting on completions using the following methods:

- Busy waiting/polling mode – in this case, the application repeatedly invokes a method that checks if a completion has occurred
- Notification-driven mode – in this case, the application can use OS primitives (e.g., `linux event fd`) to notify the thread whenever some completion has occurred

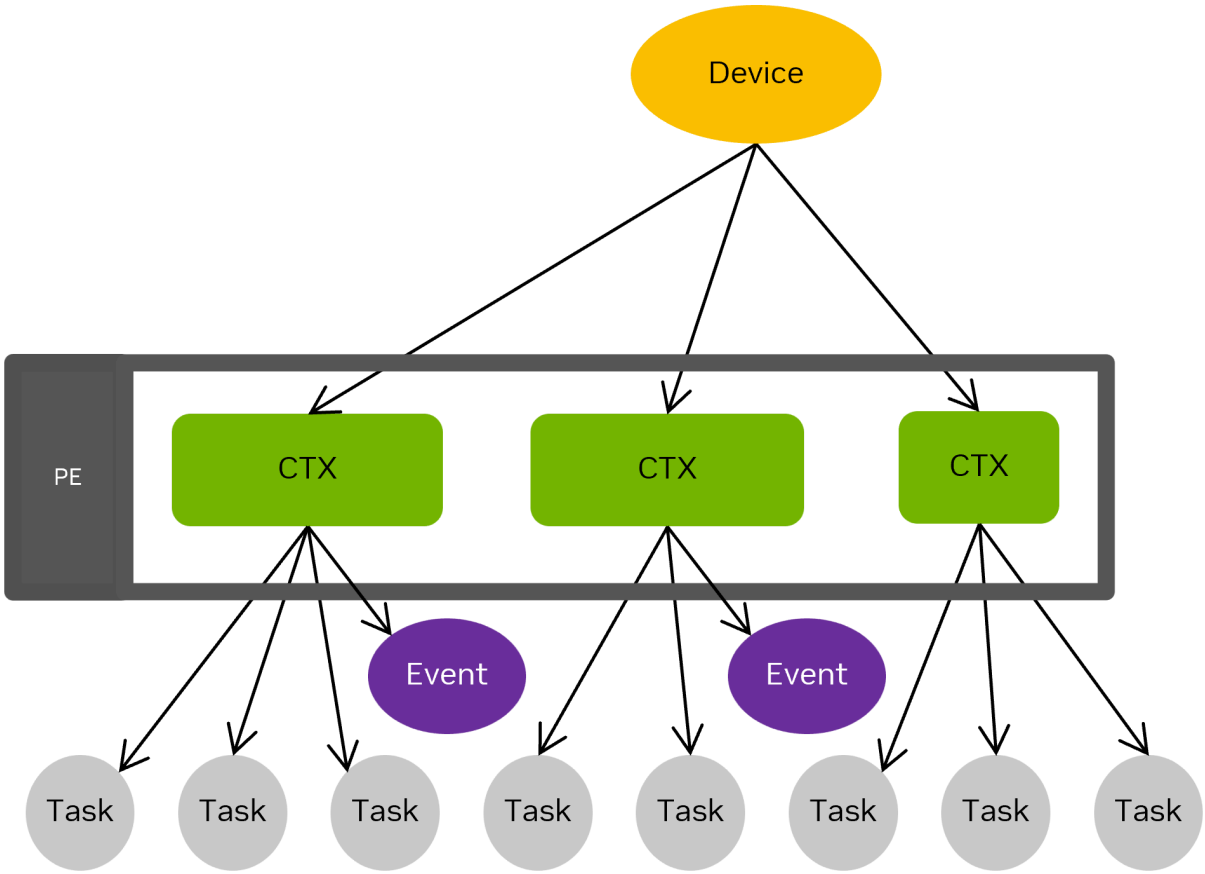
Once completion occurs, whether caused by a task or event, the relevant callback is invoked as part of the PE method.

A single PE instance allows waiting on multiple tasks/events from different contexts. As such, it is possible for an application to utilize a single PE per thread.

Info

For more details about the DOCA Progress Engine, see section "DOCA Progress Engine".

The following diagram illustrates how a combination of various DOCA modules combine DOCA cross-library processing runtime.



The diagram shows 3 contexts utilizing the same device, each context has some tasks/events that have been submitted/registered by the application. All 3 contexts are connected to the same PE, where the application can use the same PE to wait on all completions at once.

Info

For more details about DOCA Execution model see section "DOCA Execution Model".