NVIDIA DOCA GPUNetIO Programming Guide

Programming Guide

MLNX-15-060587 _v2.2.0 | October 2023

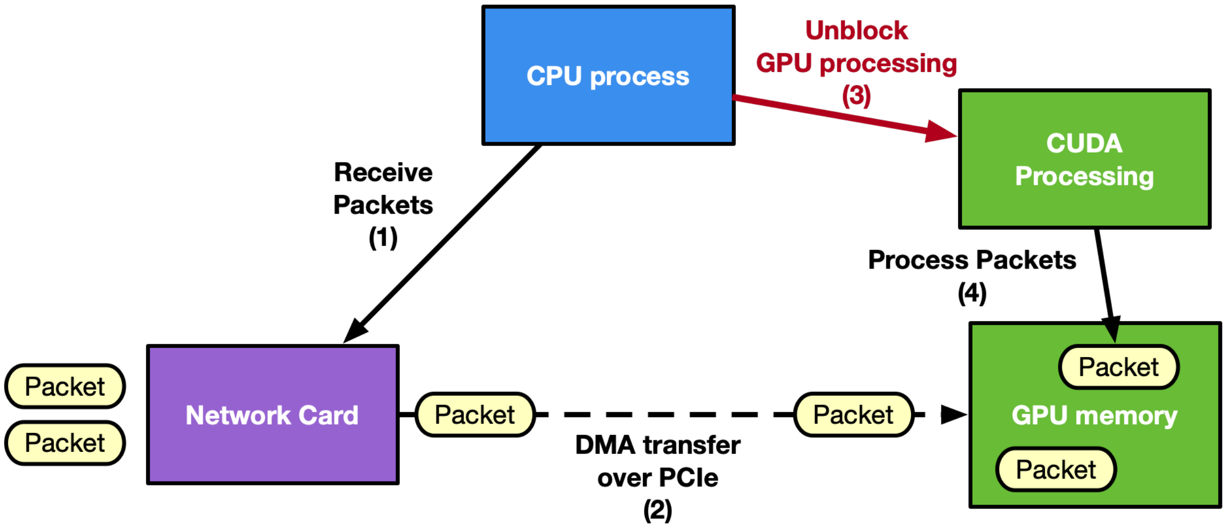
Table of Contents

Chapter 1. Introduction

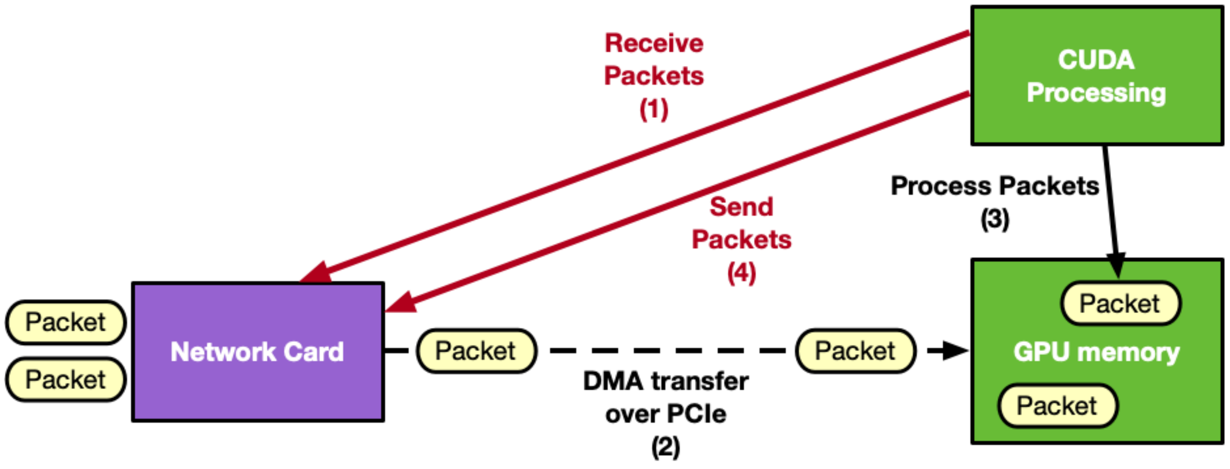
Real-time GPU processing of network packets is a technique useful for application domains involving signal processing, network security, information gathering, input reconstruction, and more. These applications involve the CPU in the critical path (CPU-centric approach) to coordinate the network card (NIC) for receiving packets in the GPU memory (GPUDirect RDMA) and notifying a packet-processing CUDA kernel waiting on the GPU for a new set of packets. In lower-power platforms, the CPU can easily become the bottleneck, masking GPU value. The aim is to maximize the zero-packet-loss throughput at the the lowest latency possible.

A CPU-centric approach may not be scalable when increasing the number of clients connected to the application as the time between two receive operations on the same queue (client) would increase with the number of queues. The new DOCA GPUNetIO library allows developers to orchestrate these kinds of applications while optimizing performance, combining GPUDirect RDMA for data-path acceleration, GDRCopy library to give the CPU direct access to GPU memory, and GPUDirect Async kernel-initiated communications to allow a CUDA kernel to directly control the NIC.

CPU-centric approach:



GPU-centric approach:



DOCA GPUNetIO enables GPU-centric solutions that remove the CPU from the critical path by providing the following features:

- GPUDirect Async Kernel-Initiated Network (GDAKIN) communications – a CUDA kernel can invoke GPUNetIO device functions to receive or send, directly interacting with the NIC
  - CPU intervention is not needed in the application critical path
- GPUDirect RDMA – receive packets directly into a contiguous GPU memory area
- Semaphores – provide a standardized I/O communication protocol between the receiving entity and the CUDA kernel real-time packet processing
- Smart memory allocation – allocate aligned GPU memory buffers exposing them to direct CPU access
  - Combination of CUDA and DPDK gpudev library (which requires the GDRCopy library) already embedded in the DPDK released with DOCA
- Ethernet protocol management on GPU

Morpheus and Aerial 5G SDK are examples of NVIDIA applications actively using DOCA GPUNetIO.

For a deep dive into the technology and motivations, please refer to the NVIDIA Blog post Inline GPU Packet Processing with NVIDIA DOCA GPUNetIO.

Chapter 2. Prerequisites

DOCA GPUNetIO requires a properly configured environment. The following subsections describe the required setup.

Note: Currently, DOCA GPUNetIO is included only in DOCA for Host package for Ubuntu 20.04 and Ubuntu 22.04 with CUDA 12.1 or newer.

To install all DOCA GPUNetIO components, run:

```
apt install -y doca-gpu doca-gpu-dev
```

Ensure IOMMU is disabled. It can be explicitly disabled through the grub command line as follows:

```
$ sudo vim /etc/default/grub
# Add iommu=off to the CMDLINE along with other options
# GRUB_CMDLINE_LINUX_DEFAULT="iommu=off"
$ sudo update-grub
$ sudo reboot
```

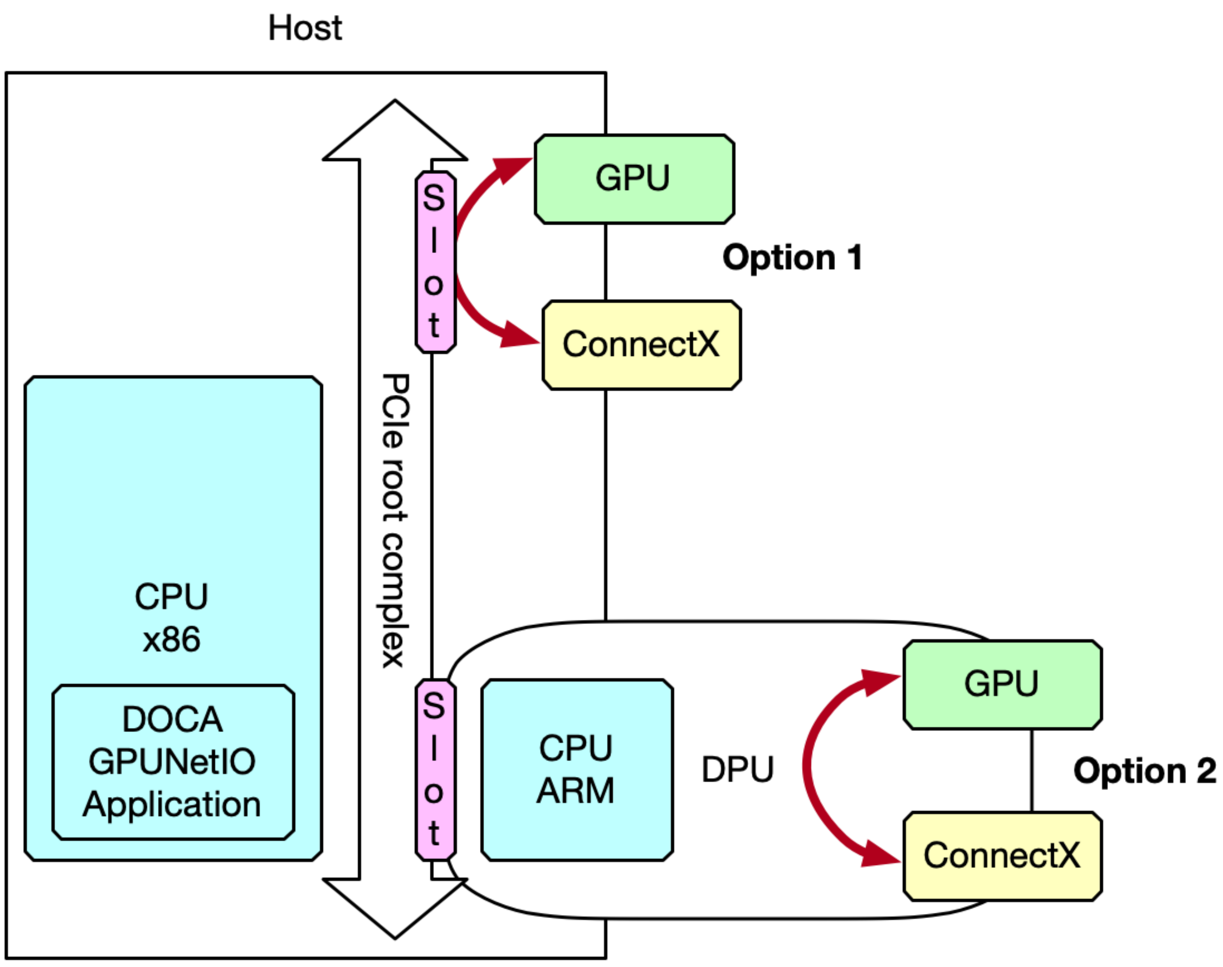
WARNING: DOCA GPUNetIO has been tested on bare-metal and in docker but never in a virtualized environment. Using KVM is discouraged for now.

2.1. Hardware Topology

Internal hardware topology of the system should be GPUDirect-RDMA-friendly to maximize the internal throughput between the GPU and the NIC.

Assuming the application is running on the host's CPU cores, there must be a dedicated PCIe connection between the GPU and the NIC which can be implemented in two ways:

- Connecting an additional PCIe switch to a PCIe slot in the host system bus
- Connecting a DPU converged card exposing the GPU and NIC to the host



You may check the topology of your system using `lspci -tvvv` or `nvidia-smi topo -m`.

On some host systems, the PCIe access control services (ACS) must be disabled to ensure direct communication between the NIC and the GPU. Please refer to this page and this page for more information.

2.1.1. Option 1: Network Card in Ethernet Mode

Note: NVIDIA® ConnectX® firmware must be 22.36.1010 or later.

DOCA GPUNetIO allows a CUDA kernel to control the network card when dealing with Ethernet protocol. For this reason, the ConnectX NIC on your system must be set in Ethernet mode.

```
# Start MST
mst start
mst status -v

MST modules:
------------
```

```
    MST PCI module is not loaded
    MST PCI configuration module loaded
PCI devices:
------------
DEVICE_TYPE             MST                           PCI       RDMA            NET
                      NUMA
ConnectX6DX(rev:0)      /dev/mst/mt4125_pciconf0.1    b5:00.1   mlx5_1          net-
ens6f1                0
ConnectX6DX(rev:0)      /dev/mst/mt4125_pciconf0      b5:00.0   mlx5_0          net-
ens6f0                0

# Configure Ethernet mode
mlxconfig -d /dev/mst/mt4125_pciconf0 s KEEP_ETH_LINK_UP_P1=1 KEEP_ETH_LINK_UP_P2=1
 KEEP_IB_LINK_UP_P1=0 KEEP_IB_LINK_UP_P2=0
mlxconfig -d /dev/mst/mt4125_pciconf0 --yes set ACCURATE_TX_SCHEDULER=1
 REAL_TIME_CLOCK_ENABLE=1

# Cold reboot
ipmitool power cycle
```

2.1.2. Option 2: DPU Converged Card

Note: DPU firmware must be 24.35.2000 or newer.

To expose and use the GPU and the NIC on the DPU converged card from an application running on the host, configure the DPU to operate in NIC mode:

```
# Enable MST
sudo mst start
sudo mst status

# MST devices:
# ------------
# /dev/mst/mt41686_pciconf0        - PCI configuration cycles access.
#                                    domain:bus:dev.fn=0000:b8:00.0 addr.reg=88
 data.reg=92 cr_bar.gw_offset=-1
#                                    Chip revision is: 01

# Expose the GPU on the DPU converged card to host. For BF2 offset is 4, for BF3
 offset is 8
sudo mlxconfig -d /dev/mst/mt41686_pciconf0 --yes s PCI_DOWNSTREAM_PORT_OWNER[4]=0x0

# Set the BlueField-2 port to Ethernet mode (not InfiniBand)
sudo mlxconfig -d /dev/mst/mt41686_pciconf0 --yes set LINK_TYPE_P1=2 LINK_TYPE_P2=2

# Set the BlueField-2 to operate in DPU (Embedded CPU) mode
sudo mlxconfig -d /dev/mst/mt41686_pciconf0 --yes set INTERNAL_CPU_MODEL=1
 INTERNAL_CPU_PAGE_SUPPLIER=1 INTERNAL_CPU_ESWITCH_MANAGER=1
 INTERNAL_CPU_IB_VPORT0=1 INTERNAL_CPU_OFFLOAD_ENGINE=DISABLED

# Accurate scheduling related settings
sudo mlxconfig -d /dev/mst/mt41686_pciconf0 --yes set ACCURATE_TX_SCHEDULER=1
 REAL_TIME_CLOCK_ENABLE=1

# Cold reboot
sudo ipmitool power cycle

# Verify that the DPU firmware changes have been applied
sudo mlxconfig -d /dev/mst/mt41686_pciconf0 q LINK_TYPE_P1 LINK_TYPE_P2
 INTERNAL_CPU_MODEL INTERNAL_CPU_PAGE_SUPPLIER INTERNAL_CPU_ESWITCH_MANAGER
 INTERNAL_CPU_IB_VPORT0 INTERNAL_CPU_OFFLOAD_ENGINE ACCURATE_TX_SCHEDULER
 REAL_TIME_CLOCK_ENABLE
         LINK_TYPE_P1                              ETH(2)
         LINK_TYPE_P2                              ETH(2)
         INTERNAL_CPU_MODEL                            EMBEDDED_CPU(1)
         INTERNAL_CPU_PAGE_SUPPLIER                    EXT_HOST_PF(1)
         INTERNAL_CPU_ESWITCH_MANAGER                  EXT_HOST_PF(1)
         INTERNAL_CPU_IB_VPORT0                        EXT_HOST_PF(1)
         INTERNAL_CPU_OFFLOAD_ENGINE                   DISABLED(1)
         ACCURATE_TX_SCHEDULER                         True(1)
         REAL_TIME_CLOCK_ENABLE                        True(1)
```

2.2. PCIe Configuration

On some host systems, the PCIe access control services (ACS) must be disabled to ensure direct communication between the NIC and the GPU. Please refer to this page and this page for more information.

2.3. GPU Configuration

On the host, CUDA Toolkit 12.1 or newer must be installed. It is also recommended to enable persistence mode to decrease initial application latency `nvidia-smi -pm 1`.

To allow the NIC to send and receive packets using GPU memory, it is required to launch the NVIDIA kernel module `nvidia-peermem` (using `modprobe nvidia-peermem`).

To allow the CPU to directly access the GPU memory without the need for CUDA API, DPDK and DOCA require the GDRCopy kernel module to be installed on the system:

```
# Run nvidia-peermem kernel module
sudo modprobe nvidia-peermem

# Install GDRCopy
sudo apt install -y check kmod
git clone https://github.com/NVIDIA/gdrcopy.git /opt/mellanox/gdrcopy
cd /opt/mellanox/gdrcopy
make
# Run gdrdrv kernel module
./insmod.sh

# Double check nvidia-peermem and gdrdrv module are running
$ lsmod | egrep gdrdrv
gdrdrv                 24576  0
nvidia              55726080  4 nvidia_uvm,nvidia_peermem,gdrdrv,nvidia_modeset

# Export library path
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/opt/mellanox/gdrcopy/src

# Ensure CUDA library path is in the env var
export PATH="/usr/local/cuda-12/bin:${PATH}"
export LD_LIBRARY_PATH="/usr/local/cuda-12/lib:/usr/local/cuda-12/lib64:
${LD_LIBRARY_PATH}"
export CPATH="$(echo /usr/local/cuda-12/targets/{x86_64,sbsa}-linux/include | sed
 's/ /:/'):${CPATH}"
```

A good practice in GPU network applications is to spread (through RSS) incoming traffic among different receive queues to enhance the degree of parallelism in processing incoming packets. Therefore, it is important to double check that the BAR1 mapping is large enough to hold multiple receive queues.

To verify the BAR1 mapping space of a GPU you can use `nvidia-smi`:

```
$ nvidia-smi -q
```

```
==============NVSMI LOG==============

Timestamp                                 : Wed Apr 19 09:35:39 2023
Driver Version                            : 530.30.02
CUDA Version                              : 12.1

Attached GPUs                             : 1
GPU 00000000:CA:00.0
    Product Name                          : NVIDIA A100 80GB PCIe
    Product Brand                         : NVIDIA
    Product Architecture                  : Ampere
    Display Mode                          : Enabled
    Display Active                        : Disabled
    Persistence Mode                      : Enabled
.....
    BAR1 Memory Usage
        Total                             : 131072 MiB
        Used                              : 1 MiB
        Free                              : 131071 MiB
```

Chapter 3. Architecture

A GPU packet processing network application can be split into two fundamental phases:

- Setup on the CPU (devices configuration, memory allocation, launch of CUDA kernels, etc.)
- Main data path where GPU and NIC interact to exercise their functions

DOCA GPUNetIO provides different building blocks, some of them in combination with the DOCA Ethernet library, to create a full pipeline running entirely on the GPU.

During the setup phase on the CPU, applications must:

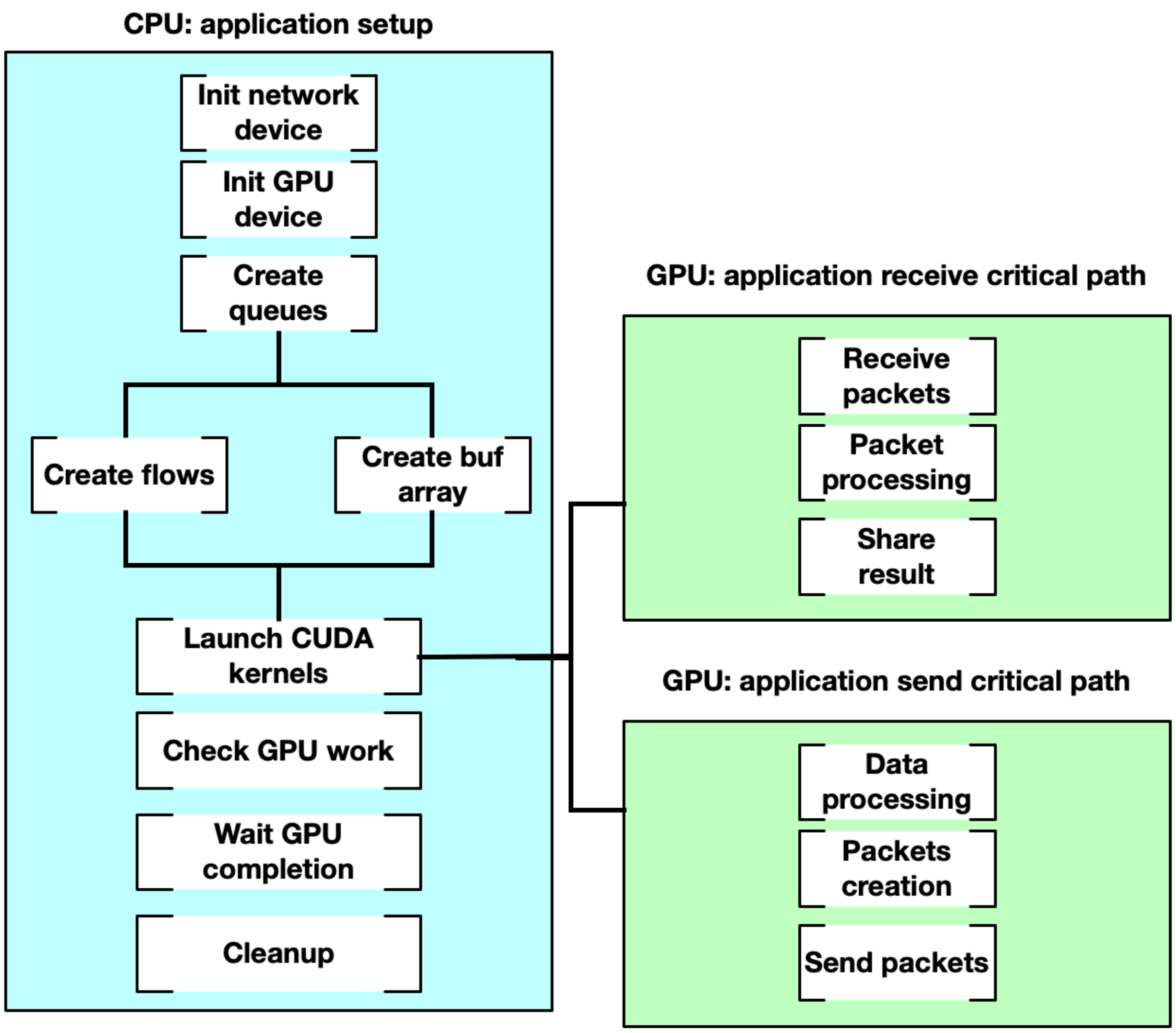
1. Prepare all the objects on the CPU.
2. Export a GPU handler for them.
3. Launch a CUDA kernel passing the object's GPU handler to work with the object during the data path.

For this reason, DOCA GPUNetIO is composed of two libraries:

- `libdoca_gpunetio` with functions invoked by CPU to prepare the GPU and allocate memory and objects
- `libdoca_gpunetio_device` with functions invoked by GPU within CUDA kernels during the data path

Important: The pkgconfig file for the DOCA GPUNetIO shared library is `doca-gpu.pc`. However, there is no pkgconfig file for the DOCA GPUNetIO CUDA device's static library `/opt/mellanox/doca/lib/x86_64-linux-gnu/libdoca_gpunetio_device.a`, so it must be explicitly linked to the CUDA application if DOCA GPUNetIO CUDA device functions are required.

The following diagram presents the typical flow:



Refer to NVIDIA DOCA GPU Packet Processing Application Guide the for an example of using DOCA GPUNetIO to send and receive Ethernet packets.

Chapter 4. API

This section details the specific structures and operations related to the main DOCA GPUNetIO API on CPU and GPU. GPUNetIO headers are:

- `doca_gpunetio.h` – CPU functions
- `doca_gpunetio_dev_buf.cuh` – GPU functions to manage a DOCA buffer array
- `doca_gpunetio_dev_eth_rxq.cuh` – GPU functions to manage a DOCA Ethernet receive queue
- `doca_gpunetio_dev_eth_txq.cuh` – GPU functions to manage a DOCA Ethernet send queue
- `doca_gpunetio_dev_sem.cuh` – GPU functions to manage a DOCA GPUNetIO semaphore

This section lists the main functions of DOCA GPUNetIO. To better understand their usage, refer to Building Blocks which includes several code examples.

Tip: To better understand structures, objects, and functions related to Ethernet send and receive, please refer to the NVIDIA DOCA Ethernet Programming Guide.

Tip: To better understand DOCA core objects like `doca_mmap` or `doca_buf_array`, please refer to the NVIDIA DOCA Core Programming Guide.

All DOCA Core and Ethernet object used with GPUNetIO have a GPU export function to obtain a GPU handler for that object. The following are a few examples:

- `doca_buf_array` is exported as `doca_gpu_buf_arr`:

```
struct doca_mmap *mmap;
struct doca_buf_arr *buf_arr_cpu;
struct doca_gpu_buf_arr *buf_arr_gpu;

doca_mmap_create(NULL, &(mmap));
/* Populate and start mmap */
doca_buf_arr_create(mmap, &buf_arr_cpu);
/* Populate and start buf arr attributes. Set datapath on GPU */
/* Export the buf array CPU handler to a buf array GPU handler */
doca_buf_arr_get_gpu_handle(buf_arr_cpu, &(buf_arr_gpu));
/* To use the GPU handler, pass it as parameter of the CUDA kernel */
cuda_kernel<<<...>>>(buf_arr_gpu, ...);
```

- `doca_eth_rxq` is exported as `doca_gpu_eth_rxq`:

```
struct doca_mmap *mmap;
struct doca_eth_rxq *eth_rxq_cpu;
struct doca_gpu_eth_rxq *eth_rxq_gpu;

doca_eth_rxq_create(&eth_rxq_cpu);
/* Populate and start Ethernet receive queue attributes. Set datapath on GPU */
/* Export the Ethernet receive queue CPU handler to a Ethernet receive queue GPU
 handler */
doca_eth_rxq_get_gpu_handle(eth_rxq_cpu, &(eth_rxq_gpu));
/* To use the GPU handler, pass it as parameter of the CUDA kernel */
cuda_kernel<<<...>>>(eth_rxq_gpu, ...);
```

4.1. doca_gpu_mem_type

This enum lists all the possible memory types that can be allocated with GPUNetIO.

```
enum doca_gpu_mem_type {
 DOCA_GPU_MEM_GPU            = 0,
 DOCA_GPU_MEM_GPU_CPU        = 1,
 DOCA_GPU_MEM_CPU            = 2,
 DOCA_GPU_MEM_CPU_GPU        = 3,
};
```

Note: With regards to the syntax, the text string after the `DOCA_GPU_MEM` prefix signifies `<where-memory-resides>_<who-has-access>`.

`DOCA_GPU_MEM_GPU`
Memory resides on the GPU and is accessible from the GPU only.

`DOCA_GPU_MEM_GPU_CPU`
Memory resides on the GPU and is accessible also by the CPU.

`DOCA_GPU_MEM_CPU`
Memory resides on the CPU and is accessible from the CPU only.

`DOCA_GPU_MEM_CPU_GPU`
Memory resides on the CPU and is accessible also by the GPU.

Typical usage of the `DOCA_GPU_MEM_GPU_CPU` memory type is to send a notification from the CPU to the GPU (e.g., a CUDA kernel periodically checking to see if the exit condition set by the CPU is met).

4.2. doca_gpu_create

This is the first function a GPUNetIO application must invoke to create an handler on a GPU device. The function initializes a pointer to a structure in memory with type `struct doca_gpu *`.

```
doca_error_t doca_gpu_create(const char *gpu_bus_id, struct doca_gpu **gpu_dev);
```

`gpu_bus_id`
`<PCIe-bus>:<device>.<function>` of the GPU device you want to use in your application.

`gpu_dev [out]`
GPUNetIO handler to that GPU device.

To get the PCIe address, users can use the commands `lspci` or `nvidia-smi`.

4.3. doca_gpu_mem_alloc

This CPU function allocates different flavors of memory.

```
doca_error_t doca_gpu_mem_alloc(struct doca_gpu *gpu_dev, size_t size, size_t
 alignment, enum doca_gpu_mem_type mtype, void **memptr_gpu, void **memptr_cpu)
```

gpu_dev
: GPUNetIO device handler.

size
: Size, in bytes, of the memory area to allocate.

alignment
: Memory address alignment to use. If 0, default one will be used.

mtype
: Type of memory to allocate.

memptr_gpu [out]
: GPU pointer to use to modify that memory from the GPU if memory is allocated on or is visible by the GPU.

memptr_cpu [out]
: CPU pointer to use to modify that memory from the CPU if memory is allocated on or is visible by the CPU. Can be NULL if memory is GPU-only.

WARNING: Make sure to use the right pointer on the right device! If an application tries to access the memory using the `memptr_gpu` address from the CPU, a segmentation fault will result.

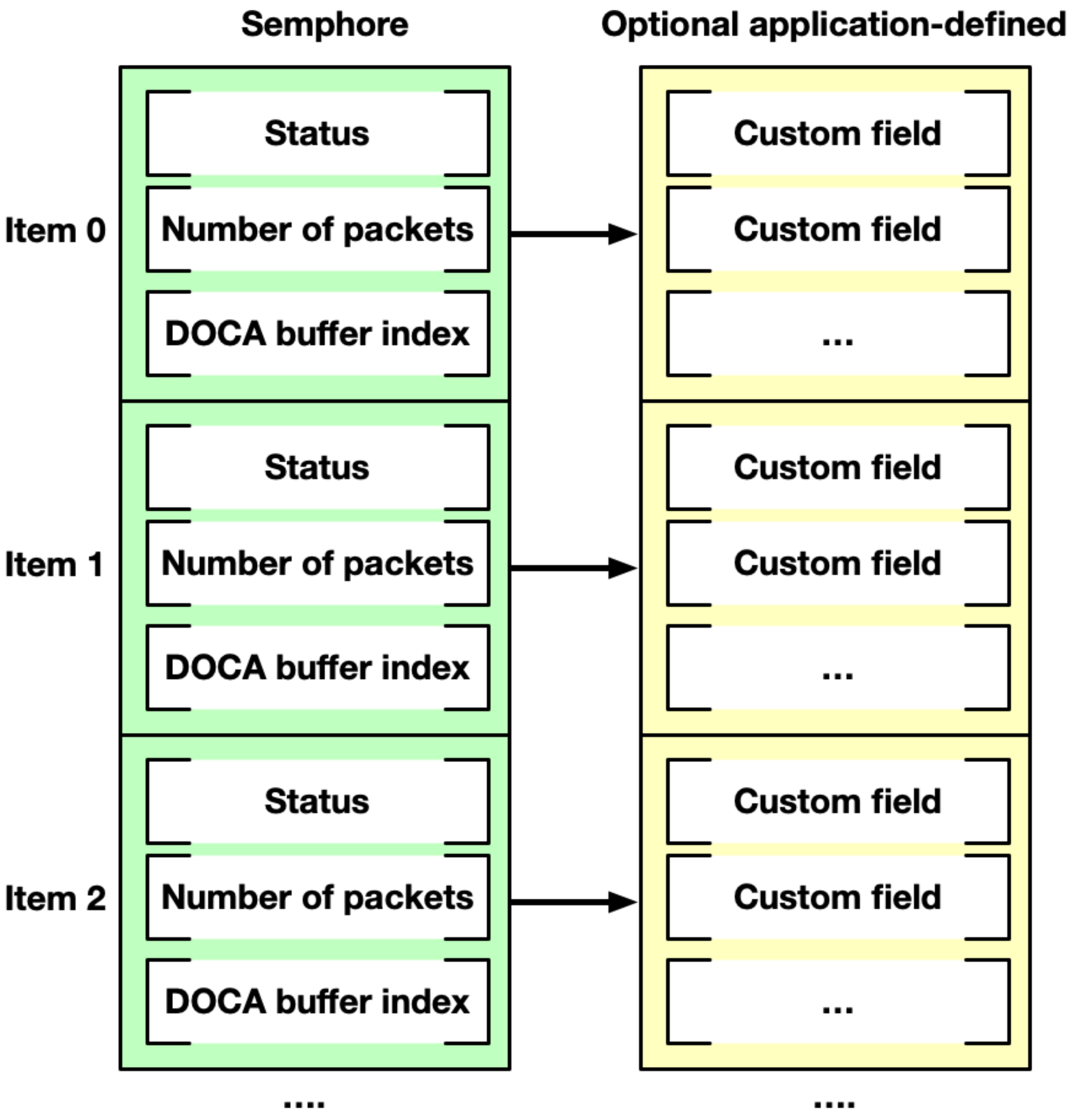
4.4. doca_gpu_semaphore_create

Creates a new instance of a DOCA GPUNetIO semaphore. A semaphore is composed by a list of items each having, by default, a status flag, number of packets, and the index of a `doca_gpu_buf` in a `doca_gpu_buf_arr`.

For example, a GPUNetIO semaphore can be used in applications where a CUDA kernel is responsible for receiving packets in a `doca_gpu_buf_arr` array associated with an Ethernet receive queue object, `doca_gpu_eth_rxq` (see doca_gpu_dev_eth_rxq_receive_*), and dispatching packet info to a second CUDA kernel which processes them.

Another way to use a GPUNetIO semaphore is to exchange data across different entities like two CUDA kernels or a CUDA kernel and a CPU thread. The reason for this scenario may be that the CUDA kernel needs to provide the outcome of the packet processing to the CPU which would in turn compile a statistics report. Therefore, it is possible to associate a custom application-defined structure to each item in the semaphore. This way, the semaphore can be used as a message passing object.

Both situations are illustrated under Receive and Process.



Entities communicating through a semaphore must adopt a poll/update mechanism according to the following logic:

- Update:
  1. Populate the next item of the semaphore (packets' info and/or custom application-defined info).
  2. Set status flag to READY.
- Poll:
  1. Wait for the next item to have a status flag equal to `READY`.
  2. Read and process info.

3. Set status flag to `DONE`.

```
doca_error_t doca_gpu_semaphore_create(struct doca_gpu *gpu_dev, struct
 doca_gpu_semaphore **semaphore)
```

`gpu_dev`
GPUNetIO handler.

`semaphore [out]`
GPUNetIO semaphore handler associated to the GPU device.

4.5. doca_gpu_semaphore_set_memory_type

This function defines the type of memory for the semaphore allocation.

```
doca_error_t doca_gpu_semaphore_set_memory_type(struct doca_gpu_semaphore
 *semaphore, enum doca_gpu_mem_type mtype)
```

`semaphore`
GPUNetIO semaphore handler.

`mtype`
Type of memory to allocate the custom info structure.

- If the application must share packet info only across CUDA kernels, then `DOCA_GPU_MEM_GPU` is the suggested memory type.
- If the application must share info from a CUDA kernel to a CPU (e.g., to report some or output of the pipeline computation) then `DOCA_GPU_MEM_CPU_GPU` is the suggested memory type.

4.6. doca_gpu_semaphore_set_items_num

This function defines the number of items in a semaphore.

```
doca_error_t doca_gpu_semaphore_set_items_num(struct doca_gpu_semaphore *semaphore,
 uint32_t num_items)
```

`semaphore`
GPUNetIO semaphore handler.

`num_items`
Number of items to allocate.

4.7. doca_gpu_semaphore_set_custom_info

This function associates an application-specific structure to semaphore items as explained under doca_gpu_semaphore_create.

```
doca_error_t doca_gpu_semaphore_set_custom_info(struct doca_gpu_semaphore
 *semaphore, uint32_t nbytes, enum doca_gpu_mem_type mtype)
```

`semaphore`
GPUNetIO semaphore handler.

`nbytes`
Size of the custom info structure to associate.

mtype
Type of memory to allocate the custom info structure.

- If the application must share packet info only across CUDA kernels, then `DOCA_GPU_MEM_GPU` is the suggested memory type.
- If the application must share info from a CUDA kernel to a CPU (e.g., to report statistics or output of the pipeline computation) then `DOCA_GPU_MEM_CPU_GPU` is the suggested memory type.

4.8. doca_gpu_semaphore_get_status

From the CPU, query the status of a semaphore item. If the semaphore is allocated with `DOCA_GPU_MEM_GPU`, this function results in a segmentation fault.

```
doca_error_t doca_gpu_semaphore_get_status(struct doca_gpu_semaphore *semaphore_cpu,
 uint32_t idx, enum doca_gpu_semaphore_status *status)
```

semaphore_cpu
GPUNetIO semaphore CPU handler.

idx
Semaphore item index.

status [out]
Output semaphore status.

4.9. doca_gpu_semaphore_get_custom_info_a

From the CPU, retrieve the address of the custom info structure associated to a semaphore item. If the semaphore or the custom info is allocated with `DOCA_GPU_MEM_GPU` this function results in a segmentation fault.

```
doca_error_t doca_gpu_semaphore_get_custom_info_addr(struct doca_gpu_semaphore
 *semaphore_cpu, uint32_t idx, void **custom_info)
```

semaphore_cpu
GPUNetIO semaphore CPU handler.

idx
Semaphore item index.

custom_info [out]
Output semaphore custom info address.

4.10. doca_gpu_dev_eth_rxq_receive_*

To acquire packets in a CUDA kernel, DOCA GPUNetIO offers different flavors of the receive function for different scopes: per CUDA block, per CUDA warp, and per CUDA thread.

```
__device__ doca_error_t doca_gpu_dev_eth_rxq_receive_block(struct doca_gpu_eth_rxq
 *eth_rxq, uint32_t max_rx_pkts, uint64_t timeout_ns, uint32_t *num_rx_pkts,
 uint64_t *doca_gpu_buf_idx)
__device__ doca_error_t doca_gpu_dev_eth_rxq_receive_warp(struct doca_gpu_eth_rxq
 *eth_rxq, uint32_t max_rx_pkts, uint64_t timeout_ns, uint32_t *num_rx_pkts,
 uint64_t *doca_gpu_buf_idx)
```

```
__device__ doca_error_t doca_gpu_dev_eth_rxq_receive_thread(struct doca_gpu_eth_rxq
 *eth_rxq, uint32_t max_rx_pkts, uint64_t timeout_ns, uint32_t *num_rx_pkts,
 uint64_t *doca_gpu_buf_idx)
```

`eth_rxq`
: Ethernet receive queue GPU handler.

`max_rx_pkts`
: Maximum number of packets allowed.

`timeout_ns`
: Nanoseconds to wait for packets before returning.

`num_rx_pkts [out]`
: Effective number of received packets.

`doca_gpu_buf_idx [out]`
: DOCA buffer index of the first packet received in this function.

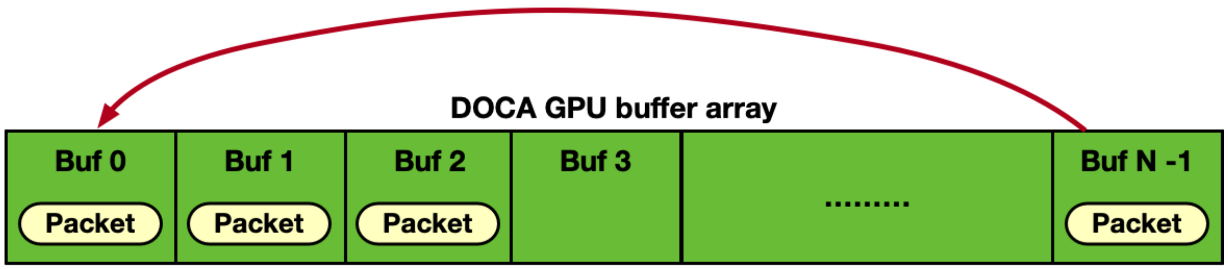
CUDA threads in the same scope (thread, warp, or block) must invoke the function on the same receive queue. The output parameters `num_rx_pkts` and `doca_gpu_buf_idx` must be visible by all threads in the scope (e.g., CUDA shared memory for warp and block).

Each packet received by this function goes to the `doca_gpu_buf_arr` internally created and associated with the Ethernet queues (see Building Blocks).

The function exits when `timeout_ns` is reached or when the maximum number of packets is received.

Note: For CUDA block scope, the block invoking the receive function must have at least 32 CUDA threads (i.e., one warp).

The output parameters indicate how many packets have been actually received (`num_rx_pkts`) and the index of the first received packet in the `doca_gpu_buf_array` internally associated with the Ethernet receive queue. Packets are stored consecutively in the `doca_gpu_buf_arr` so if the function returns `num_rx_pkts=N` and `doca_gpu_buf_idx=X`, this means that all the `doca_gpu_buf` in the `doca_gpu_buf_arr` within the range `[X, .. ,X + (N-1)]` have been filled with packets.



The DOCA buffer array is treated in a circular fashion so that once the last DOCA buffer is filled by a packet, the queue circles back to the first DOCA buffer. There is no need for the application to lock or free `doca_gpu_buf_arr` buffers.

Note: It is the application's responsibility to consume packets before they are overwritten when circling back, properly dimensioning the DOCA buffer array size and scaling across multiple receive queues.

4.11. doca_gpu_dev_eth_txq_send_*

To send packets from a CUDA kernel, DOCA GPUNetIO offers a strong and weak modes for enqueuing a packet in the Ethernet txq. For both modes, the scope is the single CUDA thread, each populating and enqueuing a different `doca_gpu_buf` from a `doca_gpu_buf_arr`.

Strong mode:

Tip: It is generally recommended to use strong mode as weak mode is more complex and is reserved for expert users.

```
__device__ doca_error_t doca_gpu_dev_eth_txq_send_enqueue_strong(struct
 doca_gpu_eth_txq *eth_txq, const struct doca_gpu_buf *buf_ptr, const uint32_t
 nbytes)
```

`eth_txq`
: Ethernet send queue GPU handler.

`buf_ptr`
: DOCA buffer from a DOCA GPU buffer array to be sent.

`nbytes`
: Number of bytes to be sent in the packet.

Weak mode:

Note: In weak mode, the developer must specify a queue descriptor number for where to enqueue the packet ensuring that no descriptor in the queue is left empty wrapping at a 16-bit mask.

```
__device__ doca_error_t doca_gpu_dev_eth_txq_send_enqueue_weak(const struct
 doca_gpu_eth_txq *eth_txq, const struct doca_gpu_buf *buf_ptr, const uint32_t
 nbytes, const uint32_t ndescr)
```

`eth_txq`
: Ethernet send queue GPU handler.

`buf_ptr`
: DOCA buffer from a DOCA GPU buffer array to be sent.

`nbytes`
: Number of bytes to be sent in the packet.

`ndescr`
: Position in the queue to place the packet. Range: 0-0xFFFF.

4.12. doca_gpu_dev_eth_txq_wait_*

To enable Accurate Send Scheduling, the "wait on time" barrier (based on timestamp) must be set in the send queue before enqueuing more packets. Similarly to doca_gpu_dev_eth_txq_send_*, `doca_gpu_dev_eth_txq_wait_*` also has a strong and weak mode.

Strong mode:

Tip: It is generally recommended to use strong mode as weak mode is more complex and is reserved for expert users.

```
__device__ doca_error_t doca_gpu_dev_eth_txq_wait_time_enqueue_strong(struct
 doca_gpu_eth_txq *eth_txq, const uint64_t wait_on_time_value)
```

`eth_txq`
Ethernet send queue GPU handler.

`wait_on_time_value`
Timestamp to specify when packets must be sent after this barrier.

Weak mode:

Note: In weak mode, the developer must specify a queue descriptor number for where to enqueue the packet ensuring that no descriptor in the queue is left empty wrapping at a 16-bit mask.

```
__device__ doca_error_t doca_gpu_dev_eth_txq_wait_time_enqueue_weak(struct
 doca_gpu_eth_txq *eth_txq, const uint64_t wait_on_time_value, const uint32_t
 ndescr)
```

`eth_txq`
Ethernet send queue GPU handler.

`wait_on_time_value`
Timestamp to specify when packets must be sent after this barrier.

`ndescr`
Position in the queue to place the packet. Range: 0-0xFFFF.

Please refer to GPUNetIO Samples to understand how to enable and use Accurate Send Scheduling.

4.13. doca_gpu_dev_eth_txq_commit_*

After enqueuing all the packets to be sent and time barriers, a commit function must be invoked on the txq queue. The right commit function must be used according to the type of enqueue mode (i.e., strong or weak) used in doca_gpu_dev_eth_txq_send_* and doca_gpu_dev_eth_txq_wait_*.

Strong mode:

```
__device__ doca_error_t doca_gpu_dev_eth_txq_commit_strong(struct doca_gpu_eth_txq
 *eth_txq)
```

eth_txq
: Ethernet send queue GPU handler.

Weak mode:

```
__device__ doca_error_t doca_gpu_dev_eth_txq_commit_weak(struct doca_gpu_eth_txq
 *eth_txq, const uint32_t descr_num)
```

eth_txq
: Ethernet send queue GPU handler.

descr_num
: Number of queue items enqueued thus far.

Only one CUDA thread in the scope (CUDA block or CUDA warp) can invoke this function on the send queue after a number of enqueue operations. Typical flow is as follows:

1. All threads in the scope enqueue packets in the send queue.
2. Synchronization point.
3. Only one thread in the scope performs the send queue commit.

4.14. doca_gpu_dev_eth_txq_push

After committing, the items in the send queue must be actually pushed to the network card.

```
__device__ doca_error_t doca_gpu_dev_eth_txq_push(struct doca_gpu_eth_txq *eth_txq)
```

eth_txq
: Ethernet send queue GPU handler.

Only one CUDA thread in the scope (CUDA block or CUDA warp) can invoke this function on the send queue after a number of enqueue or commit operations. Typical flow is as follows:

1. All threads in the scope enqueue packets in the send queue.
2. Synchronization point.
3. Only one thread in the scope does the send queue commit.
4. Only one thread in the scope does the send queue push.

Section "Produce and Send" provides an example where the scope is a block (e.g., each CUDA block operates on a different Ethernet send queue).

Chapter 5. Building Blocks

This sections explains general concepts behind the fundamental building blocks to use when creating a DOCA GPUNetIO application.

5.1. Initialize GPU and NIC

When DOCA GPUNetIO is used in combination with the NIC to send or receive Ethernet traffic, the following must be performed to properly set up the application and devices:

```
uint16_t dpdk_port_id;
struct doca_dev *ddev;
struct doca_gpu *gdev;
char *eal_param[3] = {"", "-a", "00:00.0"};

/* Initialize DPDK with empty device. DOCA device will hot-plug the network card
 later. */
rte_eal_init(3, eal_param);
/* Create DOCA device on a specific network card */
doca_dpdk_port_probe(&ddev);
get_dpdk_port_id_doca_dev(&ddev, &dpdk_port_id);
/* Create GPUNetIO handler on a specific GPU */
doca_gpu_create(gpu_pcie_address, &gdev);
```

The application would may have to enable different items depending on the task at hand.

5.2. Ethernet Receive Queue

If the DOCA application must receive Ethernet packets, receive queues must be created. The receive queue works in a circular way: At creation time, each receive queue is associated with a DOCA buffer array allocated on the GPU by the application. Each DOCA buffer of the buffer array has a maximum fixed size.

```
/* Start DPDK device */
rte_eth_dev_start(dpdk_port_id);
/* Initialise DOCA Flow */
struct doca_flow_port_cfg port_cfg;
port_cfg.port_id = port_id;
doca_flow_init(port_cfg);
doca_flow_port_start();

struct doca_eth_rxq *eth_rxq_cpu;
struct doca_gpu_eth_rxq *eth_rxq_gpu;
struct doca_mmap *mmap;
void *gpu_buffer;
```

```
/* Create DOCA Ethernet receive queues */
doca_eth_rxq_create(&eth_rxq_cpu);

/* Set Ethernet receive queue properties */
/* ... */

/* Create DOCA mmap in GPU memory to be used for the DOCA buffer array associated to
 this Ethernet queue */
doca_mmap_create(&mmap);
doca_gpu_mem_alloc(gdev, buffer_size, alignment, DOCA_GPU_MEM_GPU, (void
 **)&gpu_buffer, NULL);
doca_mmap_start(mmap);
doca_eth_rxq_set_pkt_buffer(eth_rxq_cpu, mmap, 0, buffer_size);

/* Start the Ethernet queue object */

/* Export GPU handle for the receive queue */
doca_eth_rxq_get_gpu_handle(eth_rxq_cpu, &eth_rxq_gpu);
```

It is mandatory to associate DOCA Flow pipe(s) to the receive queues. Otherwise, the application cannot receive any packet.

5.3. Ethernet Send Queue

If the DOCA application must send Ethernet packets, send queues must be created in combination with `doca_gpu_buf_arr` to prepare and send packets from GPU memory.

```
struct doca_eth_txq *eth_txq_cpu;
struct doca_gpu_eth_txq *eth_txq_gpu;

/* Create DOCA Ethernet send queues */
doca_eth_txq_create(&eth_txq_cpu);
/* Set properties to send queues */
/* Export GPU handle for the send queue */
doca_eth_rxq_get_gpu_handle(eth_txq_cpu, &eth_txq_gpu);

/* Create DOCA mmap to define memory layout and type for the DOCA buf array */
struct doca_mmap *mmap;
doca_mmap_create(&mmap);
/* Set DOCA mmap properties */

/* Create DOCA buf arr and export it to GPU */
struct doca_buf_arr *buf_arr;
struct doca_gpu_buf_arr *buf_arr_gpu;
doca_buf_arr_create(mmap, &buf_arr);
/* Set DOCA buf array properties */
...
/* Export GPU handle for the buf arr */
doca_buf_arr_get_gpu_handle(buf_arr, &buf_arr_gpu);
```

5.4. Semaphore

If the DOCA application must dispatch some packets' info across CUDA kernels or from the CUDA kernel and some CPU thread, a semaphore must be created.

A semaphore is a list of items, allocated either on the GPU or CPU (depending on the use case) visible by both the GPU and CPU. This object can be used to discipline communication across items in the GPU pipeline between CUDA kernels or a CUDA kernel and a CPU thread.

By default, each semaphore item can hold info about its status (`FREE`, `READY`, `HOLD`, `DONE`, `ERROR`), the number of received packets, and an index of a `doca_gpu_buf` in a `doca_gpu_buf_arr`.

If the semaphore must be used to exchange data with the CPU, a preferred memory layout would be `DOCA_GPU_MEM_CPU_GPU`. Whereas, if the semaphore is only needed across CUDA kernels, `DOCA_GPU_MEM_GPU` is the best memory layout to use.

As an optional feature, if the application must pass more application-specific info through the semaphore items, it is possible to attach a custom structure to each item of the semaphore.

```
# Define SEMAPHORE_ITEMS 1024

/* Application defined custom structure to pass info through semaphore items */
struct custom_info {
 int a;
 uint64_t b;
};

/* Semaphore to share info from the GPU to the CPU */
struct doca_gpu_semaphore *sem_to_cpu;
struct doca_gpu_semaphore_gpu *sem_to_cpu_gpu;

doca_gpu_semaphore_create(gdev, &sem_to_cpu);
doca_gpu_semaphore_set_memory_type(sem_to_cpu, DOCA_GPU_MEM_CPU_GPU);
doca_gpu_semaphore_set_items_num(sem_to_cpu, SEMAPHORE_ITEMS);
/* This is optional */
doca_gpu_semaphore_set_custom_info(sem_to_cpu, sizeof(struct custom_info),
 DOCA_GPU_MEM_CPU_GPU);
doca_gpu_semaphore_start(sem_to_cpu);
doca_gpu_semaphore_get_gpu_handle(sem_to_cpu, &sem_to_cpu_gpu);

/* Semaphore to share info across GPU CUDA kernels with no CPU involvment */
struct doca_gpu_semaphore *sem_to_gpu;
struct doca_gpu_semaphore_gpu *sem_to_gpu_gpu;

doca_gpu_semaphore_create(gdev, &sem_to_gpu);
doca_gpu_semaphore_set_memory_type(sem_to_gpu, DOCA_GPU_MEM_GPU);
doca_gpu_semaphore_set_items_num(sem_to_gpu, SEMAPHORE_ITEMS);
/* This is optional */
doca_gpu_semaphore_set_custom_info(sem_to_gpu, sizeof(struct custom_info),
 DOCA_GPU_MEM_GPU);
doca_gpu_semaphore_start(sem_to_gpu);
doca_gpu_semaphore_get_gpu_handle(sem_to_gpu, &sem_to_gpu_gpu);
```

5.5. Data Path on GPU

At this point, the application has created and initialized all the objects required by the GPU to exercise the data path to send or receive packets with GPUNetIO. The following subsections provide examples for doing that.

5.5.1. Receive and Process

In this example, the application must receive packets from different queues with a receiver CUDA kernel and dispatch packet info to a second CUDA kernel responsible for packet processing.

The CPU launches the CUDA kernels and waits on the semaphore for output:

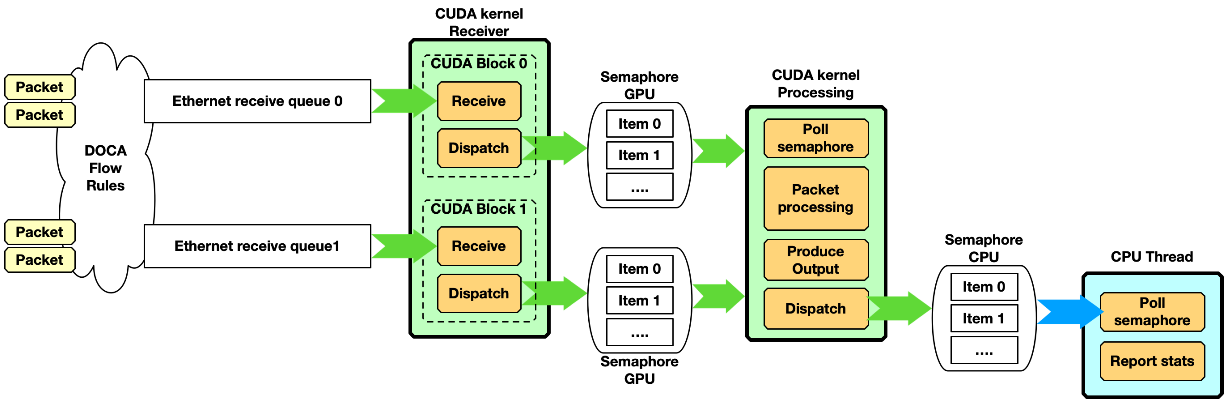
```
#define CUDA_THREADS 512
#define CUDA_BLOCKS 1
int semaphore_index = 0;
enum doca_gpu_semaphore_status status;
struct custom_info *gpu_info;

/* On the CPU */
cuda_kernel_receive_dispatch<<<CUDA_THREADS, CUDA_BLOCKS, ...,
 stream_0>>>(eth_rxq_gpu, sem_to_gpu_gpu)
cuda_kernel_process<<<CUDA_THREADS, CUDA_BLOCKS, ..., stream_1>>>(eth_rxq_gpu,
 sem_to_cpu_gpu, sem_to_gpu_gpu)

while(/* condition */) {
 doca_gpu_semaphore_get_status(sem_to_cpu, semaphore_index, &status);
 if (status == DOCA_GPU_SEMAPHORE_STATUS_READY) {
  doca_gpu_semaphore_get_custom_info_addr(sem_to_cpu, semaphore_index, (void
 **)&(gpu_info));
  report_info(gpu_info);
  doca_gpu_semaphore_set_status(sem_to_cpu, semaphore_index,
 DOCA_GPU_SEMAPHORE_STATUS_FREE);
  semaphore_index = (semaphore_index+1) % SEMAPHORE_ITEMS;
 }
}
```

On the GPU, the two CUDA kernels are running on different streams:

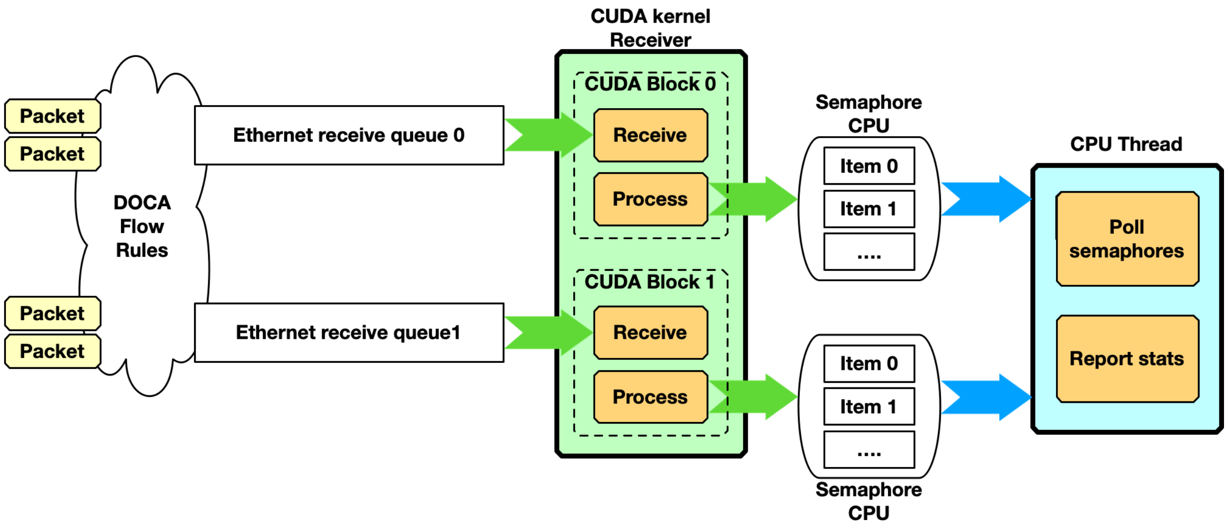
```
cuda_kernel_receive_dispatch(eth_rxq_gpu, sem_to_gpu_gpu) {
 __shared__ uint32_t rx_pkt_num;
 __shared__ uint64_t rx_buf_idx;
 int semaphore_index = 0;

 doca_gpu_dev_eth_rxq_receive_block(eth_rxq_gpu, MAX_NUM_RECEIVE_PACKETS,
 TIMEOUT_RECEIVE_NS, &rx_pkt_num, &rx_buf_idx);
 if (threadIdx.x == 0 && rx_pkt_num > 0) {
  doca_gpu_dev_sem_set_packet_info(sem_to_gpu_gpu, semaphore_index,
 DOCA_GPU_SEMAPHORE_STATUS_READY, rx_pkt_num, rx_buf_idx);
  semaphore_index = (semaphore_index+1) % SEMAPHORE_ITEMS;
 }
}

cuda_kernel_process(eth_rxq_gpu, sem_to_cpu_gpu, sem_to_gpu_gpu) {
 __shared__ uint32_t rx_pkt_num;
 __shared__ uint64_t rx_buf_idx;
 int semaphore_index = 0;
 int thread_buf_idx = 0;
 struct doca_gpu_buf *buf_ptr;
 uintptr_t buf_addr;
 struct custom_info *gpu_info;

 while (/* exit condition */) {
  if (threadIdx.x == 0) {
   do {
    result = doca_gpu_dev_sem_get_packet_info_status(sem_to_gpu_gpu,
 semaphore_index, DOCA_GPU_SEMAPHORE_STATUS_READY, &rx_pkt_num, &rx_buf_idx);
   } while(result != DOCA_ERROR_NOT_FOUND /* && other exit condition */);
  }
  __syncthreads();

  thread_buf_idx = threadIdx.x;
  while (thread_buf_idx < rx_pkt_num) {
   /* Get DOCA GPU buffer from the GPU buffer in the receive queue */
   doca_gpu_dev_eth_rxq_get_buf(eth_rxq_gpu, rx_buf_idx + thread_buf_idx, &buf_ptr);
   /* Get DOCA GPU buffer memory address */
   doca_gpu_dev_buf_get_addr(buf_ptr, &buf_addr);
   /*
    * Atomic here is has the entire CUDA block accesses the same semaphore to CPU.
```

```
     * Smarter implementation can be done at warp level, with multiple semaphores,
 etc.. to avoid this atomic
     */
    int semaphore_index_tmp = atomicAdd_block(&semaphore_index, 1);
    semaphore_index_tmp = semaphore_index_tmp % SEMAPHORE_ITEMS;
    doca_gpu_dev_sem_get_custom_info_addr(sem_to_cpu_gpu, semaphore_index_tmp, (void
 **)&gpu_info);
    populate_custom_info(buf_addr, gpu_info);
    doca_gpu_dev_sem_set_status(sem_to_cpu_gpu, semaphore_index_tmp,
 DOCA_GPU_SEMAPHORE_STATUS_READY);
   }
  }
}
```

This code can be represented with the following diagram when multiple queues and/or semaphores are used:



Please note that receiving and dispatching packets to another CUDA kernel is not required. A simpler scenario can have a single CUDA kernel receiving and processing packets:



The drawback of this approach is that the time between two receives depends on the time taken by the CUDA kernel to process received packets.

The type of pipeline that must be built heavily depends on the specific use case.

5.5.2. Produce and Send

In this example, the GPU produces some data, stores it into packets and then sends them over the network. The CPU launches the CUDA kernels and continues doing other work:

```
#define CUDA_THREADS 512
#define CUDA_BLOCKS 1
int semaphore_index = 0;
enum doca_gpu_semaphore_status status;
struct custom_info *gpu_info;

/* On the CPU */
cuda_kernel_produce_send<<<CUDA_THREADS, CUDA_BLOCKS, ..., stream_0>>>(eth_txq_gpu,
 buf_arr_gpu)

/* do other stuff */
```

On the GPU, the CUDA kernel fills the packets with meaningful data and sends them. In the following example, the scope is CUDA block so each block uses a different DOCA Ethernet send queue:

```
cuda_kernel_produce_send(eth_txq_gpu, buf_arr_gpu) {
    uint64_t doca_gpu_buf_idx = threadIdx.x;
    struct doca_gpu_buf *buf;
    uintptr_t buf_addr;
    uint32_t packet_len;

    while (/* exit condition */) {
        /* Each CUDA thread retrieves doca_gpu_buf from doca_gpu_buf_arr */
        doca_gpu_dev_buf_get_buf(buf_arr_gpu, doca_gpu_buf_idx, &buf);
        /* Get memory address of the packet in the doca_gpu_buf */
        doca_gpu_dev_buf_get_addr(buf, &buf_addr);
        /* Application produces data and crafts the packet in the doca_gpu_buf */
        populate_packet(buf_addr, &packet_len);
        /* Enqueue packet in the send queue */
        doca_gpu_dev_eth_txq_send_enqueue_strong(eth_txq_gpu, buf, packet_len);
        /* Synchronization point */
        __synchthreads();

        /* Only one CUDA thread in the block must commit and push the send queue */
        if (threadIdx.x == 0) {
            doca_gpu_dev_eth_txq_commit_strong(eth_txq_gpu);
            doca_gpu_dev_eth_txq_push(eth_txq_gpu);
        }
        /* Synchronization point */
        __synchthreads();

        /* Assume all threads in the block pushed a packet in the send queue */
        doca_gpu_buf_idx += CUDA_THREADS;
    }
}
```

Chapter 6. GPUNetIO Samples

The sample shows how to enable Accurate Send Scheduling (or wait-on-time) in the context of a GPUNetIO application. Accurate Send Scheduling is the ability of an NVIDIA NIC to send packets in the future according to some application-provided timestamps.

This means that the application can prepare packets and associate to them a timestamp to instruct the NIC on when packets should be sent in the future.

Note: This feature is supported on ConnectX-6 Dx and later.

The DOCA GPUNetIO sample provides a simple application to send packets with Accurate Send Scheduling from the GPU.

6.1. Multi-GPU Environment

If the sample is running in a multi-GPU environment, either choose the GPU to use by setting the `CUDA_VISIBLE_DEVICES` environment variable or add this simple piece of code in the `gpunetio_send_wait_time_main.c` file in the main function right after the `doca_argp_start` function.

```
int cuda_id;
cudaDeviceGetByPCIBusId(&cuda_id, sample_cfg.gpu_pcie_addr);
cudaFree(0);
cudaSetDevice(cuda_id);
```

6.2. Synchronizing Clocks

Before starting the sample, it is important to properly synchronize the CPU clock with the NIC clock. This way, timestamps provided by the system clock are synchronized with the time in the NIC.

For this purpose, at least the `phc2sys` service must be used. To install it on an Ubuntu system:

```
sudo apt install linuxptp
```

To start the `phc2sys` service properly, a config file must be created in `/lib/systemd/system/phc2sys.service`:

```
[Unit]
Description=Synchronize system clock or PTP hardware clock (PHC)
Documentation=man:phc2sys
```

```
[Service]
Restart=always
RestartSec=5s
Type=simple
ExecStart=/bin/sh -c "taskset -c 23 /usr/sbin/phc2sys -s /dev/ptp$(ethtool -T ens6f0
 | grep PTP | awk '{print $4}') -c CLOCK_REALTIME -n 24 -O 0 -R 256 -u 256"

[Install]
WantedBy=multi-user.target
```

Now `phc2sys` service can be started:

```
sudo systemctl stop systemd-timesyncd
sudo systemctl disable systemd-timesyncd
sudo systemctl daemon-reload
sudo systemctl start phc2sys.service
```

To check `phc2sys` status:

```
$ sudo systemctl status phc2sys.service

● phc2sys.service - Synchronize system clock or PTP hardware clock (PHC)
     Loaded: loaded (/lib/systemd/system/phc2sys.service; disabled; vendor preset:
 enabled)
     Active: active (running) since Mon 2023-04-03 10:59:13 UTC; 2 days ago
       Docs: man:phc2sys
   Main PID: 337824 (sh)
      Tasks: 2 (limit: 303788)
     Memory: 560.0K
        CPU: 52min 8.199s
     CGroup: /system.slice/phc2sys.service
             ├─337824 /bin/sh -c "taskset -c 126 /usr/sbin/phc2sys -s /dev/ptp\
$(ethtool -T enp23s0f1np1 | grep PTP | awk '{print \$4}') -c CLOCK_REALTIME -n 24 -O
 0 -R >
             └─337829 /usr/sbin/phc2sys -s /dev/ptp3 -c CLOCK_REALTIME -n 24 -O 0 -R
 256 -u 256

Apr 05 16:35:52 doca-vr-045 phc2sys[337829]: [457395.040] CLOCK_REALTIME rms    8
 max   18 freq +110532 +/-  27 delay   770 +/-   3
Apr 05 16:35:53 doca-vr-045 phc2sys[337829]: [457396.071] CLOCK_REALTIME rms    8
 max   20 freq +110513 +/-  30 delay   769 +/-   3
Apr 05 16:35:54 doca-vr-045 phc2sys[337829]: [457397.102] CLOCK_REALTIME rms    8
 max   18 freq +110527 +/-  30 delay   769 +/-   3
Apr 05 16:35:55 doca-vr-045 phc2sys[337829]: [457398.130] CLOCK_REALTIME rms    8
 max   18 freq +110517 +/-  31 delay   769 +/-   3
Apr 05 16:35:56 doca-vr-045 phc2sys[337829]: [457399.159] CLOCK_REALTIME rms    8
 max   19 freq +110523 +/-  32 delay   770 +/-   3
Apr 05 16:35:57 doca-vr-045 phc2sys[337829]: [457400.191] CLOCK_REALTIME rms    8
 max   20 freq +110528 +/-  33 delay   770 +/-   3
Apr 05 16:35:58 doca-vr-045 phc2sys[337829]: [457401.221] CLOCK_REALTIME rms    8
 max   19 freq +110512 +/-  38 delay   770 +/-   3
Apr 05 16:35:59 doca-vr-045 phc2sys[337829]: [457402.253] CLOCK_REALTIME rms    9
 max   20 freq +110538 +/-  47 delay   770 +/-   4
Apr 05 16:36:00 doca-vr-045 phc2sys[337829]: [457403.281] CLOCK_REALTIME rms    8
 max   21 freq +110517 +/-  38 delay   769 +/-   3
Apr 05 16:36:01 doca-vr-045 phc2sys[337829]: [457404.311] CLOCK_REALTIME rms    8
 max   17 freq +110526 +/-  26 delay   769 +/-   3
...
```

At this point, the system and NIC clocks are synchronized so timestamps provided by the CPU are correctly interpreted by the NIC.

Important: The timestamps you get may not reflect the real time and day. To get that, you must properly set the ptp4l service with an external grand master on the system. Doing that is out of the scope of this sample.

6.3. Running the Sample

The sample is shipped with the source files that must be built:

```
# Ensure DOCA and DPDK are in the pkgconfig environment variable
export PKG_CONFIG_PATH=${PKG_CONFIG_PATH}:/opt/mellanox/dpdk/lib/x86_64-linux-gnu/
pkgconfig:/opt/mellanox/doca/lib/x86_64-linux-gnu/pkgconfig:/opt/mellanox/flexio/
lib/pkgconfig

cd /opt/mellanox/doca/samples/doca_gpunetio/gpunetio_send_wait_time
meson build
ninja -C build
```

The sample sends 8 bursts of 32 raw Ethernet packets or 1kB to a dummy Ethernet address, `10:11:12:13:14:15`, in a timed way. Program the NIC to send every `t` nanoseconds (command line option `-t`).

Considering a system with GPU PCIe address `ca:00.0` and NIC PCIe address `17:00.0` , to send 32 packets every 5 milliseconds:

```
/opt/mellanox/doca/samples/doca_gpunetio_send_wait_time -n 17:00.0 -g ca:00.0 -t
 5000000
```

To verify that packets are actually sent at the right time, use a packet sniffer on the other side (e.g., `tcpdump`):

```
$ sudo tcpdump -i enp23s0f1np1 -A -s 64

17:12:23.480318 IP5 (invalid)
Sent from DOCA GPUNetIO..........................
....
17:12:23.480368 IP5 (invalid)
Sent from DOCA GPUNetIO..........................
# end of first burst of 32 packets, bump to +5ms
17:12:23.485321 IP5 (invalid)
Sent from DOCA GPUNetIO..........................
...
17:12:23.485369 IP5 (invalid)
Sent from DOCA GPUNetIO..........................
# end of second burst of 32 packets, bump to +5ms
17:12:23.490278 IP5 (invalid)
Sent from DOCA GPUNetIO..........................
...
```

The output should show a jump of approximately 5 milliseconds every 32 packets. Please note `tcpdump` may increase latency in sniffing packets and reporting the receive timestamp, so the difference between bursts of 32 packets reported may be less than expected, especially with small interval times like 500 microseconds (`-t 500000`).

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