DOCA GPUNetIO

Table of contents

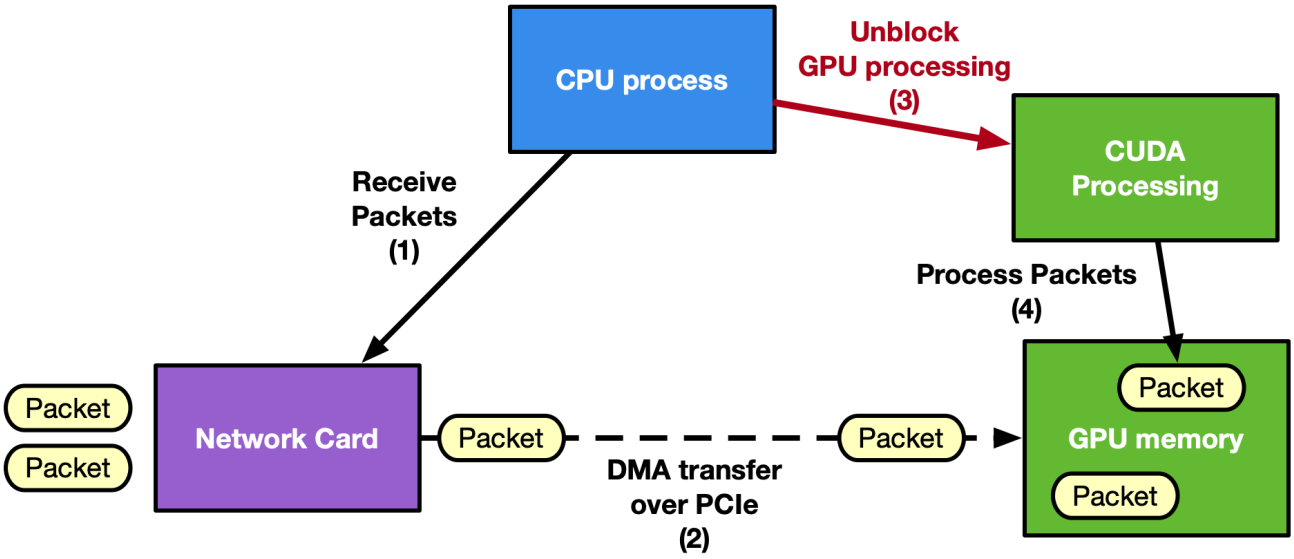
This document provides an overview and configuration instructions for DOCA GPUNetIO API.

Introduction

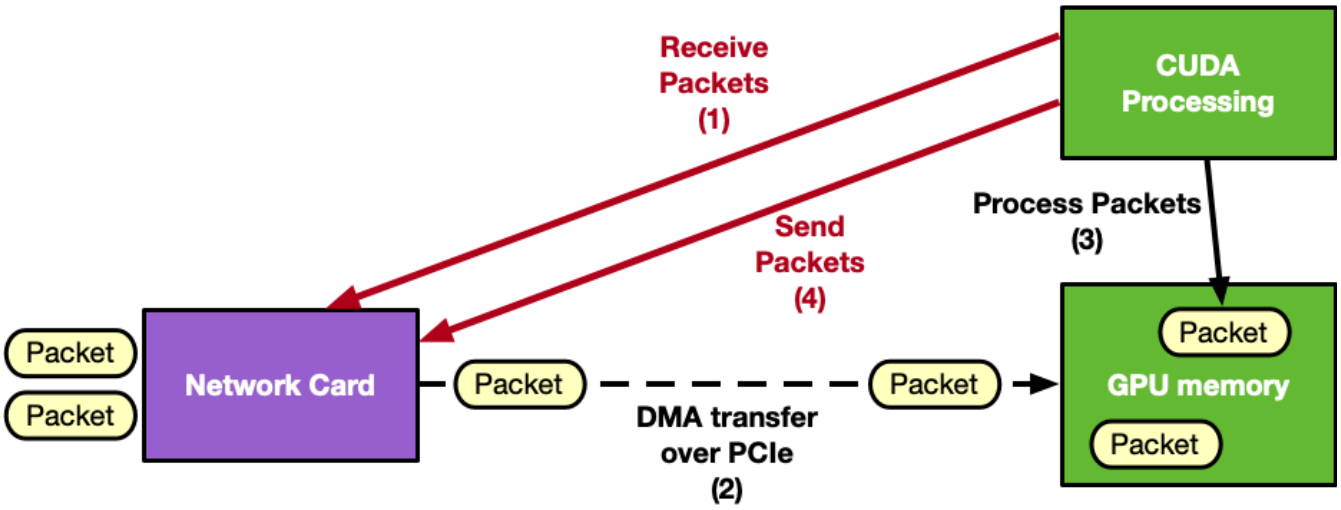
Real-time GPU processing of network packets is a technique useful for application domains involving signal processing, network security, information gathering, input reconstruction, and more. These applications involve the CPU in the critical path (CPU-centric approach) to coordinate the network card (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 network (GDAKIN) 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 technology – a GPU CUDA kernel can directly control other hardware components like the network card or NVIDIA® BlueField®'s DMA engine
    - GDAKIN communications – a GPU CUDA kernel can control network communications to send or receive data
        - GPU can control Ethernet communications
        - GPU can control RDMA communications (InfiniBand or RoCE are supported)
        - CPU intervention is not needed in the application critical path
    - DMA engine – a GPU CUDA kernel can trigger a memory copy using BlueField's DMA engine
- GPUDirect RDMA – use a contiguous GPU memory to send or receive RDMA data or Ethernet packets without CPU memory staging copies
- Semaphores – provide a standardized low-latency message passing protocol between two CUDA kernels or a CUDA kernel and a CPU thread
- Smart memory allocation – allocate aligned GPU memory buffers, possibly exposing them to direct CPU access
    - Combination of CUDA, DPDK gpudev library and GDRCopy library already embedded in the DPDK released with DOCA

- Accurate send scheduling – schedule Ethernet packets' send in the future according to a user-provided timestamp

Aerial 5G SDK, Morpheus, and Holoscan Advanced Network Operator are examples of NVIDIA applications actively using DOCA GPUNetIO.

For a deep dive into the technology and motivations, please refer to the NVIDIA blog posts Inline GPU Packet Processing with NVIDIA DOCA GPUNetIO and Unlocking GPU-Accelerated RDMA with NVIDIA DOCA GPUNetIO. Another NVIDIA blog post Realizing the Power of Real-time Network Processing with NVIDIA DOCA GPUNetIO has been published to provide more use-case examples where DOCA GPUNetIO has been useful to improve the execution.

> ⚠ **Warning**
>
> RDMA on DOCA GPUNetIO is currently supported at alpha level.

Changes From Previous Releases

Changes in 2.10

The following section details the `doca_gpunetio` library updates in version 2.10.

- Removed dependency on DPDK – So far, the `rte_eal_init` function was required before calling `doca_gpu_create`. This is not needed anymore in Ethernet or RDMA app/sample.

System Configuration

DOCA GPUNetIO requires a properly configured environment which depends on whether the application should run on the x86 host or DPU Arm cores. The following subsections describe the required configuration in both scenarios, assuming DOCA, CUDA Toolkit and NVIDIA driver are installed on the system (x86 host or BlueField Arm) where the DOCA GPUNetIO is built and executed.

DOCA GPUNetIO is available for all DOCA for host and BFB packages downloadable here.

Assuming the DOCA package has been downloaded and the prerequisites listed below have been satisfied, to install DOCA GPUNetIO components, run:

- For Ubuntu/Debian:

```
apt install doca-all doca-sdk-gpunetio libdoca-sdk-gpunetio-dev
```

- For RHEL:

```
yum install doca-all doca-sdk-gpunetio doca-sdk-gpunetio-devel
```

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

Note

To achieve the best performance, when building any DOCA GPUNetIO sample or application, set the `buildtype` to `release` instead of `debug` in the `meson.build` file.

As DOCA GPUNetIO is present in both DOCA-for-Host and DOCA BFB (for BlueField Arm), a GPUNetIO application can be executed either on the host CPU or on the BlueField's Arm cores. The following subsections provide a description of both scenarios.

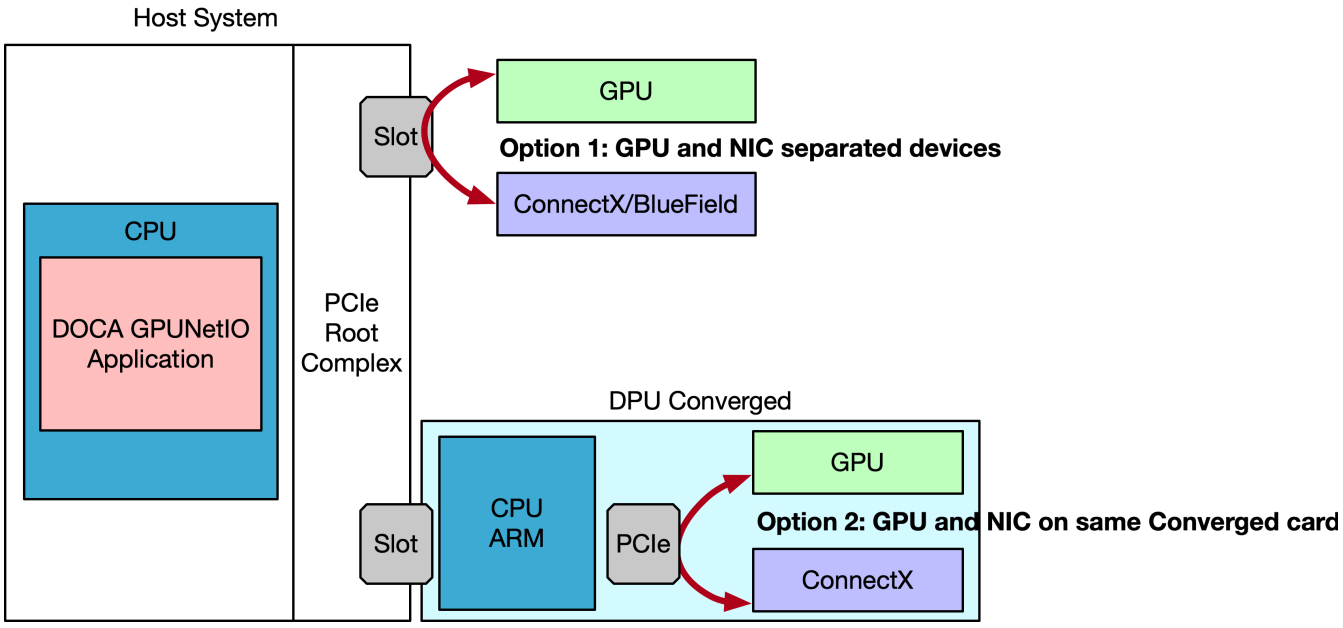
Note

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

Application on Host CPU

Assuming the DOCA GPUNetIO application is running on the host x86 CPU cores, it is highly recommended to have a dedicated PCIe connection between the GPU and the NIC. This topology can be realized in two ways:

- Adding an additional PCIe switch to one of the PCIe root complex slots and attaching to this switch a GPU and a NVIDIA® ConnectX® adapter
- Connecting an NVIDIA® Converged Accelerator DPU to the PCIe root complex and setting it to NIC mode (i.e., exposing the GPU and NIC devices to the host)



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

Option 1: ConnectX Adapter in Ethernet Mode

Note

NVIDIA® ConnectX® firmware must be 22.36.1010 or later. It is highly recommended to only use NVIDIA adapter from ConnectX-6 Dx and later.

DOCA GPUNetIO allows a CUDA kernel to control the NIC when working with Ethernet protocol. For this reason, the ConnectX must be set to Ethernet mode.

To do that, follow these steps:

1. Start MST, check the status, and copy the MST device name:

```
# 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
```

2. Configure the NIC to Ethernet mode and enable Accurate Send Scheduling (if required on the send side):

Info

The following example assumes that the adapter is dual-port. If single port, only P1 options apply.

```
mlxconfig -d <mst_device> 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 <mst_device> --yes set ACCURATE_TX_SCHEDULER=1
REAL_TIME_CLOCK_ENABLE=1
```

3. Perform cold reboot to apply the configuration changes:

```
ipmitool power cycle
```

Option 2: DPU Converged Accelerator in NIC mode

To expose and use the GPU and the NIC on the converged accelerator DPU to an application running on the Host x86, configure the DPU to operate in NIC mode.

To do that, follow these steps:

Info

Valid for both NVIDIA® BlueField®-2 and NVIDIA® BlueField®-3 converged accelerator DPUs.

1. Start MST, check the status, and copy the MST device name:

```
# 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
```

2. Expose the GPU on the converged accelerator DPU to the host.
   - For BlueField-2, the `PCI_DOWNSTREAM_PORT_OWNER` offset must be set to 4:

     ```
     sudo mlxconfig -d <mst_device> --yes s
     PCI_DOWNSTREAM_PORT_OWNER[4]=0x0
     ```

   - For BlueField-3, the `PCI_DOWNSTREAM_PORT_OWNER` offset must be set to 8:

     ```
     sudo mlxconfig -d <mst_device> --yes s
     PCI_DOWNSTREAM_PORT_OWNER[8]=0x0
     ```

3. Set BlueField to Ethernet mode, enable Accurate Send Scheduling (if required on the send side), and set it to NIC mode:

   ```
   sudo mlxconfig -d <mst_device> --yes set LINK_TYPE_P1=2
   LINK_TYPE_P2=2 INTERNAL_CPU_MODEL=1
   ```

```
INTERNAL_CPU_PAGE_SUPPLIER=1 INTERNAL_CPU_ESWITCH_MANAGER=1
INTERNAL_CPU_IB_VPORT0=1 INTERNAL_CPU_OFFLOAD_ENGINE=DISABLED
sudo mlxconfig -d <mst_device> --yes set ACCURATE_TX_SCHEDULER=1
REAL_TIME_CLOCK_ENABLE=1
```

4. Perform cold reboot to apply the configuration changes:

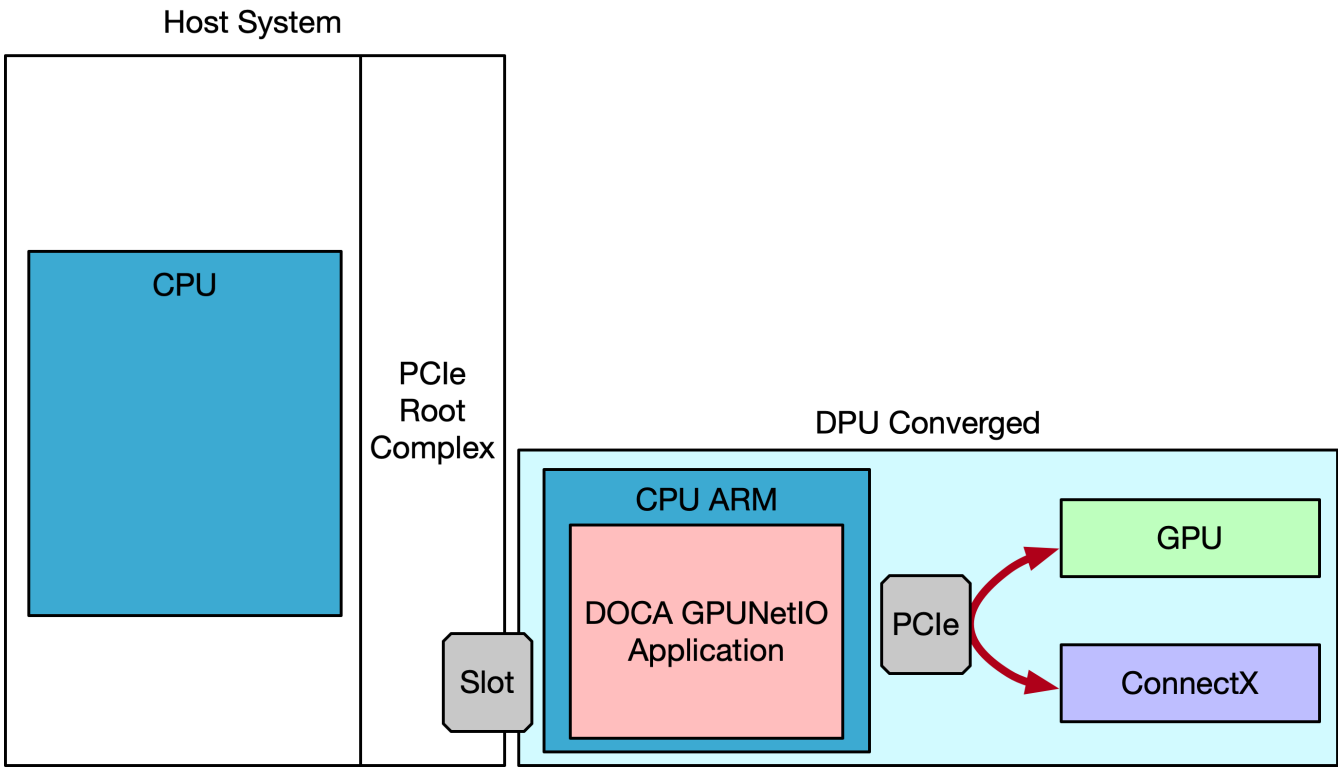
```
ipmitool power cycle
```

5. Verify configuration:

```
sudo mlxconfig -d <mst_device> 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)
```

Application on BlueField Converged Arm CPU

In this scenario, the DOCA GPUNetIO is running on the CPU Arm cores of the BlueField using the GPU and NIC on the same BlueField.



The converged accelerator DPU must be set to CPU mode after flashing the right BFB image (refer to DOCA Installation Guide for Linux for details). From the x86 host, configure the DPU as detailed in the following steps:

> **Info**
>
> Valid for both BlueField-2 and BlueField-3 converged accelerator DPUs.

1. Start MST, check the status, and copy the MST device name:

```
# 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
```

2. Set the DPU as the GPU owner.

    1. For BlueField-2 the `PCI_DOWNSTREAM_PORT_OWNER` offset must be set to 4:

    ```
    sudo mlxconfig -d <mst_device> --yes s
    PCI_DOWNSTREAM_PORT_OWNER[4]=0xF
    ```

    2. For BlueField-3 the `PCI_DOWNSTREAM_PORT_OWNER` offset must be set to 8:

    ```
    sudo mlxconfig -d <mst_device> --yes s
    PCI_DOWNSTREAM_PORT_OWNER[8]=0xF
    ```

3. Set BlueField to Ethernet mode and enable Accurate Send Scheduling (if required on the send side):

```
sudo mlxconfig -d <mst_device> --yes set LINK_TYPE_P1=2
LINK_TYPE_P2=2 INTERNAL_CPU_MODEL=1
```

```
INTERNAL_CPU_PAGE_SUPPLIER=0 INTERNAL_CPU_ESWITCH_MANAGER=0
INTERNAL_CPU_IB_VPORT0=0 INTERNAL_CPU_OFFLOAD_ENGINE=ENABLED
sudo mlxconfig -d <mst_device> --yes set ACCURATE_TX_SCHEDULER=1
REAL_TIME_CLOCK_ENABLE=1
```

4. Perform cold reboot to apply the configuration changes:

```
ipmitool power cycle
```

5. Verify configuration:

```
mlxconfig -d <mst_device> 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
...
Configurations:                                      Next
Boot
        LINK_TYPE_P1
ETH(2)
        LINK_TYPE_P2
ETH(2)
        INTERNAL_CPU_MODEL
EMBEDDED_CPU(1)
        INTERNAL_CPU_PAGE_SUPPLIER
ECPF(0)
        INTERNAL_CPU_ESWITCH_MANAGER
ECPF(0)
        INTERNAL_CPU_IB_VPORT0
ECPF(0)
        INTERNAL_CPU_OFFLOAD_ENGINE
ENABLED(0)
```

```
        ACCURATE_TX_SCHEDULER
True(1)
        REAL_TIME_CLOCK_ENABLE
True(1)
```

At this point, it should be possible to SSH into BlueField to access the OS installed on it. Before installing DOCA GPUNetIO as previously described, CUDA Toolkit (and NVIDIA driver) must be installed.

PCIe Configuration

On some x86 systems, the Access Control Services (ACS) must be disabled to ensure direct communication between the NIC and GPU, whether they reside on the same converged accelerator DPU or on different PCIe slots in the system. The recommended solution is to disable ACS control via BIOS (e.g., Supermicro or HPE) on your PCIe bridge. Alternatively, it is also possible to disable it via command line, but it may not be as effective as the BIOS option. Assuming system topology Option 2, with a converged accelerator DPU as follows:

```
$ lspci -tvvv...+-[0000:b0]-+-00.0  Intel Corporation Device 09a2
 |           +-00.1  Intel Corporation Device 09a4
 |           +-00.2  Intel Corporation Device 09a3
 |           +-00.4  Intel Corporation Device 0998
 |           \-02.0-[b1-b6]----00.0-[b2-b6]--+-00.0-[b3]--+-00.0
Mellanox Technologies MT42822 BlueField-2 integrated ConnectX-6
Dx network controller
 |                                          |            +-00.1
Mellanox Technologies MT42822 BlueField-2 integrated ConnectX-6
Dx network controller
 |                                          |            \-00.2
Mellanox Technologies MT42822 BlueField-2 SoC Management
Interface
 |                                         \-01.0-[b4-b6]---
-00.0-[b5-b6]----08.0-[b6]----00.0  NVIDIA Corporation Device
```

```
20b8
```

The PCIe switch address to consider is `b2:00.0` (entry point of the DPU). ACSCtl must have all negative values:

```
setpci -s b2:00.0 ECAP_ACS+0x6.w=0000
```

To verify that the setting has been applied correctly:

```
$ sudo lspci -s b2:00.0 -vvvv | grep -i ACSCtl
ACSCtl: SrcValid- TransBlk- ReqRedir- CmpltRedir- UpstreamFwd-
EgressCtrl- DirectTrans-
```

Please refer to this page and this page for more information.

If the application still does not report any received packets, try to disable IOMMU. On some systems, it can be done from the BIOS looking for the the `VT-d` or `IOMMU` from the NorthBridge configuration and change that setting to `Disable` and save it. The system may also require adding `intel_iommu=off` or `amd_iommu=off` to the kernel options. That can be done through the grub command line as follows:

```
$ sudo vim /etc/default/grub
# GRUB_CMDLINE_LINUX_DEFAULT="iommu=off intel_iommu=off <more options>"
$ sudo update-grub
$ sudo reboot
```

GPU Configuration

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

GDRCopy

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

```
# 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/bin:${PATH}"
export LD_LIBRARY_PATH="/usr/local/cuda/lib:/usr/local/cuda/lib64:${LD_LIBRARY_PATH}"
export CPATH="$(echo /usr/local/cuda/targets/{x86_64,sbsa}-linux/include | sed 's/ /:/'):${CPATH}"
```

GPU Memory Mapping (nvidia-peermem vs. dmabuf)

To allow the NIC to send and receive packets using GPU memory, it is required to launch the NVIDIA kernel module `nvidia-peermem`. It is shipped by default with the CUDA Toolkit installation.

```
sudo modprobe nvidia-peermem
```

Mapping buffers through the `nvidia-peermem` module is the legacy mapping mode.

Alternatively, DOCA offers the ability to map GPU memory through the `dmabuf` providing a set high-level function. Prerequisites are DOCA installed on a system with:

- Linux Kernel ⩾ 6.2
- libibverbs ⩾ 1.14.44
- CUDA Toolkit 12.5 or older – installed with the `-m=kernel-open` flag (which implies the NVIDIA driver in open-source mode)
- CUDA Toolkit 12.6 or newer – open kernel mode is enabled by default

> **Note**
>
> Installing DOCA on kernel 6.2 to enable the `dmabuf` is experimental.

An example can be found in the DOCA GPU Packet Processing application:

```
/* Get from CUDA the dmabuf file-descriptor for the GPU memory
buffer */
result = doca_gpu_dmabuf_fd(gpu_dev, gpu_buffer_addr,
gpu_buffer_size, &(dmabuf_fd));
if (result != DOCA_SUCCESS) {
        /* If it fails, create a DOCA mmap for the GPU memory
buffer with the nvidia-peermem legacy method */
        doca_mmap_set_memrange(gpu_buffer_mmap, gpu_buffer_addr,
gpu_buffer_size);
} else {
```

```
        /* If it succeeds, create a DOCA mmap for the GPU memory
buffer using the dmabuf method */
        doca_mmap_set_dmabuf_memrange(gpu_buffer_mmap, dmabuf_fd,
gpu_buffer_addr, 0, gpu_buffer_size);
}
```

If the function `doca_gpu_dmabuf_fd` fails, it probably means the NVIDIA driver is not installed with the open-source mode.

Later, when calling the `doca_mmap_start`, the DOCA library tries to map the GPU memory buffer using the `dmabuf` file descriptor. If it fails (something incorrectly set on the Linux system), it fallbacks trying to map the GPU buffer with the legacy mode (`nvidia-peermem`). If it fails, an informative error is returned.

GPU BAR1 Size

Every time a GPU buffer is mapped to the NIC (e.g., buffers associated with send or receive queues), a portion of the GPU BAR1 mapping space is used. Therefore, it is important to check that the BAR1 mapping is large enough to hold all the bytes the DOCA GPUNetIO application is trying to map. To verify the BAR1 mapping space of a GPU you can use `nvidia-smi`:

```
$ nvidia-smi -q

==============NVSMI LOG==============
.....
Attached GPUs                             : 1
GPU 00000000:CA:00.0
    Product Name                          : NVIDIA A100 80GB PCIe
    Product Architecture                  : Ampere
    Persistence Mode                      : Enabled
.....
    BAR1 Memory Usage
        Total                             : 131072 MiB
```

```
        Used                                  : 1 MiB
        Free                                  : 131071 MiB
```

By default, some GPUs (e.g. RTX models) may have a very small BAR1 size:

```
$ nvidia-smi -q | grep -i bar -A 3
        BAR1 Memory Usage
        Total : 256 MiB
        Used : 6 MiB
        Free : 250 MiB
```

If the BAR1 size is not enough, DOCA GPUNetIO applications may exit with errors because DOCA mmap fails to map the GPU memory buffers to the NIC (e.g., `Failed to start mmap DOCA Driver call failure`). To overcome this issue, the GPU BAR1 must be increased from the BIOS. The system should have "Resizable BAR" option enabled. For further information, refer to this NVIDIA forum post.

BlueField-3 Specific Configuration

To run a DOCA GPUNetIO application on the BlueField-3 Arm cores in a converged card (section "Application on DPU Converged Arm CPU"), it is mandatory to set an NVIDIA driver option at the end of the driver configuration file:

```
cat <<EOF | sudo tee /etc/modprobe.d/nvidia.conf
options nvidia NVreg_RegistryDwords="RmDmaAdjustPeerMmioBF3=1;"
EOF
```

To make sure the option has been detected by the NVIDIA driver, run:

```
$ grep RegistryDwords /proc/driver/nvidia/params
RegistryDwords: "RmDmaAdjustPeerMmioBF3=1;"
```

```
RegistryDwordsPerDevice: ""
```

Running without Root Privileges

All DOCA GPUNetIO samples and applications over Ethernet rely on DOCA Flow. For this reason, they must be run with sudo/root privileges.

Alternatively, RDMA and DMA samples can be executed without sudo privileges, if this option is set on the NVIDIA driver.

```
cat <<EOF | sudo tee /etc/modprobe.d/nvidia.conf
options nvidia NVreg_RegistryDwords="PeerMappingOverride=1;"
EOF
```

Perform a cold reboot to the system and then check the option has been applied using the following command:

```
$ grep RegistryDwords /proc/driver/nvidia/params
RegistryDwords: "PeerMappingOverride=1;"
```

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 or DOCA RDMA 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.

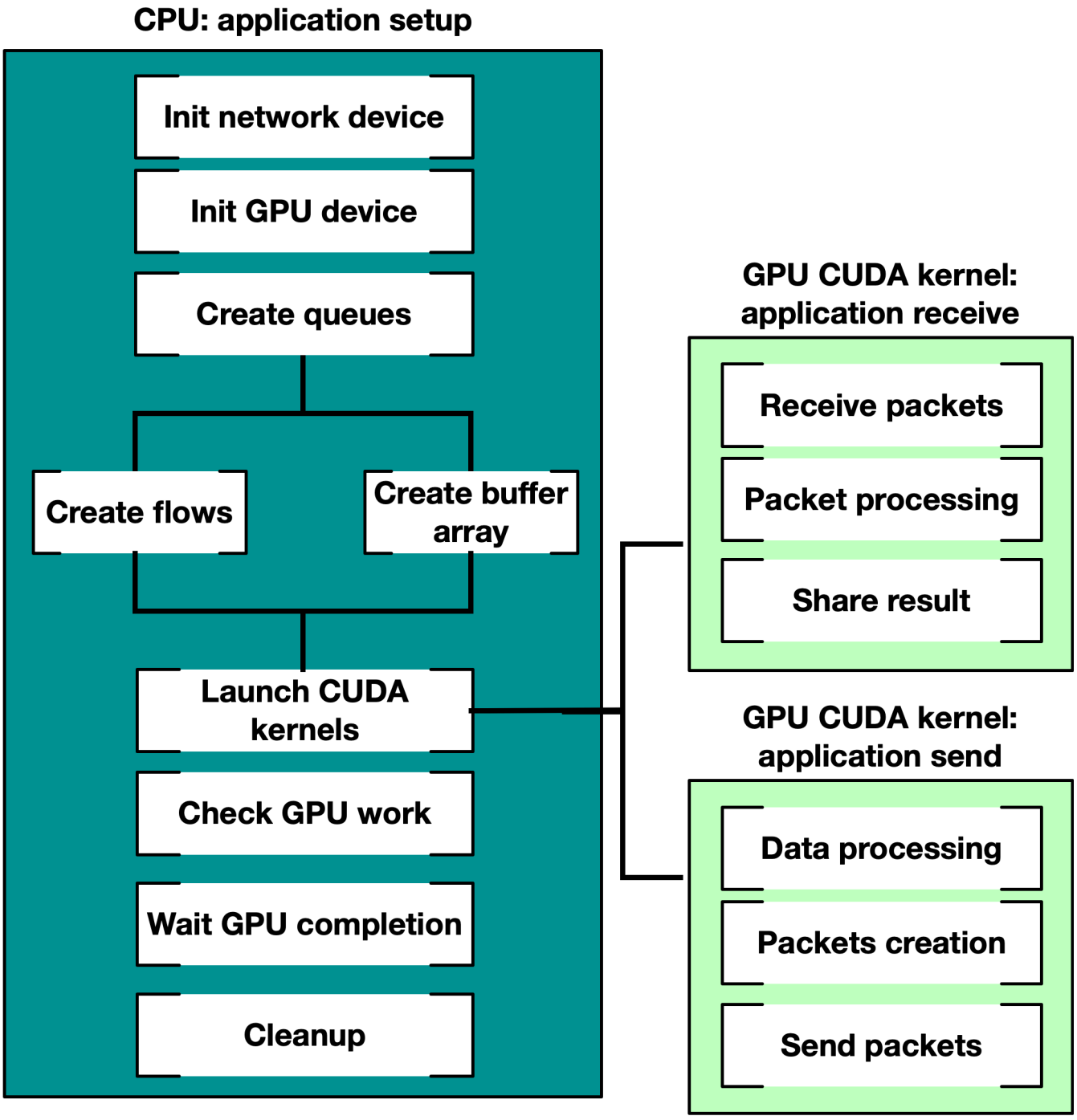
For this reason, DOCA GPUNetIO is composed of two libraries:

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

Note

The pkgconfig file for the DOCA GPUNetIO shared library is `doca-gpunetio.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 the DOCA GPU Packet Processing Application Guide for an example of using DOCA GPUNetIO to send and receive Ethernet packets.

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
- `doca_gpunetio_dev_rdma.cuh` – GPU functions to manage a DOCA RDMA queue
- `doca_gpunetio_dev_dma.cuh` – GPU functions to manage a DOCA DMA queue

This section lists the main functions of DOCA GPUNetIO. To better understand their usage, refer to section "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 DOCA Ethernet.

Tip

To better understand structures, objects, and functions related to RDMA operations, please refer to the DOCA RDMA.

Tip

To better understand structures, objects, and functions related to DMA operations, please refer to the DOCA DMA.

Tip

To better understand DOCA core objects like `doca_mmap` or `doca_buf_array`, please refer to the DOCA Core.

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(&(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;
struct doca_dev *ddev;

/* Create DOCA network device ddev */
/* Create the DOCA Ethernet receive queue */
doca_eth_rxq_create(ddev, MAX_NUM_PACKETS, MAX_PACKET_SIZE,
&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, ...);
```

CPU Functions

In this section there is the list of DOCA GPUNetIO functions that can be used on the CPU only.

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_TYPE_GPU        = 0,
    DOCA_GPU_MEM_TYPE_GPU_CPU    = 1,
    DOCA_GPU_MEM_TYPE_CPU_GPU    = 2,
};
```

Note

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

- `DOCA_GPU_MEM_TYPE_GPU` – memory resides on the GPU and is accessible from the GPU only
- `DOCA_GPU_MEM_TYPE_GPU_CPU` – memory resides on the GPU and is accessible also by the CPU
- `DOCA_GPU_MEM_TYPE_CPU_GPU` – memory resides on the CPU and is accessible also by the GPU

Typical usage of the `DOCA_GPU_MEM_TYPE_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).

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`.

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! I f an application tries to access the memory using the `memptr_gpu` address from the CPU, a segmentation fault will result.

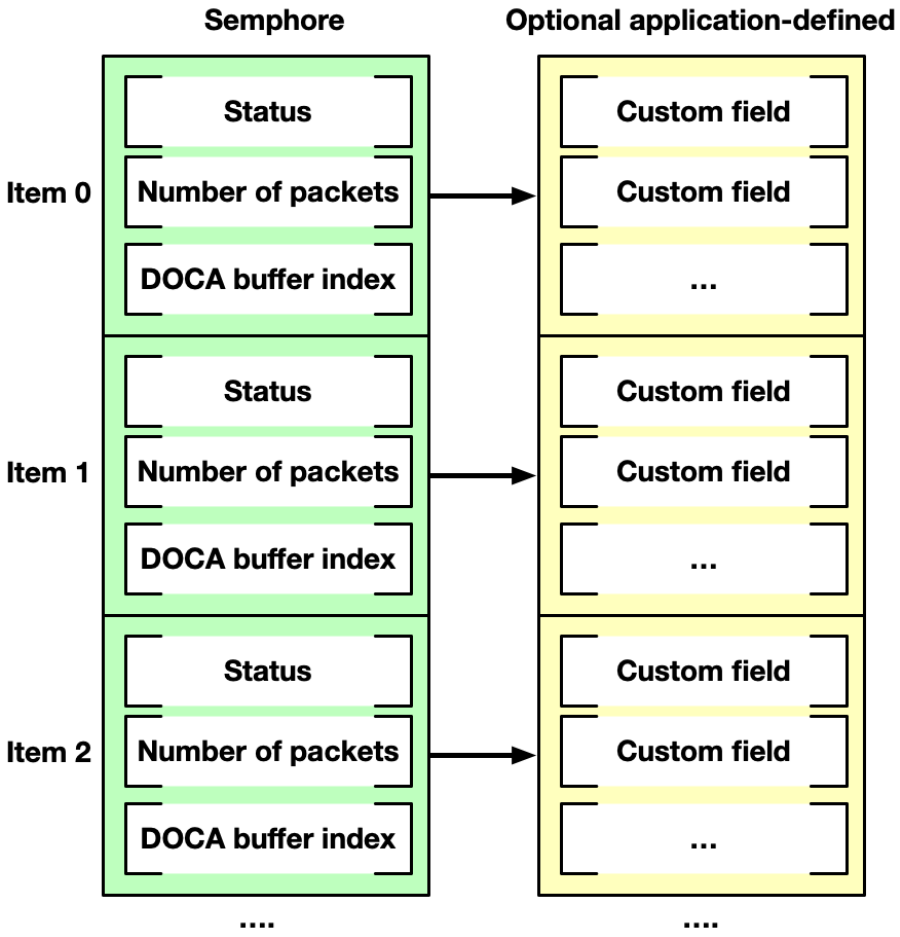
doca_gpu_semaphore_create

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 section "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 in the "Receive and Process" section.



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

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 statistics or output of the pipeline computation), then `DOCA_GPU_MEM_CPU_GPU` is the suggested memory type

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

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

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

doca_gpu_semaphore_get_custom_info_addr

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

DOCA PE

A DOCA Ethernet Txq context, exported for GPUNetIO usage, can be tracked via DOCA PE on the CPU side to check if there are errors when sending packets or to retrieve notification info after sending a packet with any of the `doca_gpu_dev_eth_txq_*_enqueue_*` functions on the GPU. An example can be found in the DOCA GPU packet processing application with ICMP traffic.

Strong Mode vs. Weak Mode

Some Ethernet and RDMA GPU functions present two modes of operation: Weak and strong.

- In weak mode, the application calculates the next available position in the queue. With the help of functions like `doca_gpu_eth_txq_get_info`, `doca_gpu_rdma_get_info`, or `doca_gpu_dev_rdma_recv_get_info` it is possible to know the next available position in the queue and the mask of the number of total entries in the queue (so the incremental descriptor index can be wrapped). In this 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. It's a bit more complex to manage but it should result in better performance and developer can emphasize GPU memory coalescing enqueuing sequential operations using sequential memory locations.

- In strong mode, the GPU function enqueues the Ethernet/RDMA operation in the next available position in the queue. It is simpler to manage as developer does not have to worry about operation's position, but it may introduce an extra latency to atomically guarantee the access of multiple threads to the same queue. Moreover, it does not guarantee that sequential operations refer to sequential memory locations.

> **Note**
>
> All strong mode functions work at the CUDA block level. That is, it is not possible to access the same Eth/RDMA queue at the

same time from two different CUDA blocks.

In sections "Produce and Send" and "CUDA Kernel for RDMA Write", there are a few examples about how to use the weak mode API.

GPU Functions – Ethernet

This section provides a list of DOCA GPUNetIO functions that can be used for Ethernet network operations on the GPU only within a CUDA kernel.

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 to receive. It ensures the number of packets returned by the function is lower or equal to this number.
- `timeout_ns` – Nanoseconds to wait for packets before returning
- `num_rx_pkts [out]` – Effective number of received packets. With CUDA block or warp scopes, this variable should be visible in memory by all the other threads

(shared or global memory).

- `doca_gpu_buf_idx [out]` – DOCA buffer index of the first packet received in this function. With CUDA block or warp scopes, this variable should be visible in memory by all the other threads (shared or global memory).

> **Note**
>
> If both `max_rx_pkts` and `timeout_ns` are 0, the function never returns.

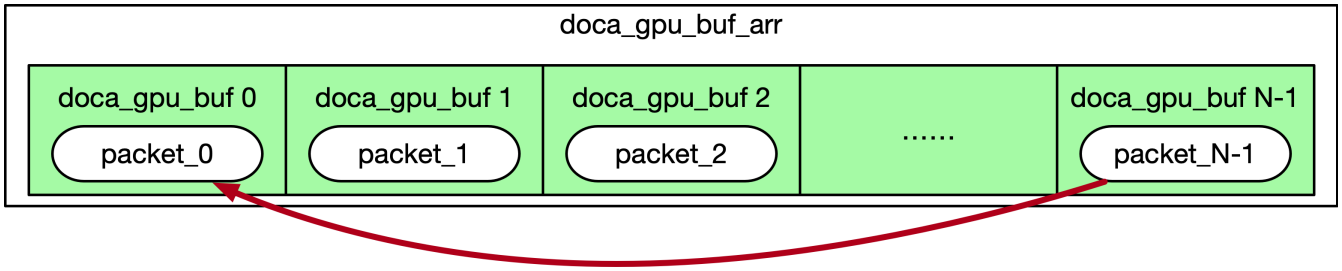
CUDA threads in the same scope (thread, warp, or block) must invoke the function on the same receive queue. The output parameters `num_rx_pkts` and `doca_gpu_buf_idx` must be visible by all threads in the scope (e.g., CUDA shared memory for warp and block).

Each packet received by this function goes to the `doca_gpu_buf_arr` internally created and associated with the Ethernet queues (see section "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 received (`num_rx_pkts`) and the index of the first received packet in the `doca_gpu_buf_arr` 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.

doca_gpu_send_flags

This enum lists all the possible flags for the txq functions. The usage of those flags makes sense if a DOCA PE has been attached to the DOCA Ethernet Txq context with GPU data path and a CPU thread, in a loop, keeps invoking `doca_pe_progress`.

> **Warning**
>
> If no DOCA PE has been attached to the DOCA Ethernet Txq context, it's mandatory to use the `DOCA_GPU_SEND_FLAG_NONE` flag.

```
enum doca_gpu_mem_type {
    DOCA_GPU_SEND_FLAG_NONE       = 0,
```

```
    DOCA_GPU_SEND_FLAG_NOTIFY     = 1 << 0,
};
```

- `DOCA_GPU_SEND_FLAG_NONE` (default) – send is executed and no notification info is returned. If an error occurs, an event is generated. This error can be detected from the CPU side using DOCA PE.
- `DOCA_GPU_SEND_FLAG_NOTIFY` – once the send (or wait) is executed, return a notification with packet info. This notification can be detected from the CPU side using DOCA PE.

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` in the send queue.

```
__device__ doca_error_t doca_gpu_dev_eth_txq_get_info(struct
doca_gpu_eth_txq *eth_txq, uint32_t *curr_position, uint32_t
*mask_max_position)
```

- `eth_txq` – Ethernet send queue GPU handler
- `curr_position` – Next available position in the queue
- `mask_max_position` – Mask of the total number of positions in the queue

```
__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,
const uint32_t flags_bitmask)
```

- `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
- `flags_bitmask` – One of the flags in the `doca_gpu_send_flags` enum

```
__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, const uint32_t flags_bitmask)
```

- `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 - `mask_max_position`.
- `flags_bitmask` – One of the flags in the `doca_gpu_send_flags` enum

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. Like `doca_gpu_dev_eth_txq_send_*`, `doca_gpu_dev_eth_txq_wait_*` also has a strong and weak mode.

```
__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,
const uint32_t flags_bitmask)
```

- `eth_txq` – Ethernet send queue GPU handler
- `wait_on_time_value` – Timestamp to specify when packets must be sent after this barrier
- `flags_bitmask` – One of the flags in the `doca_gpu_send_flags` enum

```
__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,
const uint32_t flags_bitmask)
```

- `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 - `mask_max_position`.
- `flags_bitmask` – One of the flags in the `doca_gpu_send_flags` enum

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

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_*`.

```
__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

```
__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 several 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.

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 several 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).

GPU Functions – RDMA

This section provides a list of DOCA GPUNetIO functions that can be used on the GPU only within a CUDA kernel to execute RDMA operations. These functions offer a strong and a weak mode.

```
__device__ doca_error_t __device__ doca_error_t
doca_gpu_dev_rdma_get_info(struct doca_gpu_dev_rdma *rdma, uint32_t
connection_index, uint32_t *curr_position, uint32_t
*mask_max_position)
```

- `rdma` – RDMA queue GPU handler
- `connection_index` – In case of RDMA CM, the connection index must be specified. By default, it is 0.
- `curr_position` – Next available position in the queue
- `mask_max_position` – Mask of the total number of positions in the queue

```
__device__ doca_error_t __device__ doca_error_t
doca_gpu_dev_rdma_recv_get_info(struct doca_gpu_dev_rdma_r *rdma_r,
uint32_t *curr_position, uint32_t *mask_max_position)
```

- `rdma_r` – RDMA receive queue GPU handler
- `curr_position` – Next available position in the queue
- `mask_max_position` – Mask of the total number of positions in the queue

doca_gpu_dev_rdma_write_*

To RDMA write data onto a remote memory location from a CUDA kernel, DOCA GPUNetIO offers strong and weak modes for enqueuing operations on the RDMA queue. For both modes, the scope is the single CUDA thread.

```
__device__ doca_error_t doca_gpu_dev_rdma_write_strong(struct
doca_gpu_dev_rdma *rdma,

uint32_t connection_index,

struct doca_gpu_buf *remote_buf, uint64_t remote_offset,

struct doca_gpu_buf *local_buf, uint64_t local_offset,

size_t length, uint32_t imm,

const enum doca_gpu_dev_rdma_write_flags flags)
```

- `rdma` – RDMA queue GPU handler
- `connection_index` – In case of RDMA CM, the connection index must be specified. By default, it is 0.

- `remote_buf` – Remote DOCA buffer from a DOCA GPU buffer array to write data to
- `remote_offset` – Offset, in bytes, to write data to in the remote buffer
- `local_buf` – Local DOCA buffer from a DOCA GPU buffer array from which to fetch data to write
- `local_offset` – Offset, in bytes, to fetch data from in the local buffer
- `length` – Number of bytes to write
- `imm` – Immediate value `uint32_t`
- `flags` – One of the flags in the `doca_gpu_dev_rdma_write_flags` enum

```
__device__ doca_error_t doca_gpu_dev_rdma_write_weak(struct
doca_gpu_dev_rdma *rdma,

uint32_t connection_index,

struct doca_gpu_buf *remote_buf, uint64_t remote_offset,

struct doca_gpu_buf *local_buf, uint64_t local_offset,

size_t length, uint32_t imm,

const enum doca_gpu_dev_rdma_write_flags flags,

uint32_t position);
```

- `rdma` – RDMA queue GPU handler
- `connection_index` – In case of RDMA CM, the connection index must be specified. By default, it is 0.
- `remote_buf` – Remote DOCA buffer from a DOCA GPU buffer array to write data to

- `remote_offset` – Offset, in bytes, to write data to in the remote buffer
- `local_buf` – Local DOCA buffer from a DOCA GPU buffer array where to fetch data to write
- `local_offset` – Offset, in bytes, to fetch data in the local buffer
- `length` – Number of bytes to write
- `imm` – Immediate value `uint32_t`
- `flags` – One of the flags in the `doca_gpu_dev_rdma_write_flags` enum
- `position` – Position in the queue to place the RDMA operation. Range: 0 - `mask_max_position`.

doca_gpu_dev_rdma_read_*

To RDMA read data onto a remote memory location from a CUDA kernel, DOCA GPUNetIO offers strong and weak modes to enqueue operations on the RDMA queue. For both modes, the scope is the single CUDA thread.

```
__device__ doca_error_t doca_gpu_dev_rdma_read_strong(struct
doca_gpu_dev_rdma *rdma,

uint32_t connection_index,

struct doca_gpu_buf *remote_buf, uint64_t remote_offset,

struct doca_gpu_buf *local_buf, uint64_t local_offset,

size_t length,

const uint32_t flags_bitmask)
```

- `rdma` – RDMA queue GPU handler
- `connection_index` – In case of RDMA CM, the connection index must be specified. By default, it is 0.
- `remote_buf` – Remote DOCA buffer from a DOCA GPU buffer array where to read data
- `remote_offset` – Offset in bytes to read data to in the remote buffer
- `local_buf` – Local DOCA buffer from a DOCA GPU buffer array where to store remote data
- `local_offset` – Offset in bytes to store data in the local buffer
- `length` – Number of bytes to be read
- `flags_bitmask` – Must be 0; reserved for future use

```
__device__ doca_error_t doca_gpu_dev_rdma_read_weak(struct
doca_gpu_dev_rdma *rdma,

uint32_t connection_index,

struct doca_gpu_buf *remote_buf, uint64_t remote_offset,

struct doca_gpu_buf *local_buf, uint64_t local_offset,

size_t length,

const uint32_t flags_bitmask,

uint32_t position);
```

- `rdma` – RDMA queue GPU handler

- `connection_index` – In case of RDMA CM, the connection index must be specified. By default, it is 0.
- `remote_buf` – Remote DOCA buffer from a DOCA GPU buffer array where to read data
- `remote_offset` – Offset in bytes to read data to in the remote buffer
- `local_buf` – Local DOCA buffer from a DOCA GPU buffer array where to store remote data
- `local_offset` – Offset in bytes to store data in the local buffer
- `length` – Number of bytes to be read
- `flags_bitmask` – Must be 0; reserved for future use
- `position` – Position in the queue to place the RDMA operation. Range: 0 - `mask_max_position`.

doca_gpu_dev_rdma_send_*

To RDMA send data from a CUDA kernel, DOCA GPUNetIO offers strong and weak modes for enqueuing operations on the RDMA queue. For both modes, the scope is the single CUDA thread.

```
__device__ doca_error_t doca_gpu_dev_rdma_send_strong(struct
doca_gpu_dev_rdma *rdma,

uint32_t connection_index,

struct doca_gpu_buf *local_buf, uint64_t local_offset,

size_t length, uint32_t imm,
```

```
const enum doca_gpu_dev_rdma_write_flags flags)
```

- `rdma` – RDMA queue GPU handler
- `connection_index` – In case of RDMA CM, the connection index must be specified. By default, it is 0.
- `local_buf` – Local DOCA buffer from a DOCA GPU buffer array from which to fetch data to send
- `local_offset` – Offset in bytes to fetch data in the local buffer
- `length` – Number of bytes to send
- `imm` – Immediate value `uint32_t`
- `flags` – One of the flags in the `doca_gpu_dev_rdma_write_flags` enum

```
__device__ doca_error_t doca_gpu_dev_rdma_send_weak(struct
doca_gpu_dev_rdma *rdma,

uint32_t connection_index,

struct doca_gpu_buf *local_buf, uint64_t local_offset,

size_t length, uint32_t imm,

const enum doca_gpu_dev_rdma_write_flags flags,

uint32_t position);
```

- `rdma` – RDMA queue GPU handler

- `connection_index` – In case of RDMA CM, the connection index must be specified. By default, it is 0.
- `local_buf` – Local DOCA buffer from a DOCA GPU buffer array from which to fetch data to send
- `local_offset` – Offset in bytes to fetch data in the local buffer
- `length` – Number of bytes to send
- `imm` – Immediate value `uint32_t`
- `flags` – One of the flags in the `doca_gpu_dev_rdma_write_flags` enum
- `position` – Position in the queue to place the RDMA operation. Range: 0 - `mask_max_position`.

doca_gpu_dev_rdma_commit_*

Once all RDMA write, send or read requests have been enqueue in the RDMA queue, a synchronization point must be reached to consolidate and execute those requests. Only 1 CUDA thread can invoke this function at a time.

```
__device__ doca_error_t doca_gpu_dev_rdma_commit_strong(struct
doca_gpu_dev_rdma *rdma, uint32_t connection_index)
```

- `rdma` – RDMA queue GPU handler
- `connection_index` – In case of RDMA CM, the connection index must be specified. By default, it is 0.

```
__device__ doca_error_t doca_gpu_dev_rdma_commit_weak(struct
doca_gpu_dev_rdma *rdma, uint32_t connection_index, uint32_t
num_ops)
```

- `rdma` – RDMA queue GPU handler
- `connection_index` – In case of RDMA CM, the connection index must be specified. By default, it is 0.
- `num_ops` – Number of RDMA requests enqueued since the last commit

doca_gpu_dev_rdma_wait_all

After a commit, RDMA requests are executed by the network card as applications move forward doing other operations. If the application needs to verify all RDMA operations have been done by the network card, this "wait all" function can be used to wait for all previous posted operations. Only 1 CUDA thread can invoke this function at a time.

```
__device__ doca_error_t doca_gpu_dev_rdma_wait_all(struct
doca_gpu_dev_rdma *rdma, uint32_t *num_commits)
```

- `rdma` – RDMA queue GPU handler
- `num_commits` – Output parameter; the number of commit operations completed

Info

This function is optional, and it can be used to ensure all the RDMA Send/Write/Read operations have actually been executed before moving forward with the application.

doca_gpu_dev_rdma_recv_*

To receive data from an RDMA send, send with immediate, or write with immediate, the destination peer should post a receive operation. DOCA GPUNetIO RDMA receive

operations must be done with a `doca_gpu_dev_rdma_r` handler. This handler can be obtained with the function `doca_gpu_dev_rdma_get_recv`.

> **Note**
>
> All receive operations must use this object.

```
__device__ doca_error_t doca_gpu_dev_rdma_get_recv(struct
doca_gpu_dev_rdma *rdma, struct doca_gpu_dev_rdma_r **rdma_r)
```

- `rdma` – RDMA queue GPU handler
- `rdma_r` – RDMA receive queue GPU handler

Even for the receive side, in this case, DOCA GPUNetIO offers strong and weak modes for enqueuing operations on the RDMA queue. For both modes, the scope is the single CUDA thread.

```
__device__ doca_error_t doca_gpu_dev_rdma_recv_strong(struct
doca_gpu_dev_rdma_r *rdma_r,

struct doca_gpu_buf *recv_buf,

size_t recv_length,

uint64_t recv_offset,

const uint32_t flags_bitmask)
```

- `rdma_r` – RDMA receive queue GPU handler

- `recv_buf` – Local DOCA buffer from a DOCA GPU buffer array from which to fetch data to send
- `recv_length` – Number of bytes to send
- `recv_offset` – Offset in bytes to fetch data in the local buffer
- `flags_bitmask` – Must be 0; reserved for future use

```
__device__ doca_error_t doca_gpu_dev_rdma_recv_weak(struct
doca_gpu_dev_rdma_r *rdma_r,

struct doca_gpu_buf *recv_buf,

size_t recv_length,

uint64_t recv_offset,

const uint32_t flags_bitmask,

uint32_t position);
```

- `rdma_r` – RDMA receive queue GPU handler
- `recv_buf` – Local DOCA buffer from a DOCA GPU buffer array from which to fetch data to send
- `recv_length` – Number of bytes to send
- `recv_offset` – Offset in bytes to fetch data in the local buffer
- `flags_bitmask` - Must be 0; reserved for future use
- `position` – Position in the queue to place the RDMA operation. Range: 0 - `mask_max_position`.

doca_gpu_dev_rdma_recv_commit_*

After posting several RDMA receive operations, a commit function must be invoked to activate the receive in the queue. Only 1 CUDA thread can invoke this function at a time.

```
__device__ doca_error_t doca_gpu_dev_rdma_recv_commit_strong(struct doca_gpu_dev_rdma_r *rdma_r)
```

- `rdma_r` – RDMA receive queue GPU handler

```
__device__ doca_error_t doca_gpu_dev_rdma_recv_commit_weak(struct doca_gpu_dev_rdma_r *rdma_r, uint32_t num_ops)
```

- `rdma_r` – RDMA receive queue GPU handler
- `num_ops` – Number of RDMA receive requests enqueued since the last commit

doca_gpu_dev_rdma_recv_wait_all

This function waits for the completion of all previously posted RDMA receive operation. Only 1 CUDA thread can invoke this function at a time. It works in blocking or non-blocking mode.

```
enum doca_gpu_dev_rdma_recv_wait_flags {
        DOCA_GPU_RDMA_RECV_WAIT_FLAG_NB = 0,    /**< Non-Blocking mode: the wait receive function doca_gpu_dev_rdma_recv_wait
                                                 * checks if the receive operation happened (data has been received)
                                                 * and exit from the function. If nothing has been received,
                                                 * the function doesn't block the execution.
                                                 */
```

```
        DOCA_GPU_RDMA_RECV_WAIT_FLAG_B  = 1, /**< Blocking mode: the wait receive function doca_gpu_dev_rdma_recv_wait
                                            * blocks the execution waiting for the receive operations to be executed.
                                            */
};
```

Function:

```
__device__ doca_error_t doca_gpu_dev_rdma_recv_wait_all(struct doca_gpu_dev_rdma_r *rdma_r, const enum doca_gpu_dev_rdma_recv_wait_flags flags, uint32_t *num_ops, uint32_t *imm_val)
```

- `rdma_r` – RDMA receive queue GPU handler
- `flags` – receive flags
- `num_ops` – Output parameter. Function reports number of completed operations.
- `imm_val` – Output parameter. Application-provided buffer where the function can store received immediate values, if any (or 0xFFFFFFFF if no immediate value is received). If `nullptr`, the function ignores this parameter.

GPU Functions – DMA

This section provides a list of DOCA GPUNetIO functions that can be used on the GPU only within a CUDA kernel to execute DMA operations.

doca_gpu_dev_dma_memcpy

This function allows a CUDA kernel to trigger a DMA memory copy operation through the DMA GPU engine. There is no strong/weak mode here, the DMA is assuming the strong behavior by default.

```
__device__ doca_error_t doca_gpu_dev_dma_memcpy(struct doca_gpu_dma
*dma, struct doca_gpu_buf *src_buf, uint64_t src_offset, struct
doca_gpu_buf *dst_buf, uint64_t dst_offset, size_t length);
```

- `dma` – DMA queue GPU handler
- `src_buf` – memcpy source buffer
- `src_offset` – fetch data starting from this source buffer offset
- `dst_buf` – memcpy destination buffer
- `dst_offset` – copy data starting from this destination buffer offset
- `lenght` – number of bytes to copy

doca_gpu_dev_dma_commit

After posting several DMA memory copies, a commit function must be invoked to execute the operations enqueued in the DMA queue. Only 1 CUDA thread can invoke this function at a time.

```
__device__ doca_error_t doca_gpu_dev_dma_commit(struct doca_gpu_dma
*dma);
```

- `dma` – DMA queue GPU handler

Building Blocks

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

Initialize GPU

A GPU object must be created and associated to the GPU device at a specific PCIe address:

```
struct doca_gpu *gdev;

/* Create GPUNetIO handler on a specific GPU */
doca_gpu_create(gpu_pcie_address, &gdev);
```

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);
```

Ethernet Queue with GPU Data Path

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.

```
/* 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_dev *ddev;
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(ddev, MAX_PACKETS_NUM, MAX_PACKETS_SIZE,
&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);
/* Set DOCA mmap properties */
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);
/* This DOCA Ethernet Rxq object will be managed by the GPU */
doca_ctx_set_datapath_on_gpu();
/* 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.

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_dev *ddev;
struct doca_eth_txq *eth_txq_cpu;
struct doca_gpu_eth_txq *eth_txq_gpu;

/* Create DOCA Ethernet send queues */
doca_eth_txq_create(ddev, QUEUE_DEPTH, &eth_txq_cpu);
/* Set properties to send queues */

/* This DOCA Ethernet Rxq object will be managed by the GPU */
doca_ctx_set_datapath_on_gpu();
/* Start the Ethernet queue object */
/* Export GPU handle for the send queue */
doca_eth_txq_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);
```

Receive and Process

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.

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;
        }
}
```

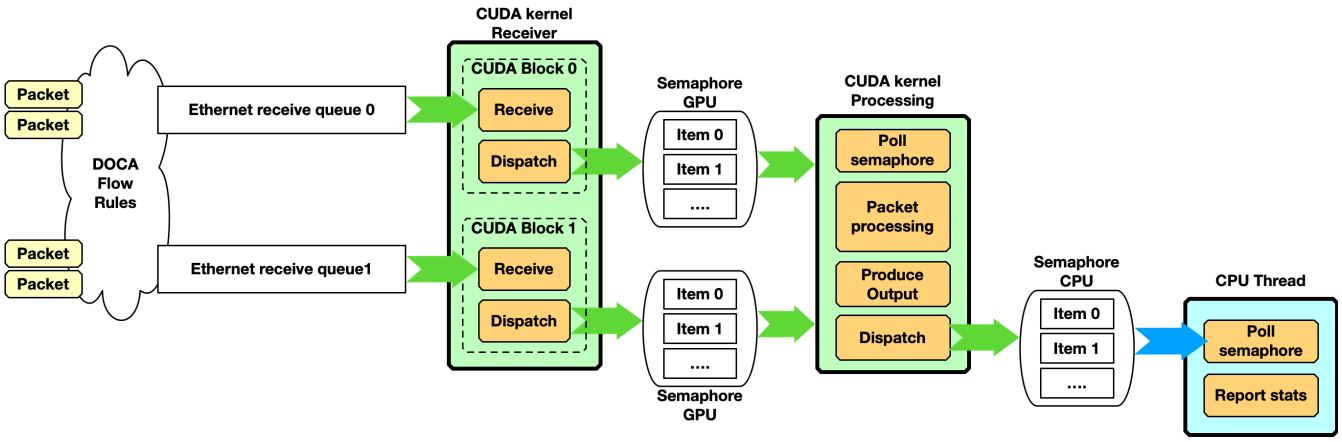
On the GPU, the two CUDA kernels are running on different streams:

```
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;

        while (/* exit condition */) {
                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;
```

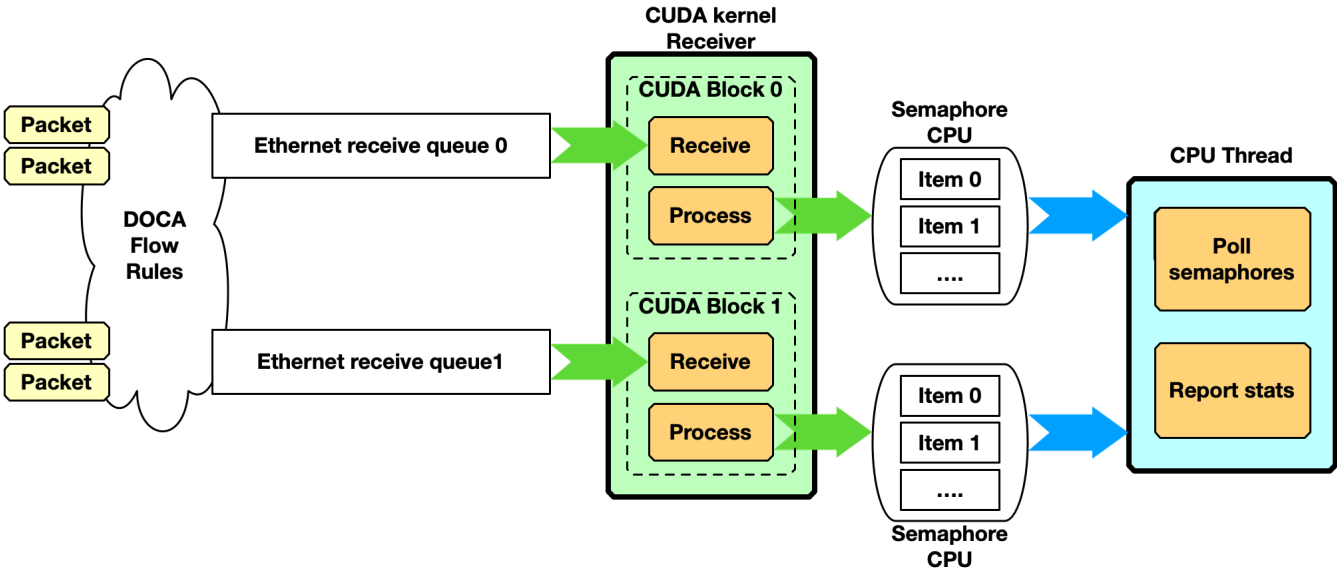
```
        struct custom_info *gpu_info;

        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);
                }
                __syncthreads();

                if (threadIdx.x == 0) {

doca_gpu_dev_sem_set_status(sem_to_gpu_gpu, semaphore_index,
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.

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;
        uint32_t curr_position;
        uint32_t mask_max_position;
        uint32_t num_pkts_per_send = blockDim.x;

        /* Get last occupied position in the Tx queue */
        doca_gpu_dev_eth_txq_get_info(eth_txq_gpu,
&curr_position, &mask_max_position);
        __syncthreads();

        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 with weak mode: each thread posts the
packet in a different and sequential position of the queue */

doca_gpu_dev_eth_txq_send_enqueue_weak(eth_txq_gpu, buf,
packet_len, ((curr_position + doca_gpu_buf_idx) &
mask_max_position), DOCA_GPU_SEND_FLAG_NONE);

                /* 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_weak(eth_txq_gpu, num_pkts_per_send);
                        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 += blockDim.x;
        }
}
```

RDMA Queue with GPU Data Path

To execute RDMA operations from a GPU CUDA kernel, in the setup phase, the application must first create a DOCA RDMA queue, export the RDMA as context, and then set the datapath of the context on the GPU (as shown in the following code snippet).

The following is a pseudo-code to serve as a guide. Please refer to real function signatures in header files (`*.h`) and documentation for a complete overview of the functions.

```
struct doca_dev *doca_device;                /* DOCA device */
struct doca_gpu *gpudev;          /* DOCA GPU device */
struct doca_rdma *rdma;                      /* DOCA RDMA instance */
struct doca_gpu_dev_rdma *gpu_rdma; /* DOCA RDMA instance GPU handler */
struct doca_ctx *rdma_ctx;

// Initialize IBDev RDMA device
open_doca_device_with_ibdev_name(&doca_device)

// Initialize the GPU device
```

```
doca_gpu_create(&gpudev);
// Create the RDMA queue object with the DOCA device
doca_rdma_create(doca_device, &(rdma));
// Export the RDMA queue object context
rdma_ctx = doca_rdma_as_ctx(rdma)

// Set RDMA queue attributes

// Set GPU data path for the RDMA object
doca_ctx_set_datapath_on_gpu(ctx, gpudev)
doca_ctx_start(rdma_ctx);
```

At this point, the application has an RDMA queue usable from a GPU CUDA kernel. The next step would be to establish a connection using some OOB (out-of-band) mechanism (e.g., Linux sockets) to exchange RDMA queue info so each peer can connect to the other's queues.

To exchange data, users must create DOCA GPU buffer arrays to send or receive data. If the application also requires read or write, then the GPU memory associated with the buffer arrays must be exported and exchanged with the remote peers using the OOB mechanism.

```
/* 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 */
doca_mmap_start(mmap);
/* Export mmap info to share with remote peer */
doca_mmap_export_rdma(mmap, ...);

/* Exchange export info with remote peer */

/* 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);
```

Please refer to the "RDMA Client Server" sample as a basic layout to implement all the steps described in this section.

CUDA Kernel for RDMA Write

Assuming the RDMA queues and buffer arrays are correctly created and exchanged across peers, the application can launch a CUDA kernel to remotely write data. As typically applications use `strong` mode, the following code snippet shows how to use `weak` mode to post multiple writes from different CUDA threads in the same CUDA block.

```
__global__ void rdma_write_bw(struct doca_gpu_dev_rdma *rdma_gpu,
struct doca_gpu_buf_arr *local_buf_arr, struct doca_gpu_buf_arr
*remote_buf_arr)
{
        struct doca_gpu_buf *remote_buf;
        struct doca_gpu_buf *local_buf;
        struct doca_gpu_dev_rdma *rdma_gpu;
        struct doca_gpu_buf_arr *server_local_buf_arr;
        struct doca_gpu_buf_arr *server_remote_buf_arr;
        uint32_t curr_position;
        uint32_t mask_max_position;
        uint32_t num_ops;

        doca_gpu_dev_buf_get_buf(server_local_buf_arr,
threadIdx.x, &local_buf);
        doca_gpu_dev_buf_get_buf(server_remote_buf_arr,
threadIdx.x, &remote_buf);
        /* Get RDMA queue current available position and mask of the max position number */
        doca_gpu_dev_rdma_get_info(rdma_gpu, &curr_position,
&mask_max_position);

        doca_gpu_dev_rdma_write_weak(rdma_gpu,
```

```
                                                        /* Write into this remote buffer at offset
0 */
                                                        remote_buf, 0,
                                                        /* Fetch data from this local buffer at
offset 0 */
                                                        local_buf, 0,
                                                        /* Number of bytes to write */
                                                        msg_size,
                                                        /* Don't use immediate */
                                                        0,
DOCA_GPU_RDMA_WRITE_FLAG_NONE,
                                                        /* Position in the RDMA queue to post
the write */
                                                        (curr_position +
threadIdx.x) & mask_max_position);

                /* Wait all CUDA threads to post their RDMA Write */
                __syncthreads();

                if (threadIdx.x == 0) {
                        /* Only 1 CUDA thread can push the write op just posted */
                        doca_gpu_dev_rdma_commit_weak(rdma_gpu,
blockDim.x);
                        doca_gpu_dev_rdma_wait_all(rdma_gpu,
&num_ops);
                }
                __syncthreads();
}
```

Info

The code in the "RDMA Client Server" sample shows how to use write and send with immediate flag set.

GPUNetIO Samples

This section contains two samples that show how to enable simple GPUNetIO features. Be sure to correctly set the following environment variables:

```
export PATH=${PATH}:/usr/local/cuda/bin
export CPATH="$(echo /usr/local/cuda/targets/{x86_64,sbsa}-linux/include | sed 's/ /:/'):${CPATH}"
export
PKG_CONFIG_PATH=${PKG_CONFIG_PATH}:/usr/lib/pkgconfig:/opt/mellanox
linux-gnu/pkgconfig:/opt/mellanox/dpdk/lib/{x86_64,aarch64}-
linux-gnu/pkgconfig:/opt/mellanox/doca/lib/{x86_64,aarch64}-
linux-gnu/pkgconfigexport
LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/usr/local/cuda/lib64:/opt/mella
linux-gnu:/opt/mellanox/doca/lib/{x86_64,aarch64}-linux-gnu
```

Info

All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

Note

Please ensure the arch of your GPU is included in the `meson.build` file before building the samples (e.g., `sm_80` for Ampere, `sm_89` for L40, `sm_90` for H100, etc).

Ethernet Send Wait Time

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 application-provided timestamps.

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

> **Info**
>
> This NVIDIA blog post offers an example for how this feature has been used in 5G networks.

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

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 15 /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 the status of `phc2sys`:

```
$ 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 15 /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.

> ⚠ **Warning**
>
> 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.

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
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`).

The following example programs a system with GPU PCIe address `ca:00.0` and NIC PCIe address `17:00.0` to send 32 packets every 5 milliseconds:

```
# Ensure DOCA and DPDK are in the LD_LIBRARY_PATH environment variable
$ sudo ./build/doca_gpunetio_send_wait_time -n 17:00.0 -g ca:00.0
-t 5000000[09:22:54:165778][1316878][DOCA][INF]
[gpunetio_send_wait_time_main.c:195][main] Starting the sample
[09:22:54:438260][1316878][DOCA][INF]
[gpunetio_send_wait_time_main.c:224][main] Sample configuration:
                GPU ca:00.0
                NIC 17:00.0
                Timeout 5000000ns
EAL: Detected CPU lcores: 128
...
EAL: Probe PCI driver: mlx5_pci (15b3:a2d6) device: 0000:17:00.0
(socket 0)
[09:22:54:819996][1316878][DOCA][INF]
[gpunetio_send_wait_time_sample.c:607][gpunetio_send_wait_time]
Wait on time supported mode: DPDK
EAL: Probe PCI driver: gpu_cuda (10de:20b5) device: 0000:ca:00.0
(socket 1)
[09:22:54:830212][1316878][DOCA][INF]
[gpunetio_send_wait_time_sample.c:252][create_tx_buf] Mapping
send queue buffer (0x0x7f48e32a0000 size 262144B) with legacy
nvidia-peermem mode
[09:22:54:832462][1316878][DOCA][INF]
[gpunetio_send_wait_time_sample.c:657][gpunetio_send_wait_time]
Launching CUDA kernel to send packets
[09:22:54:842945][1316878][DOCA][INF]
[gpunetio_send_wait_time_sample.c:664][gpunetio_send_wait_time]
Waiting 10 sec for 256 packets to be sent
[09:23:04:883309][1316878][DOCA][INF]
[gpunetio_send_wait_time_sample.c:684][gpunetio_send_wait_time]
Sample finished successfully
```

```
[09:23:04:883339][1316878][DOCA][INF]
[gpunetio_send_wait_time_main.c:239][main] Sample finished
successfully
```

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.

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`).

Ethernet Simple Receive

This simple application shows the fundamental steps to build a DOCA GPUNetIO receiver application with one queue for UDP packets and one CUDA kernel receiving those packets from the GPU, printing packet info to the console.

> ⚠ **Warning**
>
> Invoking a `printf` from a CUDA kernel is not good practice for release software and should be used only to print debug information as it slows down the overall execution of the CUDA kernel.

To build and run the application:

```
# Ensure DOCA and DPDK are in the pkgconfig environment variable
cd /opt/mellanox/doca/samples/doca_gpunetio/gpunetio_simple_receive
meson build
ninja -C build
```

To test the application, this guide assumes the usual setup with two machines: one with the DOCA receiver application and the second one acting as packet generator. As UDP packet generator, this example considers the `nping` application that can be easily installed easily on any Linux machine.

The command to send 10 UDP packets via `nping` on the packet generator machine is:

```
$ nping --udp -c 10 -p 2090 192.168.1.1 --data-length 1024 --
delay 500ms

Starting Nping 0.7.80 ( https://nmap.org/nping ) at 2023-11-20
11:05 UTC
SENT (0.0018s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (0.5018s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (1.0025s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (1.5025s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (2.0032s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (2.5033s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (3.0040s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (3.5040s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (4.0047s) UDP packet with 1024 bytes to 192.168.1.1:2090
SENT (4.5048s) UDP packet with 1024 bytes to 192.168.1.1:2090

Max rtt: N/A | Min rtt: N/A | Avg rtt: N/A
UDP packets sent: 10 | Rcvd: 0 | Lost: 10 (100.00%)
Nping done: 1 IP address pinged in 5.50 seconds
```

Assuming the DOCA Simple Receive sample is waiting on the other machine at IP address `192.168.1.1`.

The DOCA Simple Receive sample is launched on a system with NIC at `17:00.1` PCIe address and GPU at `ca:00.0` PCIe address:

```
# Ensure DOCA and DPDK are in the LD_LIBRARY_PATH environment variable
$ sudo ./build/doca_gpunetio_simple_receive -n 17:00.1 -g ca:00.0
[11:00:30:397080][2328673][DOCA][INF]
[gpunetio_simple_receive_main.c:159][main] Starting the sample
[11:00:30:652622][2328673][DOCA][INF]
[gpunetio_simple_receive_main.c:189][main] Sample configuration:
        GPU ca:00.0
```

```
        NIC 17:00.1

EAL: Detected CPU lcores: 128
EAL: Detected NUMA nodes: 2
EAL: Detected shared linkage of DPDK
EAL: Multi-process socket /var/run/dpdk/rte/mp_socket
EAL: Selected IOVA mode 'PA'
EAL: VFIO support initialized
TELEMETRY: No legacy callbacks, legacy socket not created
EAL: Probe PCI driver: mlx5_pci (15b3:a2d6) device: 0000:17:00.1
(socket 0)
[11:00:31:036760][2328673][DOCA][WRN][engine_model.c:72]
[adapt_queue_depth] adapting queue depth to 128.
[11:00:31:928926][2328673][DOCA][WRN][engine_port.c:321]
[port_driver_process_properties] detected representor used in VNF
mode (driver port id 0)
EAL: Probe PCI driver: gpu_cuda (10de:20b5) device: 0000:ca:00.0
(socket 1)
[11:00:31:977261][2328673][DOCA][INF]
[gpunetio_simple_receive_sample.c:425][create_rxq] Creating
Sample Eth Rxq

[11:00:31:977841][2328673][DOCA][INF]
[gpunetio_simple_receive_sample.c:466][create_rxq] Mapping
receive queue buffer (0x0x7f86cc000000 size 33554432B) with
nvidia-peermem mode
[11:00:32:043182][2328673][DOCA][INF]
[gpunetio_simple_receive_sample.c:610][gpunetio_simple_receive]
Launching CUDA kernel to receive packets
[11:00:32:055193][2328673][DOCA][INF]
[gpunetio_simple_receive_sample.c:614][gpunetio_simple_receive]
Waiting for termination
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth
dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth
dst 10:70:fd:fa:77:e9
```

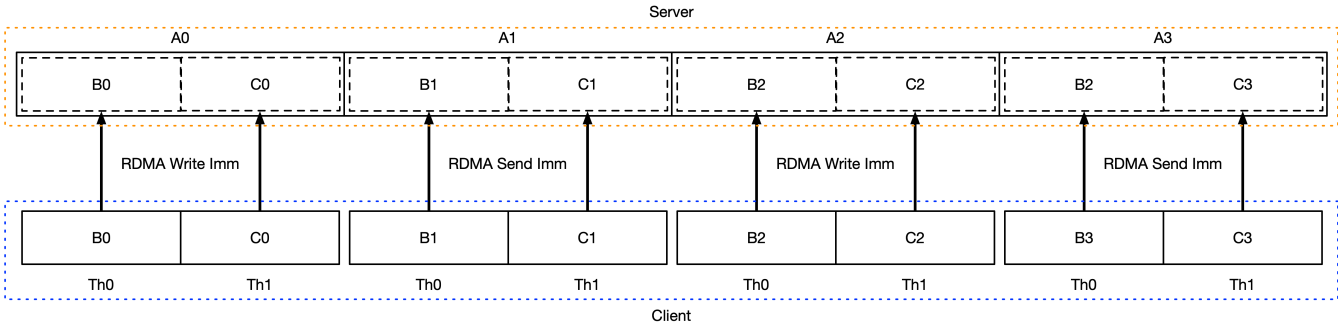
```
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth
dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth
dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth
dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth
dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth
dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth
dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth
dst 10:70:fd:fa:77:e9
Thread 0 received UDP packet with Eth src 10:70:fd:fa:77:f5 - Eth
dst 10:70:fd:fa:77:e9

# Type Ctrl+C to kill the sample

[11:01:44:265141][2328673][DOCA][INF]
[gpunetio_simple_receive_sample.c:45][signal_handler] Signal 2
received, preparing to exit!
[11:01:44:265189][2328673][DOCA][INF]
[gpunetio_simple_receive_sample.c:620][gpunetio_simple_receive]
Exiting from sample
[11:01:44:265533][2328673][DOCA][INF]
[gpunetio_simple_receive_sample.c:362][destroy_rxq] Destroying
Rxq
[11:01:44:307829][2328673][DOCA][INF]
[gpunetio_simple_receive_sample.c:631][gpunetio_simple_receive]
Sample finished successfully
[11:01:44:307861][2328673][DOCA][INF]
[gpunetio_simple_receive_main.c:204][main] Sample finished
successfully
```

RDMA Client Server

This sample exhibits how to use the GPUNetIO RDMA API to receive and send/write with immediate using a single RDMA queue.

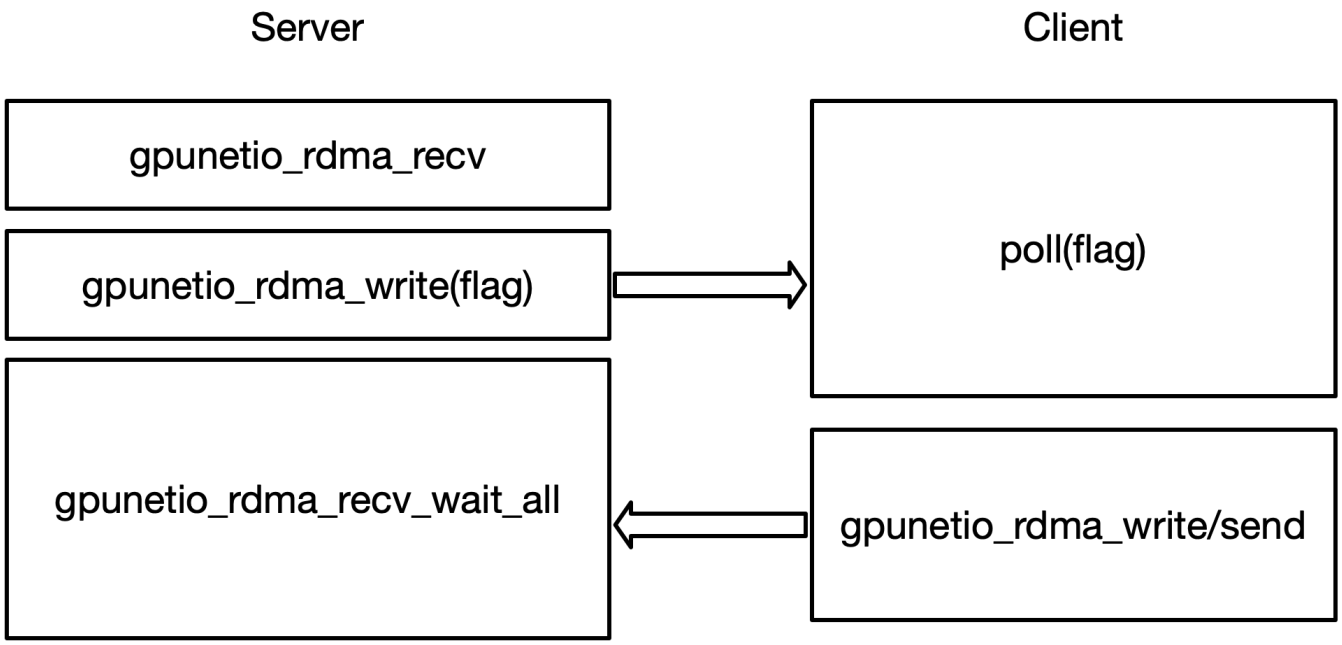
The server has a GPU buffer array A composed by `GPU_BUF_NUM doca_gpu_buf` elements, each 1kB in size. The client has two GPU buffer arrays, B and C, each composed by `GPU_BUF_NUM doca_gpu_buf` elements, each 512B in size.

The goal is for the client to fill a single server buffer of 1kB with two GPU buffers of 512B as illustrated in the following figure:



To show how to use RDMA write and send, even buffers are sent from the client with write immediate, while odd buffers are sent with send immediate. In both cases, the server must pre-post the RDMA receive operations.

For each buffer, the CUDA kernel code repeats the handshake:



Once all buffers are filled, the server double checks that all values are valid. The server output should be as follows:

```
# Ensure DOCA and DPDK are in the LD_LIBRARY_PATH environment variable
$
cd /opt/mellanox/doca/samples/doca_gpunetio/gpunetio_rdma_client_se
$ ./build/doca_gpunetio_rdma_client_server_write -gpu 17:00.0 -d
mlx5_0

[14:11:43:000930][1173110][DOCA][INF]
[gpunetio_rdma_client_server_write_main.c:250][main] Starting the
sample
...
[14:11:43:686610][1173110][DOCA][INF][rdma_common.c:91]
[oob_connection_server_setup] Listening for incoming connections
[14:11:45:681523][1173110][DOCA][INF][rdma_common.c:105]
[oob_connection_server_setup] Client connected at IP:
192.168.2.28 and port: 46274
...
[14:11:45:771807][1173110][DOCA][INF]
[gpunetio_rdma_client_server_write_sample.c:644]
[rdma_write_server] Before launching CUDA kernel, buffer array A
is:
[14:11:45:771822][1173110][DOCA][INF]
[gpunetio_rdma_client_server_write_sample.c:646]
[rdma_write_server] Buffer 0 -> offset 0: 1111 | offset 128: 1111
[14:11:45:771837][1173110][DOCA][INF]
[gpunetio_rdma_client_server_write_sample.c:646]
[rdma_write_server] Buffer 1 -> offset 0: 1111 | offset 128: 1111
[14:11:45:771851][1173110][DOCA][INF]
[gpunetio_rdma_client_server_write_sample.c:646]
[rdma_write_server] Buffer 2 -> offset 0: 1111 | offset 128: 1111
[14:11:45:771864][1173110][DOCA][INF]
[gpunetio_rdma_client_server_write_sample.c:646]
[rdma_write_server] Buffer 3 -> offset 0: 1111 | offset 128: 1111
```

```
RDMA Recv 2 ops completed with immediate values 0 and 1!
RDMA Recv 2 ops completed with immediate values 1 and 2!
RDMA Recv 2 ops completed with immediate values 2 and 3!
RDMA Recv 2 ops completed with immediate values 3 and 4!
[14:11:45:781561][1173110][DOCA][INF]
[gpunetio_rdma_client_server_write_sample.c:671]
[rdma_write_server] After launching CUDA kernel, buffer array A
is:
[14:11:45:781574][1173110][DOCA][INF]
[gpunetio_rdma_client_server_write_sample.c:673]
[rdma_write_server] Buffer 0 -> offset 0: 2222 | offset 128: 3333
[14:11:45:781583][1173110][DOCA][INF]
[gpunetio_rdma_client_server_write_sample.c:673]
[rdma_write_server] Buffer 1 -> offset 0: 2222 | offset 128: 3333
[14:11:45:781593][1173110][DOCA][INF]
[gpunetio_rdma_client_server_write_sample.c:673]
[rdma_write_server] Buffer 2 -> offset 0: 2222 | offset 128: 3333
[14:11:45:781602][1173110][DOCA][INF]
[gpunetio_rdma_client_server_write_sample.c:673]
[rdma_write_server] Buffer 3 -> offset 0: 2222 | offset 128: 3333
[14:11:45:781640][1173110][DOCA][INF]
[gpunetio_rdma_client_server_write_main.c:294][main] Sample
finished successfully
```

On the other side, assuming the server is at IP address `192.168.2.28`, the client output should be as follows:

```
# Ensure DOCA and DPDK are in the LD_LIBRARY_PATH environment variable

$
cd /opt/mellanox/doca/samples/doca_gpunetio/gpunetio_rdma_client_se
$ ./build/doca_gpunetio_rdma_client_server_write -gpu 17:00.0 -d
mlx5_0 -c 192.168.2.28
```

```
[16:08:22:335744][160913][DOCA][INF]
[gpunetio_rdma_client_server_write_main.c:197][main] Starting the
sample
...
[16:08:25:753316][160913][DOCA][INF][rdma_common.c:147]
[oob_connection_client_setup] Connected with server successfully
......
Client waiting on flag 7f6596735000 for server to post RDMA Recvs
Thread 0 post rdma write imm 0
Thread 1 post rdma write imm 0
Client waiting on flag 7f6596735001 for server to post RDMA Recvs
Thread 0 post rdma send imm 1
Thread 1 post rdma send imm 1
Client waiting on flag 7f6596735002 for server to post RDMA Recvs
Thread 0 post rdma write imm 2
Thread 1 post rdma write imm 2
Client waiting on flag 7f6596735003 for server to post RDMA Recvs
Thread 0 post rdma send imm 3
Thread 1 post rdma send imm 3
[16:08:25:853454][160913][DOCA][INF]
[gpunetio_rdma_client_server_write_main.c:241][main] Sample
finished successfully
```

Note

With RDMA, the network device must be specified by name (e.g., `mlx5_0`) instead of the PCIe address (as is the case for Ethernet).

It is also possible to enable the RDMA CM mode, establishing two connections with the same RDMA GPU handler. An example on the client side:

```
# Ensure DOCA and DPDK are in the LD_LIBRARY_PATH environment variable
```

```
$
cd /opt/mellanox/doca/samples/doca_gpunetio/gpunetio_rdma_client_se
$ ./build/samples/doca_gpunetio_rdma_client_server_write -d
mlx5_0 -gpu 17:00.0 -gid 3 -c 10.137.189.28 -cm --server-addr-type
ipv4 --server-addr 192.168.2.28

[11:30:34:489781][3853018][DOCA][INF]
[gpunetio_rdma_client_server_write_main.c:461][main] Starting the
sample
...
[11:30:35:038828][3853018][DOCA][INF]
[gpunetio_rdma_client_server_write_sample.c:950]
[rdma_write_client] Client is waiting for a connection
establishment
[11:30:35:082039][3853018][DOCA][INF]
[gpunetio_rdma_client_server_write_sample.c:963]
[rdma_write_client] Client - Connection 1 is established
...
[11:30:35:095282][3853018][DOCA][INF]
[gpunetio_rdma_client_server_write_sample.c:1006]
[rdma_write_client] Establishing connection 2..
[11:30:35:097521][3853018][DOCA][INF]
[gpunetio_rdma_client_server_write_sample.c:1016]
[rdma_write_client] Client is waiting for a connection
establishment
[11:30:35:102718][3853018][DOCA][INF]
[gpunetio_rdma_client_server_write_sample.c:1029]
[rdma_write_client] Client - Connection 2 is established
[11:30:35:102783][3853018][DOCA][INF]
[gpunetio_rdma_client_server_write_sample.c:1046]
[rdma_write_client] Client, terminate kernels
Client waiting on flag 7f16067b5000 for server to post RDMA Recvs
Thread 0 post rdma write imm 0
Thread 1 post rdma write imm 1
Client waiting on flag 7f16067b5001 for server to post RDMA Recvs
```

```
Thread 0 post rdma send imm 1
Thread 1 post rdma send imm 2
Client waiting on flag 7f16067b5002 for server to post RDMA Recvs
Thread 0 post rdma write imm 2
Thread 1 post rdma write imm 3
Client waiting on flag 7f16067b5003 for server to post RDMA Recvs
Thread 0 post rdma send imm 3
Thread 1 post rdma send imm 4
Client posted and completed 4 RDMA commits on connection 0.
Waiting on the exit flag.
Client waiting on flag 7f16067b5000 for server to post RDMA Recvs
Thread 0 post rdma write imm 0
Thread 1 post rdma write imm 1
Client waiting on flag 7f16067b5001 for server to post RDMA Recvs
Thread 0 post rdma send imm 1
Thread 1 post rdma send imm 2
Client waiting on flag 7f16067b5002 for server to post RDMA Recvs
Thread 0 post rdma write imm 2
Thread 1 post rdma write imm 3
Client waiting on flag 7f16067b5003 for server to post RDMA Recvs
Thread 0 post rdma send imm 3
Thread 1 post rdma send imm 4
Client posted and completed 4 RDMA commits on connection 1.
Waiting on the exit flag.
[11:30:35:122448][3853018][DOCA][INF]
[gpunetio_rdma_client_server_write_main.c:512][main] Sample
finished successfully
```

In case of RDMA CM, the command option `-cm` must be specified on the server side.

> ⚠ **Warning**

Printing from a CUDA kernel is not recommended for performance. It may make sense for debugging purposes and for simple samples like this one.

GPU DMA Copy

This sample exhibits how to use the DOCA DMA and DOCA GPUNetIO libraries to DMA copy a memory buffer from the CPU to the GPU (with DOCA DMA CPU functions) and from the GPU to the CPU (with DOCA GPUNetIO DMA device functions) from a CUDA kernel. This sample requires a DPU as it uses the DMA engine on it.

```
$ cd /opt/mellanox/doca/samples/doca_gpunetio/gpunetio_dma_memcpy

# Build the sample and then execute

$ ./build/doca_gpunetio_dma_memcpy -g 17:00.0 -n ca:00.0
[15:44:04:189462][862197][DOCA][INF]
[gpunetio_dma_memcpy_main.c:164][main] Starting the sample
EAL: Detected CPU lcores: 64
EAL: Detected NUMA nodes: 2
EAL: Detected shared linkage of DPDK
EAL: Selected IOVA mode 'VA'
EAL: No free 2048 kB hugepages reported on node 0
EAL: No free 2048 kB hugepages reported on node 1
EAL: VFIO support initialized
TELEMETRY: No legacy callbacks, legacy socket not created
EAL: Probe PCI driver: gpu_cuda (10de:2331) device: 0000:17:00.0
(socket 0)
[15:44:04:857251][862197][DOCA][INF]
[gpunetio_dma_memcpy_sample.c:211][init_sample_mem_objs] The CPU
source buffer value to be copied to GPU memory: This is a sample
piece of text from CPU
```

```
[15:44:04:857359][862197][DOCA][WRN][doca_mmap.cpp:1743]
[doca_mmap_set_memrange] Mmap 0x55aec6206140: Memory range isn't
cache-line aligned - addr=0x55aec52ceb10. For best performance
align address to 64B
[15:44:04:858839][862197][DOCA][INF]
[gpunetio_dma_memcpy_sample.c:158][init_sample_mem_objs] The GPU
source buffer value to be copied to CPU memory: This is a sample
piece of text from GPU
[15:44:04:921702][862197][DOCA][INF]
[gpunetio_dma_memcpy_sample.c:570][submit_dma_memcpy_task]
Success, DMA memcpy job done successfully
CUDA KERNEL INFO: The GPU destination buffer value after the
memcpy: This is a sample piece of text from CPU
CPU received message from GPU: This is a sample piece of text
from GPU
[15:44:04:930087][862197][DOCA][INF]
[gpunetio_dma_memcpy_sample.c:364][gpu_dma_cleanup] Cleanup DMA
ctx with GPU data path
[15:44:04:932658][862197][DOCA][INF]
[gpunetio_dma_memcpy_sample.c:404][gpu_dma_cleanup] Cleanup DMA
ctx with CPU data path
[15:44:04:954156][862197][DOCA][INF]
[gpunetio_dma_memcpy_main.c:197][main] Sample finished
successfully
```

failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

Trademarks

NVIDIA and the NVIDIA logo are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.
