# Table of contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VirtIO Acceleration through Hardware vDPA</td>
<td>5</td>
</tr>
<tr>
<td>Bridge Offload</td>
<td>11</td>
</tr>
<tr>
<td>Link Aggregation</td>
<td>13</td>
</tr>
<tr>
<td>Controlling Host PF and VF Parameters</td>
<td>24</td>
</tr>
</tbody>
</table>
NVIDIA® BlueField® and NVIDIA® ConnectX® platforms provide robust support for diverse applications through hardware-based offloads, offering unparalleled scalability, performance, and efficiency.

This section lists the extensive switching capabilities enabled by DOCA libraries and services on these platforms. It includes detailed configurations of Open Virtual Switch (OVS) such as the setup of representors, virtualization options, and optional bridge configurations. These subsections guide users through the steps to effectively implement these software components.

**DOCA Representors Model**

1. **Note**
   
   This model is only applicable when the BlueField is operating DPU mode.

BlueField® DPU uses netdev representors to map each one of the host side physical and virtual functions:

1. Serve as the tunnel to pass traffic for the virtual switch or application running on the Arm cores to the relevant PF or VF on the Arm side.

2. Serve as the channel to configure the embedded switch with rules to the corresponding represented function.

Those representors are used as the virtual ports being connected to OVS or any other virtual switch running on the Arm cores.

When in ECPF ownership mode, we see 2 representors for each one of the DPU's network ports: one for the uplink, and another one for the host side PF (the PF representor created even if the PF is not probed on the host side). For each one of the VFs created on the host side a corresponding representor would be created on the Arm side. The naming convention for the representors is as follows:

- Uplink representors: p<port_number>
- PF representors: pf<port_number>hpf
- VF representors: pf<port_number>vf<function_number>

The diagram below shows the mapping of between the PCIe functions exposed on the host side and the representors. For the sake of simplicity, we show a single port model (duplicated for the second port).

The red arrow demonstrates a packet flow through the representors, while the green arrow demonstrates the packet flow when steering rules are offloaded to the embedded switch. More details on that are available in the switch offload section.

Note

The MTU of host functions (PF/VF) must be smaller than the MTUs of both the uplink and corresponding PF/VF representor. For example, if
the host PF MTU is set to 9000, both uplink and PF representor must be set to above 9000.

This section contains the following pages:

- VirtIO Acceleration through Hardware vDPA
- Bridge Offload
- Link Aggregation
- Controlling Host PF and VF Parameters

**Note**

DOCA also provides OpenvSwitch Acceleration (OVS in DOCA) which implements a virtual switch service, designed to work with NVIDIA NICs and DPUs to utilize ASAP² (Accelerated Switching and Packet Processing) technology for data-path acceleration, providing the most efficient performance and feature set due to its architecture and use of DOCA libraries.
VirtIO Acceleration through Hardware vDPA

Hardware vDPA Installation

Hardware vDPA requires QEMU v2.12 (or with upstream 6.1.0) and DPDK v20.11 as minimal versions.

To install QEMU:

1. Clone the sources:

```bash
    git clone https://git.qemu.org/git/qemu.git
    cd qemu
    git checkout v2.12
```

2. Build QEMU:

```bash
    mkdir bin
    cd bin
    ../configure --target-list=x86_64-softmmu --enable-kvm
    make -j24
```

To install DPDK:

1. Clone the sources:

```bash
    git clone git://dpdk.org/dpdk
    cd dpdk
```
2. Install dependencies (if needed):

```
git checkout v20.11
yum install cmake gcc libnl3-devel libudev-devel make pkgconfig valgrind-devel pandoc libibverbs libmlx5 libmnl-devel -y
```

3. Configure DPDK:

```
export RTE_SDK=$PWD
make config T=x86_64-native-linuxapp-gcc
cd build
sed -i 's/\(CONFIG_RTE_LIBRTE_MLX5_PMD=\)n/\1y/g' .config
sed -i 's/\(CONFIG_RTE_LIBRTE_MLX5_VDPA_PMD=\)n/\1y/g' .config
make -j
```

4. Build DPDK:

```
make -j
```

5. Build the vDPA application:

```
cd $RTE_SDK/examples/vdpa/
make -j
```

### Hardware vDPA Configuration

To configure huge pages:

```
mkdir -p /hugepages
mount -t hugetlbfs hugetlbfs /hugepages
echo <more> > /sys/devices/system/node/node0/hugepages/hugepages-1048576kB/nr_hugepages
```
To configure a vDPA VirtIO interface in an existing VM's xml file (using libvirt):

1. Open the VM's configuration XML for editing:

```
virsh edit <domain name>
```

2. Perform the following:

1. Change the top line to:

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
```

2. Assign a memory amount and use 1GB page size for huge pages (size must be the same as that used for the vDPA application), so that the memory configuration looks as follows:

```
<memory unit='KiB'>4194304</memory>
<currentMemory unit='KiB'>4194304</currentMemory>
<memoryBacking>
    <hugepages>
        <page size='1048576' unit='KiB'/>
    </hugepages>
</memoryBacking>
```

3. Assign an amount of CPUs for the VM CPU configuration, so that the vcpu and cputune configuration looks as follows:

```
<vcpu placement='static'>5</vcpu>
<cputune>
    <vcpupin vcpu='0' cpuset='14'/>
    <vcpupin vcpu='1' cpuset='16'/>
    <vcpupin vcpu='2' cpuset='18'/>
```
4. Set the memory access for the CPUs to be shared, so that the \texttt{cpu} configuration looks as follows:

\begin{verbatim}
<cpu mode='custom' match='exact' check='partial'>
  <model fallback='allow'>Skylake-Server-IBRS</model>
  <numa>
    <cell id='0' cpus='0-4' memory='8388608' unit='KiB' memAccess='shared'/>
  </numa>
</cpu>
\end{verbatim}

5. Set the emulator in use to be the one built in \textbf{step 2}, so that the emulator configuration looks as follows:

\begin{verbatim}
<emulator><path to qemu executable></emulator>
\end{verbatim}

6. Add a virtio interface using QEMU command line argument entries, so that the new interface snippet looks as follows:

\begin{verbatim}
<qemu:commandline>
  <qemu:arg value='\-chardev'/>
  <qemu:arg value='socket,id=charnet1,path=/tmp/sock-virtio0'/>
  <qemu:arg value='\-netdev'>vhost-user,chardev=charnet1,queues=16,id=hostnet1'</qemu:arg>
  <qemu:arg value='\-device'>virtio-net-pci,mq=on,vectors=6,netdev=hostnet1,id=net1,mac=e4:11:c6:45:f2,addr=0x6, page-per-vq=on,rx_queue_size=1024,tx_queue_size=1024'</qemu:arg>
</qemu:commandline>
\end{verbatim}

\begin{tcolorbox}[colback=yellow!5!white,colframe=yellow,fonttitle=	tight,sharp corners]
\textbf{Note}
\end{tcolorbox}
Running Hardware vDPA

1. Create the ASAP² environment:
   1. Create the VFs.
   2. Enter switchdev mode.
   3. Set up OVS.

2. Run the vDPA application:

   ```bash
cd $RTE_SDK/examples/vdpa/build
./vdpa -w <VF PCI BDF>,class=vdpa --log-level=pmd,info -- -i
```

3. Create a vDPA port via the vDPA application CLI:

   ```bash
create /tmp/sock-virtio0 <PCI DEVICE BDF>
```

**Note**

Hardware vDPA supports switchdev mode only.

In this snippet, the vhostuser socket file path, the amount of queues, the MAC and the PCIe slot of the virtio device can be configured.
4. Start the VM:

```bash
virsh start <domain name>
```

For further information on the vDPA application, visit the Vdpa Sample Application DPDK documentation.
Bridge Offload

Note

Bridge offload is supported switchdev mode only.

Note

Bridge offload is supported from kernel version 5.15 onward.

A Linux bridge is an in-kernel software network switch (based on and implementing a subset of IEEE 802.1D standard) used to connect Ethernet segments together in a protocol-independent manner. Packets are forwarded based on L2 Ethernet header addresses.

mlx5 provides the ability to offload bridge dataplane unicast packet forwarding and VLAN management to hardware.

Basic Configuration

1. Initialize the ASAP² environment:
   1. Create the VFs.
   2. Enter switchdev mode.

2. Create a bridge and add mlx5 representors to bridge:

   ip link add name bridge0 type bridge
### Configuring VLAN

1. Enable VLAN filtering on the bridge:

   ```
   ip link set bridge0 type bridge vlan_filtering 1
   ```

2. Configure port VLAN matching (trunk mode). In this configuration, only packets with specified VID are allowed.

   ```
   bridge vlan add dev enp8s0f0_0 vid 2
   ```

3. Configure port VLAN tagging (access mode). In this configuration, VLAN header is pushed/popped upon reception/transmission on port.

   ```
   bridge vlan add dev enp8s0f0_0 vid 2 pvid untagged
   ```

### VF LAG Support

Bridge supports offloading on bond net device that is fully initialized with mlx5 uplink representors and is in single (shared) FDB LAG mode. Details about initialization of LAG are provided in section "SR-IOV VF LAG".

To add a bonding net device to bridge:

```
ip link set bond0 master bridge0
```

For further information on interacting with Linux bridge via iproute2 bridge tool, refer to `man 8 bridge`. 
Link Aggregation

Contents:

Network bonding enables combining two or more network interfaces into a single interface. It increases the network throughput, bandwidth and provides redundancy if one of the interfaces fails.

NVIDIA® BlueField® networking platforms (DPUs or SuperNICs) have an option to configure network bonding on the Arm side in a manner transparent to the host. Under such configuration, the host would only see a single PF.

Note

This functionality is supported when BlueField is set in embedded function ownership mode for both ports.

Note

While LAG is being configured (starting with step 2 under section "LAG Configuration"), traffic cannot pass through the physical ports.

The diagram below describes this configuration:
LAG Modes

Two LAG modes are supported on BlueField:

- Queue Affinity mode
- Hash mode

Queue Affinity Mode

In this mode, packets are distributed according to the QPs.

1. To enable this mode, run:

   ```
   $ mlxconfig -d /dev/mst/<device-name> s LAG_RESOURCE_ALLOCATION=0
   ```

   Example device name: mt41686_pciconf0.

2. Add/edit the following field from `/etc/mellanox/mlnx-bf.conf` as follows:
3. Perform a BlueField system reboot for the `mlxconfig` settings to take effect. Refer to the "NVIDIA BlueField Reset and Reboot Procedures" troubleshooting page for instructions.

### Hash Mode

In this mode, packets are distributed to ports according to the hash on packet headers.

#### Note

For this mode, prerequisite steps 3 and 4 are not required.

1. To enable this mode, run:

   ```sh
   $ mlxconfig -d /dev/mst/<device-name> s LAG_RESOURCE_ALLOCATION=1
   
   Example device name: mt41686_pconft0.
   
   2. Add/edit the following field from `/etc/mellanox/mlnx-bf.conf` as follows:

   ```
   LAG_HASH_MODE="yes"
   ```

3. Perform a BlueField system reboot for the `mlxconfig` settings to take effect. Refer to the "NVIDIA BlueField Reset and Reboot Procedures" troubleshooting page for instructions.
**Prerequisites**

1. Set the **LAG mode** to work with.

2. (Optional) Hide the second PF on the host. Run:

   ```
   $ mlxconfig -d /dev/mst/<device-name> s HIDE_PORT2_PF=True NUM_OF_PF=1
   
   Example device name: mt41686_pciconf0.
   ```

   **Note**

   Perform a BlueField system reboot for the `mlxconfig` settings to take effect. Refer to the "NVIDIA BlueField Reset and Reboot Procedures" troubleshooting page for instructions.

3. Delete any installed Scalable Functions (SFs) on the Arm side.

4. Stop the driver on the host side. Run:

   ```
   $ systemctl stop openibd
   ```

5. The uplink interfaces (p0 and p1) on the Arm side must be disconnected from any OVS bridge.

**LAG Configuration**

1. Create the bond interface. Run:

   ```
   $ ip link add bond0 type bond
   $ ip link set bond0 down
   ```
2. Subordinate both the uplink representors to the bond interface. Run:

```
$ ip link set p0 down
$ ip link set p1 down
$ ip link set p0 master bond0
$ ip link set p1 master bond0
```

3. Bring the interfaces up. Run:

```
$ ip link set p0 up
$ ip link set p1 up
$ ip link set bond0 up
```

The following is an example of LAG configuration in Ubuntu:

```
# cat /etc/network/interfaces

# interfaces(5) file used by ifup(8) and ifdown(8)
# Include files from /etc/network/interfaces.d:
source /etc/network/interfaces.d/*
auto lo
iface lo inet loopback
#p0
auto p0
iface p0 inet manual
    bond-master bond1
```
As a result, only the first PF of the BlueFields would be available to the host side for networking and SR-IOV.

⚠️ **Warning**

When in shared RQ mode (enabled by default), the uplink interfaces (p0 and p1) must always stay enabled. Disabling them will break LAG support and VF-to-VF communication on same host.

For OVS configuration, the bond interface is the one that needs to be added to the OVS bridge (interfaces p0 and p1 should not be added). The PF representor for the first port (pf0hpf) of the LAG must be added to the OVS bridge. The PF representor for the second port (pf1hpf) would still be visible, but it should not be added to OVS bridge. Consider the following examples:

```bash
#
#p1
auto p1
iface p1 inet manual
    bond-master bond1
#bond1
auto bond1
iface bond1 inet static
    address 192.168.1.1
    netmask 255.255.0.0
    mtu 1500
    bond-mode 2
    bond-slaves p0 p1
    bond-miimon 100
    pre-up (sleep 2 && ifup p0) &
    pre-up (sleep 2 && ifup p1) &
```

```
#bond1
auto bond1
iface bond1 inet static
    address 192.168.1.1
    netmask 255.255.0.0
    mtu 1500
    bond-mode 2
    bond-slaves p0 p1
    bond-miimon 100
    pre-up (sleep 2 && ifup p0) &
    pre-up (sleep 2 && ifup p1) &
```

```
Warning
When in shared RQ mode (enabled by default), the uplink interfaces (p0 and p1) must always stay enabled. Disabling them will break LAG support and VF-to-VF communication on same host.
```

```
for OVS configuration, the bond interface is the one that needs to be added to the OVS bridge (interfaces p0 and p1 should not be added). The PF representor for the first port (pf0hpf) of the LAG must be added to the OVS bridge. The PF representor for the second port (pf1hpf) would still be visible, but it should not be added to OVS bridge. Consider the following examples:

```
#bond1
auto bond1
iface bond1 inet static
    address 192.168.1.1
    netmask 255.255.0.0
    mtu 1500
    bond-mode 2
    bond-slaves p0 p1
    bond-miimon 100
    pre-up (sleep 2 && ifup p0) &
    pre-up (sleep 2 && ifup p1) &
```

```
For OVS configuration, the bond interface is the one that needs to be added to the OVS bridge (interfaces p0 and p1 should not be added). The PF representor for the first port (pf0hpf) of the LAG must be added to the OVS bridge. The PF representor for the second port (pf1hpf) would still be visible, but it should not be added to OVS bridge. Consider the following examples:

```
```

```
```

```
```
Warning

Trying to change bonding configuration in Queue Affinity mode (including bringing the subordinated interface up/down) while the host driver is loaded would cause FW syndrome and failure of the operation. Make sure to unload the host driver before altering BlueField bonding configuration to avoid this.

Note

When performing driver reload (openibd restart) or reboot, it is required to remove bond configuration and to reapply the configurations after the driver is fully up. Refer to steps 1-4 of "Removing LAG Configuration".

Removing LAG Configuration

1. If Queue Affinity mode LAG is configured (i.e., LAGRESOURCE_ALLOCATION=0):
   
   1. Delete any installed Scalable Functions (SFs) on the Arm side.

   2. Stop driver (openibd) on the host side. Run:

   systemctl stop openibd

2. Delete the LAG OVS bridge on the Arm side. Run:

   ovs-vsctl del-br bf-lag

   This allows for later restoration of OVS configuration for non-LAG networking.
3. Stop OVS service. Run:

```bash
systemctl stop openvswitch-switch.service
```

4. Run:

```bash
ip link set bond0 down
modprobe -rv bonding
```

As a result, both of the BlueField's network interfaces would be available to the host side for networking and SR-IOV.

5. For the host to be able to use BlueField's ports, make sure to attach the ECPF and host representor in an OVS bridge on the Arm side. Refer to "Virtual Switch on BlueField" for instructions on how to perform this.

6. Revert from \texttt{HIDE\_PORT2\_PF}, on the Arm side. Run:

```bash
mlxconfig -d /dev/mst/<device-name> s HIDE\_PORT2\_PF=False NUM\_OF\_PF=2
```

7. Restore default LAG settings in BlueField's firmware. Run:

```bash
mlxconfig -d /dev/mst/<device-name> s LAG\_RESOURCE\_ALLOCATION=DEVICE\_DEFAULT
```

8. Delete the following line from \texttt{/etc/mellanox/mlnx-bf.conf} on the Arm side:

```bash
LAG\_HASH\_MODE=...
```

9. Perform a BlueField system reboot for the \texttt{mlxconfig} settings to take effect. Refer to the "NVIDIA BlueField Reset and Reboot Procedures" troubleshooting page for instructions.
LAG on Multi-host

Only LAG hash mode is supported with BlueField multi-host.

LAG Multi-host Prerequisites

1. Enable LAG hash mode.

2. Hide the second PF on the host. Run:

   ```
   $ mlxconfig -d /dev/mst/<device-name> s HIDE_PORT2_PF=True NUM_OF_PF=1
   ```

3. Make sure NVME emulation is disabled:

   ```
   $ mlxconfig -d /dev/mst/<device-name> s NVME_EMULATION_ENABLE=0
   ```

   Example device name: mt41686_pciconf0.

4. The uplink interfaces (p0 and p4) on the Arm side, representing port0 and port1, must be disconnected from any OVS bridge. As a result, only the first PF of BlueField would be available to the host side for networking and SR-IOV.

LAG Configuration on Multi-host

1. Create the bond interface. Run:

   ```
   $ ip link add bond0 type bond
   $ ip link set bond0 down
   $ ip link set bond0 type bond miimon 100 mode 4 xmit_hash_policy layer3+4
   ```

2. Subordinate both the uplink representors to the bond interface. Run:
3. Bring the interfaces up. Run:

```bash
$ ip link set p0 down
$ ip link set p4 down
$ ip link set p0 master bond0
$ ip link set p4 master bond0
```

4. For OVS configuration, the bond interface is the one that must be added to the OVS bridge (interfaces p0 and p4 should not be added). The PF representor, pf0hpf, must be added to the OVS bridge with the bond interface. The rest of the uplink representors must be added to another OVS bridge along with their PF representors. Consider the following examples:

```bash
ovs-vsctl add-br br-lag
ovs-vsctl add-port br-lag bond0
ovs-vsctl add-port br-lag pf0hpf
ovs-vsctl add-br br1
ovs-vsctl add-port br1 p1
ovs-vsctl add-port br1 pf1hpf
ovs-vsctl add-br br2
ovs-vsctl add-port br2 p2
ovs-vsctl add-port br2 pf2hpf
ovs-vsctl add-br br3
ovs-vsctl add-port br3 p3
ovs-vsctl add-port br3 pf3hpf
```

Note
Removing LAG Configuration on Multi-host

Refer to section "Removing LAG Configuration".
Controlling Host PF and VF Parameters

Contents:

NVIDIA® BlueField® networking platforms (DPUs or SuperNICs) allow control over some of the networking parameters of the PFs and VFs running on the host side.

Setting Host PF and VF Default MAC Address

From the Arm, users may configure the MAC address of the physical function in the host. After sending the command, users must reload the NVIDIA driver in the host to see the newly configured MAC address. The MAC address goes back to the default value in the FW after system reboot.

Example:

```
$ echo "c4:8a:07:a5:29:59" > /sys/class/net/p0/smart_nic/pf/mac
$ echo "c4:8a:07:a5:29:61" > /sys/class/net/p0/smart_nic/vf0/mac
```

Setting Host PF and VF Link State

vPort state can be configured to Up, Down, or Follow. For example:

```
$ echo "Follow" > /sys/class/net/p0/smart_nic/pf/vport_state
```

Querying Configuration

To query the current configuration, run:
Zero signifies that the rate limit is unlimited.

### Disabling Host Networking PFs

It is possible to not expose ConnectX networking functions to the host for users interested in using storage or VirtIO functions only. When this feature is enabled, the host PF representors (i.e. `pf0hpf` and `pf1hpf`) will not be seen on the Arm.

- Without a PF on the host, it is not possible to enable SR-IOV, so VF representors will not be seen on the Arm either.
- Without PFs on the host, there can be no SFs on it.

To disable host networking PFs, run:

```bash
mlxconfig -d /dev/mst/mt41686_pciconf0 s NUM_OF_PF=0
```

To reactivate host networking PFs:

- For single-port BlueFields, run:
  ```bash
  mlxconfig -d /dev/mst/mt41686_pciconf0 s NUM_OF_PF=1
  ```

- For dual-port BlueFields, run:
  ```bash
  mlxconfig -d /dev/mst/mt41686_pciconf0 s NUM_OF_PF=2
  ```
Note

When there are no networking functions exposed on the host, the reactivation command must be run from the Arm.

Note

Perform a BlueField system reboot for the mlxconfig settings to take effect. Refer to the "NVIDIA BlueField Reset and Reboot Procedures" troubleshooting page for instructions.