OpenShift Container Platform 4.19

Hardware accelerators

Hardware accelerators

Last Updated: 2026-03-27

OpenShift Container Platform 4.19 Hardware accelerators

Hardware accelerators

Legal Notice

Abstract

This document provides instructions for installing and configuring the GPU Operators supported by Red Hat OpenShift AI for the provided hardware acceleration capabilities for creating artificial intelligence and machine learning (AI/ML) applications.

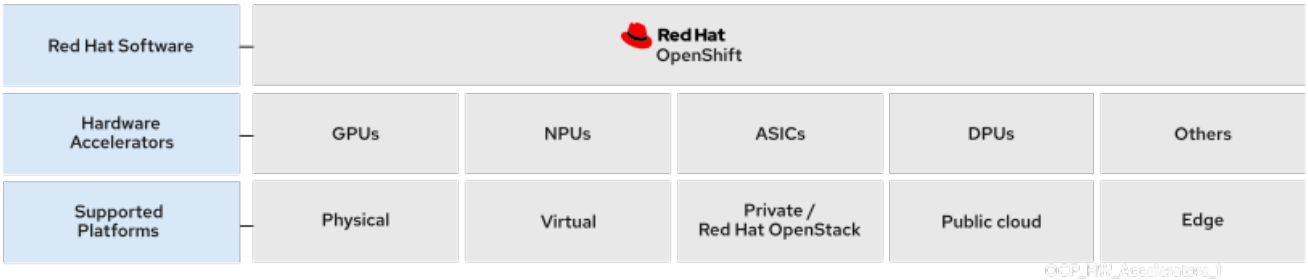
Table of Contents

CHAPTER 1. ABOUT HARDWARE ACCELERATORS

Specialized hardware accelerators play a key role in the emerging generative artificial intelligence and machine learning (AI/ML) industry. Specifically, hardware accelerators are essential to the training and serving of large language and other foundational models that power this new technology. Data scientists, data engineers, ML engineers, and developers can take advantage of the specialized hardware acceleration for data-intensive transformations and model development and serving. Much of that ecosystem is open source, with several contributing partners and open source foundations.

Red Hat OpenShift Container Platform provides support for cards and peripheral hardware that add processing units that comprise hardware accelerators:

- Graphical processing units (GPUs)
- Neural processing units (NPUs)
- Application-specific integrated circuits (ASICs)
- Data processing units (DPUs)



Specialized hardware accelerators provide a rich set of benefits for AI/ML development:

One platform for all

A collaborative environment for developers, data engineers, data scientists, and DevOps

Extended capabilities with Operators

Operators allow for bringing AI/ML capabilities to OpenShift Container Platform

Hybrid-cloud support

On-premise support for model development, delivery, and deployment

Support for AI/ML workloads

Model testing, iteration, integration, promotion, and serving into production as services

Red Hat provides an optimized platform to enable these specialized hardware accelerators in Red Hat Enterprise Linux (RHEL) and OpenShift Container Platform platforms at the Linux (kernel and userspace) and Kubernetes layers. To do this, Red Hat combines the proven capabilities of Red Hat OpenShift AI and Red Hat OpenShift Container Platform in a single enterprise-ready AI application platform.

Hardware Operators use the operating framework of a Kubernetes cluster to enable the required accelerator resources. You can also deploy the provided device plugin manually or as a daemon set. This plugin registers the GPU in the cluster.

Certain specialized hardware accelerators are designed to work within disconnected environments where a secure environment must be maintained for development and testing.

1.1. HARDWARE ACCELERATORS

Red Hat OpenShift Container Platform enables the following hardware accelerators:

- NVIDIA GPU
- AMD Instinct® GPU
- Intel® Gaudi®

Additional resources

- Introduction to Red Hat OpenShift AI
- NVIDIA GPU Operator on Red Hat OpenShift Container Platform
- AMD Instinct Accelerators
- Intel Gaudi AI Accelerators

CHAPTER 2. NVIDIA GPU ARCHITECTURE

NVIDIA supports the use of graphics processing unit (GPU) resources on OpenShift Container Platform. OpenShift Container Platform is a security-focused and hardened Kubernetes platform developed and supported by Red Hat for deploying and managing Kubernetes clusters at scale. OpenShift Container Platform includes enhancements to Kubernetes so that users can easily configure and use NVIDIA GPU resources to accelerate workloads.

The NVIDIA GPU Operator uses the Operator framework within OpenShift Container Platform to manage the full lifecycle of NVIDIA software components required to run GPU-accelerated workloads.

These components include the NVIDIA drivers (to enable CUDA), the Kubernetes device plugin for GPUs, the NVIDIA Container Toolkit, automatic node tagging using GPU feature discovery (GFD), DCGM-based monitoring, and others.

NOTE

The NVIDIA GPU Operator is only supported by NVIDIA. For more information about obtaining support from NVIDIA, see Obtaining Support from NVIDIA .

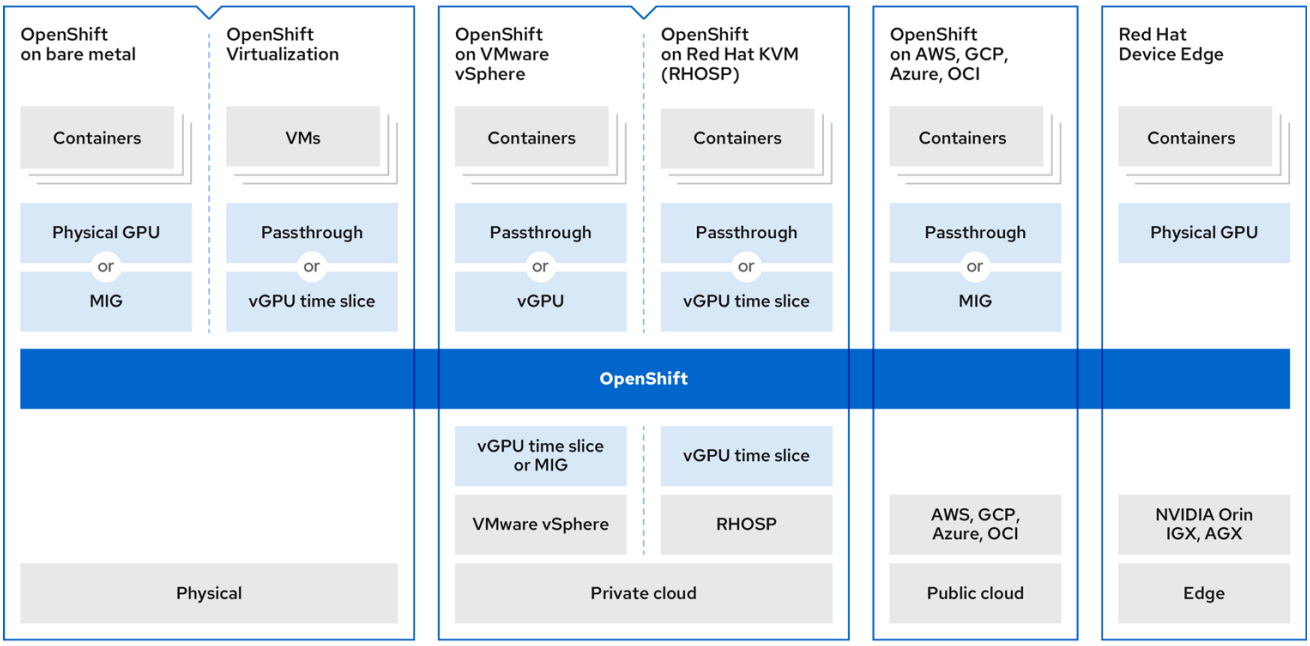
2.1. NVIDIA GPU PREREQUISITES

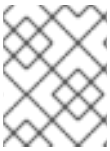
- A working OpenShift cluster with at least one GPU worker node.
- Access to the OpenShift cluster as a **cluster-admin** to perform the required steps.
- OpenShift CLI (**oc**) is installed.
- The node feature discovery (NFD) Operator is installed and a **nodefeaturediscovery** instance is created.

2.2. NVIDIA GPU ENABLEMENT

The following diagram shows how the GPU architecture is enabled for OpenShift:

Figure 2.1. NVIDIA GPU enablement





NOTE

MIG is supported on GPUs starting with the NVIDIA Ampere generation. For a list of GPUs that support MIG, see the NVIDIA MIG User Guide.

2.2.1. GPUs and bare metal

You can deploy OpenShift Container Platform on an NVIDIA-certified bare metal server but with some limitations:

- Control plane nodes can be CPU nodes.
- Worker nodes must be GPU nodes, provided that AI/ML workloads are executed on these worker nodes.
  In addition, the worker nodes can host one or more GPUs, but they must be of the same type. For example, a node can have two NVIDIA A100 GPUs, but a node with one A100 GPU and one T4 GPU is not supported. The NVIDIA Device Plugin for Kubernetes does not support mixing different GPU models on the same node.
- When using OpenShift, note that one or three or more servers are required. Clusters with two servers are not supported. The single server deployment is called single node openShift (SNO) and using this configuration results in a non-high availability OpenShift environment.

You can choose one of the following methods to access the containerized GPUs:

- GPU passthrough
- Multi-Instance GPU (MIG)

Additional resources

- Red Hat OpenShift on Bare Metal Stack

2.2.2. GPUs and virtualization

Many developers and enterprises are moving to containerized applications and serverless infrastructures, but there is still a lot of interest in developing and maintaining applications that run on virtual machines (VMs). Red Hat OpenShift Virtualization provides this capability, enabling enterprises to incorporate VMs into containerized workflows within clusters.

You can choose one of the following methods to connect the worker nodes to the GPUs:

- GPU passthrough to access and use GPU hardware within a virtual machine (VM).
- GPU (vGPU) time-slicing, when GPU compute capacity is not saturated by workloads.

Additional resources

- NVIDIA GPU Operator with OpenShift Virtualization

2.2.3. GPUs and vSphere

You can deploy OpenShift Container Platform on an NVIDIA-certified VMware vSphere server that can host different GPU types.

An NVIDIA GPU driver must be installed in the hypervisor in case vGPU instances are used by the VMs. For VMware vSphere, this host driver is provided in the form of a VIB file.

The maximum number of vGPUS that can be allocated to worker node VMs depends on the version of vSphere:

- vSphere 7.0: maximum 4 vGPU per VM
- vSphere 8.0: maximum 8 vGPU per VM

NOTE

vSphere 8.0 introduced support for multiple full or fractional heterogenous profiles associated with a VM.

You can choose one of the following methods to attach the worker nodes to the GPUs:

- GPU passthrough for accessing and using GPU hardware within a virtual machine (VM)
- GPU (vGPU) time-slicing, when not all of the GPU is needed

Similar to bare metal deployments, one or three or more servers are required. Clusters with two servers are not supported.

Additional resources

- OpenShift Container Platform on VMware vSphere with NVIDIA vGPUs

2.2.4. GPUs and Red Hat KVM

You can use OpenShift Container Platform on an NVIDIA-certified kernel-based virtual machine (KVM) server.

Similar to bare-metal deployments, one or three or more servers are required. Clusters with two servers are not supported.

However, unlike bare-metal deployments, you can use different types of GPUs in the server. This is because you can assign these GPUs to different VMs that act as Kubernetes nodes. The only limitation is that a Kubernetes node must have the same set of GPU types at its own level.

You can choose one of the following methods to access the containerized GPUs:

- GPU passthrough for accessing and using GPU hardware within a virtual machine (VM)
- GPU (vGPU) time-slicing when not all of the GPU is needed

To enable the vGPU capability, a special driver must be installed at the host level. This driver is delivered as a RPM package. This host driver is not required at all for GPU passthrough allocation.

2.2.5. GPUs and CSPs

You can deploy OpenShift Container Platform to one of the major cloud service providers (CSPs): Amazon Web Services (AWS), Google Cloud, or Microsoft Azure.

Two modes of operation are available: a fully managed deployment and a self-managed deployment.

- In a fully managed deployment, everything is automated by Red Hat in collaboration with CSP. You can request an OpenShift instance through the CSP web console, and the cluster is automatically created and fully managed by Red Hat. You do not have to worry about node failures or errors in the environment. Red Hat is fully responsible for maintaining the uptime of the cluster. The fully managed services are available on AWS, Azure, and Google Cloud. For AWS, the OpenShift service is called (Red Hat OpenShift Service on AWS). For Azure, the service is called Azure Red Hat OpenShift. For Google Cloud, the service is called OpenShift Dedicated on Google Cloud.
- In a self-managed deployment, you are responsible for instantiating and maintaining the OpenShift cluster. Red Hat provides the OpenShift-install utility to support the deployment of the OpenShift cluster in this case. The self-managed services are available globally to all CSPs.

It is important that this compute instance is a GPU-accelerated compute instance and that the GPU type matches the list of supported GPUs from NVIDIA AI Enterprise. For example, T4, V100, and A100 are part of this list.

You can choose one of the following methods to access the containerized GPUs:

- GPU passthrough to access and use GPU hardware within a virtual machine (VM).
- GPU (vGPU) time slicing when the entire GPU is not required.

Additional resources

- Red Hat Openshift in the Cloud

2.2.6. GPUs and Red Hat Device Edge

Red Hat Device Edge provides access to MicroShift. MicroShift provides the simplicity of a single-node deployment with the functionality and services you need for resource-constrained (edge) computing. Red Hat Device Edge meets the needs of bare-metal, virtual, containerized, or Kubernetes workloads deployed in resource-constrained environments.

You can enable NVIDIA GPUs on containers in a Red Hat Device Edge environment.

You use GPU passthrough to access the containerized GPUs.

Additional resources

- How to accelerate workloads with NVIDIA GPUs on Red Hat Device Edge

2.3. GPU SHARING METHODS

Red Hat and NVIDIA have developed GPU concurrency and sharing mechanisms to simplify GPU-accelerated computing on an enterprise-level OpenShift Container Platform cluster.

Applications typically have different compute requirements that can leave GPUs underutilized. Providing the right amount of compute resources for each workload is critical to reduce deployment cost and maximize GPU utilization.

Concurrency mechanisms for improving GPU utilization exist that range from programming model APIs to system software and hardware partitioning, including virtualization. The following list shows the GPU concurrency mechanisms:

- Compute Unified Device Architecture (CUDA) streams
- Time-slicing
- CUDA Multi-Process Service (MPS)
- Multi-instance GPU (MIG)
- Virtualization with vGPU

Consider the following GPU sharing suggestions when using the GPU concurrency mechanisms for different OpenShift Container Platform scenarios:

Bare metal
: vGPU is not available. Consider using MIG-enabled cards.

VMs
: vGPU is the best choice.

Older NVIDIA cards with no MIG on bare metal
: Consider using time-slicing.

VMs with multiple GPUs and you want passthrough and vGPU
: Consider using separate VMs.

Bare metal with OpenShift Virtualization and multiple GPUs
: Consider using pass-through for hosted VMs and time-slicing for containers.

Additional resources

- Improving GPU Utilization

2.3.1. CUDA streams

Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model developed by NVIDIA for general computing on GPUs.

A stream is a sequence of operations that executes in issue-order on the GPU. CUDA commands are typically executed sequentially in a default stream and a task does not start until a preceding task has completed.

Asynchronous processing of operations across different streams allows for parallel execution of tasks. A task issued in one stream runs before, during, or after another task is issued into another stream. This allows the GPU to run multiple tasks simultaneously in no prescribed order, leading to improved performance.

Additional resources

- Asynchronous Concurrent Execution

2.3.2. Time-slicing

GPU time-slicing interleaves workloads scheduled on overloaded GPUs when you are running multiple CUDA applications.

You can enable time-slicing of GPUs on Kubernetes by defining a set of replicas for a GPU, each of which can be independently distributed to a pod to run workloads on. Unlike multi-instance GPU (MIG), there is no memory or fault isolation between replicas, but for some workloads this is better than not sharing at all. Internally, GPU time-slicing is used to multiplex workloads from replicas of the same underlying GPU.

You can apply a cluster-wide default configuration for time-slicing. You can also apply node-specific configurations. For example, you can apply a time-slicing configuration only to nodes with Tesla T4 GPUs and not modify nodes with other GPU models.

You can combine these two approaches by applying a cluster-wide default configuration and then labeling nodes to give those nodes a node-specific configuration.

2.3.3. CUDA Multi-Process Service

CUDA Multi-Process Service (MPS) allows a single GPU to use multiple CUDA processes. The processes run in parallel on the GPU, eliminating saturation of the GPU compute resources. MPS also enables concurrent execution, or overlapping, of kernel operations and memory copying from different processes to enhance utilization.

Additional resources

- CUDA MPS

2.3.4. Multi-instance GPU

Using Multi-instance GPU (MIG), you can split GPU compute units and memory into multiple MIG instances. Each of these instances represents a standalone GPU device from a system perspective and can be connected to any application, container, or virtual machine running on the node. The software that uses the GPU treats each of these MIG instances as an individual GPU.

MIG is useful when you have an application that does not require the full power of an entire GPU. The MIG feature of the new NVIDIA Ampere architecture enables you to split your hardware resources into multiple GPU instances, each of which is available to the operating system as an independent CUDA-

enabled GPU.

NVIDIA GPU Operator version 1.7.0 and higher provides MIG support for the A100 and A30 Ampere cards. These GPU instances are designed to support up to seven multiple independent CUDA applications so that they operate completely isolated with dedicated hardware resources.

Additional resources

- NVIDIA Multi-Instance GPU User Guide

2.3.5. Virtualization with vGPU

Virtual machines (VMs) can directly access a single physical GPU using NVIDIA vGPU. You can create virtual GPUs that can be shared by VMs across the enterprise and accessed by other devices.

This capability combines the power of GPU performance with the management and security benefits provided by vGPU. Additional benefits provided by vGPU includes proactive management and monitoring for your VM environment, workload balancing for mixed VDI and compute workloads, and resource sharing across multiple VMs.

Additional resources

- Virtual GPUs

2.4. NVIDIA GPU FEATURES FOR OPENSHIFT CONTAINER PLATFORM

NVIDIA Container Toolkit

NVIDIA Container Toolkit enables you to create and run GPU-accelerated containers. The toolkit includes a container runtime library and utilities to automatically configure containers to use NVIDIA GPUs.

NVIDIA AI Enterprise

NVIDIA AI Enterprise is an end-to-end, cloud-native suite of AI and data analytics software optimized, certified, and supported with NVIDIA-Certified systems.
NVIDIA AI Enterprise includes support for Red Hat OpenShift Container Platform. The following installation methods are supported:

- OpenShift Container Platform on bare metal or VMware vSphere with GPU Passthrough.
- OpenShift Container Platform on VMware vSphere with NVIDIA vGPU.

GPU Feature Discovery

NVIDIA GPU Feature Discovery for Kubernetes is a software component that enables you to automatically generate labels for the GPUs available on a node. GPU Feature Discovery uses node feature discovery (NFD) to perform this labeling.
The Node Feature Discovery Operator (NFD) manages the discovery of hardware features and configurations in an OpenShift Container Platform cluster by labeling nodes with hardware-specific information. NFD labels the host with node-specific attributes, such as PCI cards, kernel, OS version, and so on.

You can find the NFD Operator in the Operator Hub by searching for “Node Feature Discovery”.

NVIDIA GPU Operator with OpenShift Virtualization

Up until this point, the GPU Operator only provisioned worker nodes to run GPU-accelerated containers. Now, the GPU Operator can also be used to provision worker nodes for running GPU-accelerated virtual machines (VMs).
You can configure the GPU Operator to deploy different software components to worker nodes depending on which GPU workload is configured to run on those nodes.

GPU Monitoring dashboard

You can install a monitoring dashboard to display GPU usage information on the cluster **Observe** page in the OpenShift Container Platform web console. GPU utilization information includes the number of available GPUs, power consumption (in watts), temperature (in degrees Celsius), utilization (in percent), and other metrics for each GPU.

Additional resources

- NVIDIA-Certified Systems
- NVIDIA AI Enterprise
- NVIDIA Container Toolkit
- Enabling the GPU Monitoring Dashboard
- MIG Support in OpenShift Container Platform
- Time-slicing NVIDIA GPUs in OpenShift
- Deploy GPU Operators in a disconnected or airgapped environment
- Node Feature Discovery Operator

CHAPTER 3. AMD GPU OPERATOR

AMD Instinct GPU accelerators combined with the AMD GPU Operator within your OpenShift Container Platform cluster lets you seamlessly harness computing capabilities for machine learning, Generative AI, and GPU-accelerated applications.

This documentation provides the information you need to enable, configure, and test the AMD GPU Operator. For more information, see AMD Instinct™ Accelerators.

3.1. ABOUT THE AMD GPU OPERATOR

The hardware acceleration capabilities of the AMD GPU Operator provide enhanced performance and cost efficiency for data scientists and developers using Red Hat OpenShift AI for creating artificial intelligence and machine learning (AI/ML) applications. Accelerating specific areas of GPU functions can minimize CPU processing and memory usage, improving overall application speed, memory consumption, and bandwidth restrictions.

3.2. INSTALLING THE AMD GPU OPERATOR

As a cluster administrator, you can install the AMD GPU Operator by using the OpenShift CLI and the web console. This is a multi-step procedure that requires the installation of the Node Feature Discovery Operator, the Kernel Module Management Operator, and then the AMD GPU Operator. Use the following steps in succession to install the AMD community release of the Operator.

Next steps

1. Install the Node Feature Discovery Operator.
2. Install the Kernel Module Management Operator.
3. Install and configure the AMD GPU Operator.

3.3. TESTING THE AMD GPU OPERATOR

Use the following procedure to test the ROCmInfo installation and view the logs for the AMD MI210 GPU.

Procedure

1. Create a YAML file that tests ROCmInfo:

```
$ cat << EOF > rocminfo.yaml

apiVersion: v1
kind: Pod
metadata:
 name: rocminfo
spec:
 containers:
 - image: docker.io/rocm/pytorch:latest
   name: rocminfo
   command: ["/bin/sh","-c"]
   args: ["rocminfo"]
   resources:
```

```
    limits:
      amd.com/gpu: 1
    requests:
      amd.com/gpu: 1
 restartPolicy: Never
EOF
```

2. Create the **rocminfo** pod:

```
$ oc create -f rocminfo.yaml
```

Example output

```
apiVersion: v1
pod/rocminfo created
```

3. Check the **rocmnfo** log with one MI210 GPU:

```
$ oc logs rocminfo | grep -A5 "Agent"
```

Example output

```
HSA Agents
==========
*******
Agent 1
*******
  Name:                    Intel(R) Xeon(R) Gold 6330 CPU @ 2.00GHz
  Uuid:                    CPU-XX
  Marketing Name:          Intel(R) Xeon(R) Gold 6330 CPU @ 2.00GHz
  Vendor Name:             CPU
--
Agent 2
*******
  Name:                    Intel(R) Xeon(R) Gold 6330 CPU @ 2.00GHz
  Uuid:                    CPU-XX
  Marketing Name:          Intel(R) Xeon(R) Gold 6330 CPU @ 2.00GHz
  Vendor Name:             CPU
--
Agent 3
*******
  Name:                    gfx90a
  Uuid:                    GPU-024b776f768a638b
  Marketing Name:          AMD Instinct MI210
  Vendor Name:             AMD
```

4. Delete the pod:

```
$ oc delete -f rocminfo.yaml
```

Example output

```
pod "rocminfo" deleted
```

CHAPTER 4. INTEL GAUDI AI ACCELERATORS

You can use Intel Gaudi AI accelerators for your OpenShift Container Platform generative AI and machine learning (AI/ML) applications. Intel Gaudi AI accelerators offer a cost-efficient, flexible, and scalable solution for optimized deep learning workloads.

Red Hat supports Intel Gaudi 2 and Intel Gaudi 3 devices. Intel Gaudi 3 devices provide significant improvements in training speed and energy efficiency.

4.1. INTEL GAUDI AI ACCELERATORS PREREQUISITES

- You have a working OpenShift Container Platform cluster with at least one GPU worker node.
- You have access to the OpenShift Container Platform cluster as a cluster-admin to perform the required steps.
- You have installed OpenShift CLI (**oc**).
- You have installed the Node Feature Discovery (NFD) Operator and created a **NodeFeatureDiscovery** instance.

Additional resources

- OpenShift Installation (Intel Gaudi documentation)
- Intel Gaudi AI Accelerator integration

CHAPTER 5. NVIDIA GPUDIRECT REMOTE DIRECT MEMORY ACCESS (RDMA)

NVIDIA GPUDirect Remote Direct Memory Access (RDMA) allows for an application in one computer to directly access the memory of another computer without needing access through the operating system. This provides the ability to bypass kernel intervention in the process, freeing up resources and greatly reducing the CPU overhead normally needed to process network communications. This is useful for distributing GPU-accelerated workloads across clusters. And because RDMA is so suited toward high bandwidth and low latency applications, this makes it ideal for big data and machine learning applications.

There are currently three configuration methods for NVIDIA GPUDirect RDMA:

Shared device

This method allows for an NVIDIA GPUDirect RDMA device to be shared among multiple pods on the OpenShift Container Platform worker node where the device is exposed.

Host device

This method provides direct physical Ethernet access on the worker node by creating an additional host network on a pod. A plugin allows the network device to be moved from the host network namespace to the network namespace on the pod.

SR-IOV legacy device

The Single Root IO Virtualization (SR-IOV) method can share a single network device, such as an Ethernet adapter, with multiple pods. SR-IOV segments the device, recognized on the host node as a physical function (PF), into multiple virtual functions (VFs). The VF is used like any other network device.

Each of these methods can be used across either the NVIDIA GPUDirect RDMA over Converged Ethernet (RoCE) or Infiniband infrastructures, providing an aggregate total of six methods of configuration.

5.1. NVIDIA GPUDIRECT RDMA PREREQUISITES

All methods of NVIDIA GPUDirect RDMA configuration require the installation of specific Operators. Use the following steps to install the Operators:

- Install the Node Feature Discovery Operator.
- Install the SR-IOV Operator.
- Install the NVIDIA Network Operator (NVIDIA documentation).
- Install the NVIDIA GPU Operator (NVIDIA documentation).

5.2. DISABLING THE IRDMA KERNEL MODULE

On some systems, including the DellR750xa, the IRDMA kernel module creates problems for the NVIDIA Network Operator when unloading and loading the DOCA drivers. Use the following procedure to disable the module.

Procedure

1. Generate the following machine configuration file by running the following command:
   -

```
$ cat <<EOF > 99-machine-config-blacklist-irdma.yaml
```

Example output

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  labels:
    machineconfiguration.openshift.io/role: worker
  name: 99-worker-blacklist-irdma
spec:
  kernelArguments:
    - "module_blacklist=irdma"
```

2. Create the machine configuration on the cluster and wait for the nodes to reboot by running the following command:

```
$ oc create -f 99-machine-config-blacklist-irdma.yaml
```

Example output

```
machineconfig.machineconfiguration.openshift.io/99-worker-blacklist-irdma created
```

3. Validate in a debug pod on each node that the module has not loaded by running the following command:

```
$ oc debug node/nvd-srv-32.nvidia.eng.rdu2.dc.redhat.com
Starting pod/nvd-srv-32nvidiaengrdu2dcredhatcom-debug-btfj2 ...
To use host binaries, run `chroot /host`
Pod IP: 10.6.135.11
If you don't see a command prompt, try pressing enter.
sh-5.1# chroot /host
sh-5.1# lsmod|grep irdma
sh-5.1#
```

5.3. CREATING PERSISTENT NAMING RULES

In some cases, device names won't persist following a reboot. For example, on R760xa systems Mellanox devices might be renamed after a reboot. You can avoid this problem by using a **MachineConfig** to set persistence.

Procedure

1. Gather the MAC address names from the worker nodes for the node into a file and provide names for the interfaces that need to persist. This example uses the file **70-persistent-net.rules** and stashes the details in it.

```
$ cat <<EOF > 70-persistent-net.rules
SUBSYSTEM=="net",ACTION=="add",ATTR{address}=="b8:3f:d2:3b:51:28",ATTR{type}=="1"
,NAME="ibs2f0"
SUBSYSTEM=="net",ACTION=="add",ATTR{address}=="b8:3f:d2:3b:51:29",ATTR{type}=="1"
,NAME="ens8f0np0"
SUBSYSTEM=="net",ACTION=="add",ATTR{address}=="b8:3f:d2:f0:36:d0",ATTR{type}=="1",
```

```
NAME="ibs2f0"
SUBSYSTEM=="net",ACTION=="add",ATTR{address}=="b8:3f:d2:f0:36:d1",ATTR{type}=="1",
NAME="ens8f0np0"
EOF
```

2. Convert that file into a base64 string without line breaks and set the output to the variable **PERSIST**:

```
$ PERSIST=`cat 70-persistent-net.rules| base64 -w 0`
```

```
$ echo $PERSIST
```

```
U1VCU1ITVEVNPT0ibmV0IixBQ1RJT049PSJhZGQiLEFUVFJ7YWRkcmVzc309PSJiODozZjp
kMjozYjo1MToyOCIsQVRUUnt0eXBlfT09IjEiLE5BTUU9ImliczJmMCIKU1VCU1ITVEVNPT0ibm
V0IixBQ1RJT049PSJhZGQiLEFUVFJ7YWRkcmVzc309PSJiODozZjpkMjozYjo1MToyOSIsQV
RUUnt0eXBlfT09IjEiLE5BTUU9ImVuczhmMG5wMCIKU1VCU1ITVEVNPT0ibmV0IixBQ1RJT0
49PSJhZGQiLEFUVFJ7YWRkcmVzc309PSJiODozZjpkMjpmMDozNjpkMCIsQVRUUnt0eXBlfT
09IjEiLE5BTUU9ImliczJmMCIKU1VCU1ITVEVNPT0ibmV0IixBQ1RJT049PSJhZGQiLEFUVFJ
7YWRkcmVzc309PSJiODozZjpkMjpmMDozNjpkMSIsQVRUUnt0eXBlfT09IjEiLE5BTUU9ImVuc
zhmMG5wMCIK
```

3. Create a machine configuration and set the base64 encoding in the custom resource file by running the following command:

```
$ cat <<EOF > 99-machine-config-udev-network.yaml
```

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  labels:
    machineconfiguration.openshift.io/role: worker
  name: 99-machine-config-udev-network
spec:
  config:
    ignition:
      version: 3.2.0
    storage:
      files:
      - contents:
          source: data:text/plain;base64,$PERSIST
        filesystem: root
        mode: 420
        path: /etc/udev/rules.d/70-persistent-net.rules
```

4. Create the machine configuration on the cluster by running the following command. After running the command, the expected output shows **machineconfig.machineconfiguration.openshift.io/99-machine-config-udev-network created**.

```
$ oc create -f 99-machine-config-udev-network.yaml
```

5. Use the **get mcp** command to view the machine configuration status:

```
$ oc get mcp
```

Example output

```
NAME     CONFIG                                             UPDATED   UPDATING   DEGRADED
MACHINECOUNT   READYMACHINECOUNT   UPDATEDMACHINECOUNT
DEGRADEDMACHINECOUNT   AGE
master   rendered-master-9adfe851c2c14d9598eea5ec3df6c187   True      False      False
1            1                 1                   0                      6h21m
worker   rendered-worker-4568f1b174066b4b1a4de794cf538fee   False     True       False
2            0                 0                   0                      6h21m
```

The nodes will reboot and when the updating field returns to **false**, you can validate on the nodes by looking at the devices in a debug pod.

5.4. CONFIGURING THE NFD OPERATOR

The Node Feature Discovery (NFD) Operator manages the detection of hardware features and configuration in an OpenShift Container Platform cluster by labeling the nodes with hardware-specific information. NFD labels the host with node-specific attributes, such as PCI cards, kernel, operating system version, and so on.

Prerequisites

- You have installed the NFD Operator.

Procedure

1. Validate that the Operator is installed and running by looking at the pods in the **openshift-nfd** namespace by running the following command:

```
$ oc get pods -n openshift-nfd
```

Example output

```
NAME                                  READY   STATUS    RESTARTS   AGE
nfd-controller-manager-8698c88cdd-t8gbc   2/2     Running   0          2m
```

2. With the NFD controller running, generate the **NodeFeatureDiscovery** instance and add it to the cluster.
   The **ClusterServiceVersion** specification for NFD Operator provides default values, including the NFD operand image that is part of the Operator payload. Retrieve its value by running the following command:

```
$ NFD_OPERAND_IMAGE=`echo $(oc get csv -n openshift-nfd -o json | jq -r
'.items[0].metadata.annotations["alm-examples"]') | jq -r '.[] | select(.kind ==
"NodeFeatureDiscovery") | .spec.operand.image'`
```

3. Optional: Add entries to the default **deviceClassWhiteList** field, to support more network adapters, such as the NVIDIA BlueField DPUs.

```
apiVersion: nfd.openshift.io/v1
```

```
kind: NodeFeatureDiscovery
metadata:
  name: nfd-instance
  namespace: openshift-nfd
spec:
  instance: ''
  operand:
    image: '${NFD_OPERAND_IMAGE}'
    servicePort: 12000
  prunerOnDelete: false
  topologyUpdater: false
  workerConfig:
    configData: |
      core:
        sleepInterval: 60s
      sources:
        pci:
          deviceClassWhitelist:
            - "02"
            - "03"
            - "0200"
            - "0207"
            - "12"
          deviceLabelFields:
            - "vendor"
```

4. Create the 'NodeFeatureDiscovery` instance by running the following command:

```
$ oc create -f nfd-instance.yaml
```

Example output

```
nodefeaturediscovery.nfd.openshift.io/nfd-instance created
```

5. Validate that the instance is up and running by looking at the pods under the **openshift-nfd** namespace by running the following command:

```
$ oc get pods -n openshift-nfd
```

Example output

```
NAME                                  READY   STATUS    RESTARTS   AGE
nfd-controller-manager-7cb6d656-jcnqb   2/2     Running   0          4m
nfd-gc-7576d64889-s28k9                 1/1     Running   0          21s
nfd-master-b7bcf5cfd-qnrmz              1/1     Running   0          21s
nfd-worker-96pfh                        1/1     Running   0          21s
nfd-worker-b2gkg                        1/1     Running   0          21s
nfd-worker-bd9bk                        1/1     Running   0          21s
nfd-worker-cswf4                        1/1     Running   0          21s
nfd-worker-kp6gg                        1/1     Running   0          21s
```

6. Wait a short period of time and then verify that NFD has added labels to the node. The NFD labels are prefixed with **feature.node.kubernetes.io**, so you can easily filter them.

```
$ oc get node -o json | jq '.items[0].metadata.labels | with_entries(select(.key |
startswith("feature.node.kubernetes.io")))'
{
  "feature.node.kubernetes.io/cpu-cpuid.ADX": "true",
  "feature.node.kubernetes.io/cpu-cpuid.AESNI": "true",
  "feature.node.kubernetes.io/cpu-cpuid.AVX": "true",
  "feature.node.kubernetes.io/cpu-cpuid.AVX2": "true",
  "feature.node.kubernetes.io/cpu-cpuid.CETSS": "true",
  "feature.node.kubernetes.io/cpu-cpuid.CLZERO": "true",
  "feature.node.kubernetes.io/cpu-cpuid.CMPXCHG8": "true",
  "feature.node.kubernetes.io/cpu-cpuid.CPBOOST": "true",
  "feature.node.kubernetes.io/cpu-cpuid.EFER_LMSLE_UNS": "true",
  "feature.node.kubernetes.io/cpu-cpuid.FMA3": "true",
  "feature.node.kubernetes.io/cpu-cpuid.FP256": "true",
  "feature.node.kubernetes.io/cpu-cpuid.FSRM": "true",
  "feature.node.kubernetes.io/cpu-cpuid.FXSR": "true",
  "feature.node.kubernetes.io/cpu-cpuid.FXSROPT": "true",
  "feature.node.kubernetes.io/cpu-cpuid.IBPB": "true",
  "feature.node.kubernetes.io/cpu-cpuid.IBRS": "true",
  "feature.node.kubernetes.io/cpu-cpuid.IBRS_PREFERRED": "true",
  "feature.node.kubernetes.io/cpu-cpuid.IBRS_PROVIDES_SMP": "true",
  "feature.node.kubernetes.io/cpu-cpuid.IBS": "true",
  "feature.node.kubernetes.io/cpu-cpuid.IBSBRNTRGT": "true",
  "feature.node.kubernetes.io/cpu-cpuid.IBSFETCHSAM": "true",
  "feature.node.kubernetes.io/cpu-cpuid.IBSFFV": "true",
  "feature.node.kubernetes.io/cpu-cpuid.IBSOPCNT": "true",
  "feature.node.kubernetes.io/cpu-cpuid.IBSOPCNTEXT": "true",
  "feature.node.kubernetes.io/cpu-cpuid.IBSOPSAM": "true",
  "feature.node.kubernetes.io/cpu-cpuid.IBSRDWROPCNT": "true",
  "feature.node.kubernetes.io/cpu-cpuid.IBSRIPINVALIDCHK": "true",
  "feature.node.kubernetes.io/cpu-cpuid.IBS_FETCH_CTLX": "true",
  "feature.node.kubernetes.io/cpu-cpuid.IBS_OPFUSE": "true",
  "feature.node.kubernetes.io/cpu-cpuid.IBS_PREVENTHOST": "true",
  "feature.node.kubernetes.io/cpu-cpuid.INT_WBINVD": "true",
  "feature.node.kubernetes.io/cpu-cpuid.INVLPGB": "true",
  "feature.node.kubernetes.io/cpu-cpuid.LAHF": "true",
  "feature.node.kubernetes.io/cpu-cpuid.LBRVIRT": "true",
  "feature.node.kubernetes.io/cpu-cpuid.MCAOVERFLOW": "true",
  "feature.node.kubernetes.io/cpu-cpuid.MCOMMIT": "true",
  "feature.node.kubernetes.io/cpu-cpuid.MOVBE": "true",
  "feature.node.kubernetes.io/cpu-cpuid.MOVU": "true",
  "feature.node.kubernetes.io/cpu-cpuid.MSRIRC": "true",
  "feature.node.kubernetes.io/cpu-cpuid.MSR_PAGEFLUSH": "true",
  "feature.node.kubernetes.io/cpu-cpuid.NRIPS": "true",
  "feature.node.kubernetes.io/cpu-cpuid.OSXSAVE": "true",
  "feature.node.kubernetes.io/cpu-cpuid.PPIN": "true",
  "feature.node.kubernetes.io/cpu-cpuid.PSFD": "true",
  "feature.node.kubernetes.io/cpu-cpuid.RDPRU": "true",
  "feature.node.kubernetes.io/cpu-cpuid.SEV": "true",
  "feature.node.kubernetes.io/cpu-cpuid.SEV_64BIT": "true",
  "feature.node.kubernetes.io/cpu-cpuid.SEV_ALTERNATIVE": "true",
  "feature.node.kubernetes.io/cpu-cpuid.SEV_DEBUGSWAP": "true",
  "feature.node.kubernetes.io/cpu-cpuid.SEV_ES": "true",
  "feature.node.kubernetes.io/cpu-cpuid.SEV_RESTRICTED": "true",
  "feature.node.kubernetes.io/cpu-cpuid.SEV_SNP": "true",
  "feature.node.kubernetes.io/cpu-cpuid.SHA": "true",
```

```
"feature.node.kubernetes.io/cpu-cpuid.SME": "true",
"feature.node.kubernetes.io/cpu-cpuid.SME_COHERENT": "true",
"feature.node.kubernetes.io/cpu-cpuid.SPEC_CTRL_SSBD": "true",
"feature.node.kubernetes.io/cpu-cpuid.SSE4A": "true",
"feature.node.kubernetes.io/cpu-cpuid.STIBP": "true",
"feature.node.kubernetes.io/cpu-cpuid.STIBP_ALWAYSON": "true",
"feature.node.kubernetes.io/cpu-cpuid.SUCCOR": "true",
"feature.node.kubernetes.io/cpu-cpuid.SVM": "true",
"feature.node.kubernetes.io/cpu-cpuid.SVMDA": "true",
"feature.node.kubernetes.io/cpu-cpuid.SVMFBASID": "true",
"feature.node.kubernetes.io/cpu-cpuid.SVML": "true",
"feature.node.kubernetes.io/cpu-cpuid.SVMNP": "true",
"feature.node.kubernetes.io/cpu-cpuid.SVMPF": "true",
"feature.node.kubernetes.io/cpu-cpuid.SVMPFT": "true",
"feature.node.kubernetes.io/cpu-cpuid.SYSCALL": "true",
"feature.node.kubernetes.io/cpu-cpuid.SYSEE": "true",
"feature.node.kubernetes.io/cpu-cpuid.TLB_FLUSH_NESTED": "true",
"feature.node.kubernetes.io/cpu-cpuid.TOPEXT": "true",
"feature.node.kubernetes.io/cpu-cpuid.TSCRATEMSR": "true",
"feature.node.kubernetes.io/cpu-cpuid.VAES": "true",
"feature.node.kubernetes.io/cpu-cpuid.VMCBCLEAN": "true",
"feature.node.kubernetes.io/cpu-cpuid.VMPL": "true",
"feature.node.kubernetes.io/cpu-cpuid.VMSA_REGPROT": "true",
"feature.node.kubernetes.io/cpu-cpuid.VPCLMULQDQ": "true",
"feature.node.kubernetes.io/cpu-cpuid.VTE": "true",
"feature.node.kubernetes.io/cpu-cpuid.WBNOINVD": "true",
"feature.node.kubernetes.io/cpu-cpuid.X87": "true",
"feature.node.kubernetes.io/cpu-cpuid.XGETBV1": "true",
"feature.node.kubernetes.io/cpu-cpuid.XSAVE": "true",
"feature.node.kubernetes.io/cpu-cpuid.XSAVEC": "true",
"feature.node.kubernetes.io/cpu-cpuid.XSAVEOPT": "true",
"feature.node.kubernetes.io/cpu-cpuid.XSAVES": "true",
"feature.node.kubernetes.io/cpu-hardware_multithreading": "false",
"feature.node.kubernetes.io/cpu-model.family": "25",
"feature.node.kubernetes.io/cpu-model.id": "1",
"feature.node.kubernetes.io/cpu-model.vendor_id": "AMD",
"feature.node.kubernetes.io/kernel-config.NO_HZ": "true",
"feature.node.kubernetes.io/kernel-config.NO_HZ_FULL": "true",
"feature.node.kubernetes.io/kernel-selinux.enabled": "true",
"feature.node.kubernetes.io/kernel-version.full": "5.14.0-427.35.1.el9_4.x86_64",
"feature.node.kubernetes.io/kernel-version.major": "5",
"feature.node.kubernetes.io/kernel-version.minor": "14",
"feature.node.kubernetes.io/kernel-version.revision": "0",
"feature.node.kubernetes.io/memory-numa": "true",
"feature.node.kubernetes.io/network-sriov.capable": "true",
"feature.node.kubernetes.io/pci-102b.present": "true",
"feature.node.kubernetes.io/pci-10de.present": "true",
"feature.node.kubernetes.io/pci-10de.sriov.capable": "true",
"feature.node.kubernetes.io/pci-15b3.present": "true",
"feature.node.kubernetes.io/pci-15b3.sriov.capable": "true",
"feature.node.kubernetes.io/rdma.available": "true",
"feature.node.kubernetes.io/rdma.capable": "true",
"feature.node.kubernetes.io/storage-nonrotationaldisk": "true",
"feature.node.kubernetes.io/system-os_release.ID": "rhcos",
"feature.node.kubernetes.io/system-os_release.OPENSHIFT_VERSION": "4.17",
"feature.node.kubernetes.io/system-os_release.OSTREE_VERSION":
```

```
"417.94.202409121747-0",
  "feature.node.kubernetes.io/system-os_release.RHEL_VERSION": "9.4",
  "feature.node.kubernetes.io/system-os_release.VERSION_ID": "4.17",
  "feature.node.kubernetes.io/system-os_release.VERSION_ID.major": "4",
  "feature.node.kubernetes.io/system-os_release.VERSION_ID.minor": "17"
}
```

7. Confirm there is a network device that is discovered:

```
$ oc describe node | grep -E 'Roles|pci' | grep pci-15b3
                    feature.node.kubernetes.io/pci-15b3.present=true
                    feature.node.kubernetes.io/pci-15b3.sriov.capable=true
                    feature.node.kubernetes.io/pci-15b3.present=true
                    feature.node.kubernetes.io/pci-15b3.sriov.capable=true
```

5.5. CONFIGURING THE SR-IOV OPERATOR

Single root I/O virtualization (SR-IOV) enhances the performance of NVIDIA GPUDirect RDMA by providing sharing across multiple pods from a single device.

Prerequisites

- You have installed the SR-IOV Operator.

Procedure

1. Validate that the Operator is installed and running by looking at the pods in the **openshift-sriov-network-operator** namespace by running the following command:

```
$ oc get pods -n openshift-sriov-network-operator
```

Example output

```
NAME                                     READY   STATUS    RESTARTS   AGE
sriov-network-operator-7cb6c49868-89486  1/1     Running   0          22s
```

2. For the default **SriovOperatorConfig** CR to work with the MLNX_OFED container, run this command to update the following values:

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovOperatorConfig
metadata:
  name: default
  namespace: openshift-sriov-network-operator
spec:
  enableInjector: true
  enableOperatorWebhook: true
  logLevel: 2
```

3. Create the resource on the cluster by running the following command:

```
$ oc create -f sriov-operator-config.yaml
```

Example output

```
sriovoperatorconfig.sriovnetwork.openshift.io/default created
```

4. Patch the sriov-operator so the MOFED container can work with it by running the following command:

```
$ oc patch sriovoperatorconfig default   --type=merge -n openshift-sriov-network-operator   --patch '{ "spec": { "configDaemonNodeSelector": { "network.nvidia.com/operator.mofed.wait": "false", "node-role.kubernetes.io/worker": "", "feature.node.kubernetes.io/pci-15b3.sriov.capable": "true" } } }'
```

Example output

```
sriovoperatorconfig.sriovnetwork.openshift.io/default patched
```

5.6. CONFIGURING THE NVIDIA NETWORK OPERATOR

The NVIDIA network Operator manages NVIDIA networking resources and networking related components such as drivers and device plugins to enable NVIDIA GPUDirect RDMA workloads.

Prerequisites

- You have installed the NVIDIA network Operator.

Procedure

1. Validate that the network Operator is installed and running by confirming the controller is running in the **nvidia-network-operator** namespace by running the following command:

```
$ oc get pods -n nvidia-network-operator
```

Example output

```
NAME                                                  READY   STATUS    RESTARTS   AGE
nvidia-network-operator-controller-manager-6f7d6956cd-fw5wg   1/1     Running   0          5m
```

2. With the Operator running, create the **NicClusterPolicy** custom resource file. The device you choose depends on your system configuration. In this example, the Infiniband interface **ibs2f0** is hard coded and is used as the shared NVIDIA GPUDirect RDMA device.

```
apiVersion: mellanox.com/v1alpha1
kind: NicClusterPolicy
metadata:
  name: nic-cluster-policy
spec:
  nicFeatureDiscovery:
    image: nic-feature-discovery
    repository: ghcr.io/mellanox
    version: v0.0.1
  docaTelemetryService:
```

```
    image: doca_telemetry
    repository: nvcr.io/nvidia/doca
    version: 1.16.5-doca2.6.0-host
  rdmaSharedDevicePlugin:
    config: |
      {
        "configList": [
          {
            "resourceName": "rdma_shared_device_ib",
            "rdmaHcaMax": 63,
            "selectors": {
              "ifNames": ["ibs2f0"]
            }
          },
          {
            "resourceName": "rdma_shared_device_eth",
            "rdmaHcaMax": 63,
            "selectors": {
              "ifNames": ["ens8f0np0"]
            }
          }
        ]
      }
    image: k8s-rdma-shared-dev-plugin
    repository: ghcr.io/mellanox
    version: v1.5.1
  secondaryNetwork:
    ipoib:
      image: ipoib-cni
      repository: ghcr.io/mellanox
      version: v1.2.0
  nvIpam:
    enableWebhook: false
    image: nvidia-k8s-ipam
    repository: ghcr.io/mellanox
    version: v0.2.0
  ofedDriver:
    readinessProbe:
      initialDelaySeconds: 10
      periodSeconds: 30
    forcePrecompiled: false
    terminationGracePeriodSeconds: 300
    livenessProbe:
      initialDelaySeconds: 30
      periodSeconds: 30
    upgradePolicy:
      autoUpgrade: true
      drain:
        deleteEmptyDir: true
        enable: true
        force: true
        timeoutSeconds: 300
        podSelector: ''
      maxParallelUpgrades: 1
      safeLoad: false
      waitForCompletion:
```

```
    timeoutSeconds: 0
  startupProbe:
    initialDelaySeconds: 10
    periodSeconds: 20
  image: doca-driver
  repository: nvcr.io/nvidia/mellanox
  version: 24.10-0.7.0.0-0
  env:
  - name: UNLOAD_STORAGE_MODULES
    value: "true"
  - name: RESTORE_DRIVER_ON_POD_TERMINATION
    value: "true"
  - name: CREATE_IFNAMES_UDEV
    value: "true"
```

3. Create the **NicClusterPolicy** custom resource on the cluster by running the following command:

```
$ oc create -f network-sharedrdma-nic-cluster-policy.yaml
```

Example output

```
nicclusterpolicy.mellanox.com/nic-cluster-policy created
```

4. Validate the **NicClusterPolicy** by running the following command in the DOCA/MOFED container:

```
$ oc get pods -n nvidia-network-operator
```

Example output

```
NAME                                                  READY   STATUS    RESTARTS   AGE
doca-telemetry-service-hwj65                          1/1     Running   2          160m
kube-ipoib-cni-ds-fsn8g                               1/1     Running   2          160m
mofed-rhcos4.16-9b5ddf4c6-ds-ct2h5                    2/2     Running   4          160m
nic-feature-discovery-ds-dtksz                        1/1     Running   2          160m
nv-ipam-controller-854585f594-c5jpp                   1/1     Running   2          160m
nv-ipam-controller-854585f594-xrnp5                   1/1     Running   2          160m
nv-ipam-node-xqttl                                    1/1     Running   2          160m
nvidia-network-operator-controller-manager-5798b564cd-5cq99   1/1     Running   2
5d23h
rdma-shared-dp-ds-p9vvg                               1/1     Running   0          85m
```

5. **rsh** into the **mofed** container to check the status by running the following command:

```
$ MOFED_POD=$(oc get pods -n nvidia-network-operator -o name | grep mofed)
$ oc rsh -n nvidia-network-operator -c mofed-container ${MOFED_POD}
sh-5.1# ofed_info -s
```

Example output

```
OFED-internal-24.07-0.6.1:
```

–

```
sh-5.1# ibdev2netdev -v
```

Example output

```
0000:0d:00.0 mlx5_0 (MT41692 - 900-9D3B4-00EN-EA0) BlueField-3 E-series SuperNIC
400GbE/NDR single port QSFP112, PCIe Gen5.0 x16 FHHL, Crypto Enabled, 16GB DDR5,
BMC, Tall Bracket                                    fw 32.42.1000 port 1 (ACTIVE) ==>
ibs2f0 (Up)
0000:a0:00.0 mlx5_1 (MT41692 - 900-9D3B4-00EN-EA0) BlueField-3 E-series SuperNIC
400GbE/NDR single port QSFP112, PCIe Gen5.0 x16 FHHL, Crypto Enabled, 16GB DDR5,
BMC, Tall Bracket                                    fw 32.42.1000 port 1 (ACTIVE) ==>
ens8f0np0 (Up)
```

6. Create a **IPoIBNetwork** custom resource file:

```
apiVersion: mellanox.com/v1alpha1
kind: IPoIBNetwork
metadata:
  name: example-ipoibnetwork
spec:
  ipam: |
    {
      "type": "whereabouts",
      "range": "192.168.6.225/28",
      "exclude": [
       "192.168.6.229/30",
       "192.168.6.236/32"
      ]
    }
  master: ibs2f0
  networkNamespace: default
```

7. Create the **IPoIBNetwork** resource on the cluster by running the following command:

```
$ oc create -f ipoib-network.yaml
```

Example output

```
ipoibnetwork.mellanox.com/example-ipoibnetwork created
```

8. Create a **MacvlanNetwork** custom resource file for your other interface:

```
apiVersion: mellanox.com/v1alpha1
kind: MacvlanNetwork
metadata:
  name: rdmashared-net
spec:
  networkNamespace: default
  master: ens8f0np0
  mode: bridge
  mtu: 1500
  ipam: '{"type": "whereabouts", "range": "192.168.2.0/24", "gateway": "192.168.2.1"}'
```

9. Create the resource on the cluster by running the following command:

```
$ oc create -f macvlan-network.yaml
```

Example output

```
macvlannetwork.mellanox.com/rdmashared-net created
```

5.7. CONFIGURING THE GPU OPERATOR

The GPU Operator automates the management of the NVIDIA drivers, device plugins for GPUs, the NVIDIA Container Toolkit, and other components required for GPU provisioning.

Prerequisites

- You have installed the GPU Operator.

Procedure

1. Check that the Operator pod is running to look at the pods under the namespace by running the following command:

```
$ oc get pods -n nvidia-gpu-operator
```

Example output

```
NAME                       READY   STATUS    RESTARTS   AGE
gpu-operator-b4cb7d74-zxpwq   1/1     Running   0          32s
```

2. Create a GPU cluster policy custom resource file similar to the following example:

```
apiVersion: nvidia.com/v1
kind: ClusterPolicy
metadata:
  name: gpu-cluster-policy
spec:
  vgpuDeviceManager:
    config:
      default: default
    enabled: true
  migManager:
    config:
      default: all-disabled
      name: default-mig-parted-config
    enabled: true
  operator:
    defaultRuntime: crio
    initContainer: {}
    runtimeClass: nvidia
    use_ocp_driver_toolkit: true
  dcgm:
    enabled: true
  gfd:
```

```
    enabled: true
  dcgmExporter:
    config:
      name: ''
    serviceMonitor:
      enabled: true
    enabled: true
  cdi:
    default: false
    enabled: false
  driver:
    licensingConfig:
      nlsEnabled: true
      configMapName: ''
    certConfig:
      name: ''
    rdma:
      enabled: false
    kernelModuleConfig:
      name: ''
    upgradePolicy:
      autoUpgrade: true
      drain:
        deleteEmptyDir: false
        enable: false
        force: false
        timeoutSeconds: 300
      maxParallelUpgrades: 1
      maxUnavailable: 25%
      podDeletion:
        deleteEmptyDir: false
        force: false
        timeoutSeconds: 300
      waitForCompletion:
        timeoutSeconds: 0
    repoConfig:
      configMapName: ''
    virtualTopology:
      config: ''
    enabled: true
    useNvidiaDriverCRD: false
    useOpenKernelModules: true
  devicePlugin:
    config:
      name: ''
      default: ''
    mps:
      root: /run/nvidia/mps
    enabled: true
  gdrcopy:
    enabled: true
  kataManager:
    config:
      artifactsDir: /opt/nvidia-gpu-operator/artifacts/runtimeclasses
  mig:
    strategy: single
```

```
  sandboxDevicePlugin:
    enabled: true
  validator:
    plugin:
      env:
        - name: WITH_WORKLOAD
          value: 'false'
  nodeStatusExporter:
    enabled: true
  daemonsets:
    rollingUpdate:
      maxUnavailable: '1'
    updateStrategy: RollingUpdate
  sandboxWorkloads:
    defaultWorkload: container
    enabled: false
  gds:
    enabled: true
    image: nvidia-fs
    version: 2.20.5
    repository: nvcr.io/nvidia/cloud-native
  vgpuManager:
    enabled: false
  vfioManager:
    enabled: true
  toolkit:
    installDir: /usr/local/nvidia
    enabled: true
```

3. When the GPU **ClusterPolicy** custom resource has generated, create the resource on the cluster by running the following command:

```
$ oc create -f gpu-cluster-policy.yaml
```

Example output

```
clusterpolicy.nvidia.com/gpu-cluster-policy created
```

4. Validate that the Operator is installed and running by running the following command:

```
$ oc get pods -n nvidia-gpu-operator
```

Example output

```
NAME                                  READY   STATUS      RESTARTS   AGE
gpu-feature-discovery-d5ngn           1/1     Running     0          3m20s
gpu-feature-discovery-z42rx           1/1     Running     0          3m23s
gpu-operator-6bb4d4b4c5-njh78         1/1     Running     0          4m35s
nvidia-container-toolkit-daemonset-bkh8l  1/1     Running     0          3m20s
nvidia-container-toolkit-daemonset-c4hzm  1/1     Running     0          3m23s
nvidia-cuda-validator-4blvg           0/1     Completed   0          106s
nvidia-cuda-validator-tw8sl           0/1     Completed   0          112s
nvidia-dcgm-exporter-rrw4g            1/1     Running     0          3m20s
nvidia-dcgm-exporter-xc78t            1/1     Running     0          3m23s
```

```
nvidia-dcgm-nvxpf                                  1/1     Running     0          3m20s
nvidia-dcgm-snj4j                                  1/1     Running     0          3m23s
nvidia-device-plugin-daemonset-fk2xz               1/1     Running     0          3m23s
nvidia-device-plugin-daemonset-wq87j               1/1     Running     0          3m20s
nvidia-driver-daemonset-416.94.202410211619-0-ngrjg   4/4     Running     0          3m58s
nvidia-driver-daemonset-416.94.202410211619-0-tm4x6   4/4     Running     0          3m58s
nvidia-node-status-exporter-jlzxh                  1/1     Running     0          3m57s
nvidia-node-status-exporter-zjffs                  1/1     Running     0          3m57s
nvidia-operator-validator-l49hx                    1/1     Running     0          3m20s
nvidia-operator-validator-n44nn                    1/1     Running     0          3m23s
```

5. Optional: When you have verified the pods are running, remote shell into the NVIDIA driver daemonset pod and confirm that the NVIDIA modules are loaded. Specifically, ensure the **nvidia_peermem** is loaded.

```
$ oc rsh -n nvidia-gpu-operator $(oc -n nvidia-gpu-operator get pod -o name -l
app.kubernetes.io/component=nvidia-driver)
sh-4.4# lsmod|grep nvidia
```

Example output

```
nvidia_fs            327680  0
nvidia_peermem        24576  0
nvidia_modeset      1507328  0
video                73728  1 nvidia_modeset
nvidia_uvm         6889472  8
nvidia             8810496  43 nvidia_uvm,nvidia_peermem,nvidia_fs,gdrdrv,nvidia_modeset
ib_uverbs           217088  3 nvidia_peermem,rdma_ucm,mlx5_ib
drm                741376  5 drm_kms_helper,drm_shmem_helper,nvidia,mgag200
```

6. Optional: Run the **nvidia-smi** utility to show the details about the driver and the hardware:

```
sh-4.4# nvidia-smi
```

+ .Example output

```
Wed Nov  6 22:03:53 2024
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.90.07              Driver Version: 550.90.07      CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================
========|
|   0  NVIDIA A40                 On  |   00000000:61:00.0 Off |                    0 |
|  0%   37C    P0           88W /  300W |       1MiB /  46068MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
|   1  NVIDIA A40                 On  |   00000000:E1:00.0 Off |                    0 |
|  0%   28C    P8           29W /  300W |       1MiB /  46068MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
```

```
+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================
========|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

1. While you are still in the driver pod, set the GPU clock to maximum using the **nvidia-smi** command:

```
$ oc rsh -n nvidia-gpu-operator nvidia-driver-daemonset-416.94.202410172137-0-ndhzc
sh-4.4# nvidia-smi -i 0 -lgc $(nvidia-smi -i 0 --query-supported-clocks=graphics --
format=csv,noheader,nounits | sort -h | tail -n 1)
```

Example output

```
GPU clocks set to "(gpuClkMin 1740, gpuClkMax 1740)" for GPU 00000000:61:00.0
All done.
```

```
sh-4.4# nvidia-smi -i 1 -lgc $(nvidia-smi -i 1 --query-supported-clocks=graphics --
format=csv,noheader,nounits | sort -h | tail -n 1)
```

Example output

```
GPU clocks set to "(gpuClkMin 1740, gpuClkMax 1740)" for GPU 00000000:E1:00.0
All done.
```

2. Validate the resource is available from a node describe perspective by running the following command:

```
$ oc describe node -l node-role.kubernetes.io/worker=| grep -E 'Capacity:|Allocatable:' -A9
```

Example output

```
Capacity:
  cpu:                128
  ephemeral-storage:          1561525616Ki
  hugepages-1Gi:              0
  hugepages-2Mi:              0
  memory:                     263596712Ki
  nvidia.com/gpu:             2
  pods:                       250
  rdma/rdma_shared_device_eth:  63
  rdma/rdma_shared_device_ib:   63
Allocatable:
  cpu:                127500m
  ephemeral-storage:          1438028263499
  hugepages-1Gi:              0
  hugepages-2Mi:              0
  memory:                     262445736Ki
  nvidia.com/gpu:             2
```

```
  pods:                    250
  rdma/rdma_shared_device_eth:  63
  rdma/rdma_shared_device_ib:   63
--
Capacity:
  cpu:                   128
  ephemeral-storage:           1561525616Ki
  hugepages-1Gi:               0
  hugepages-2Mi:               0
  memory:                  263596672Ki
  nvidia.com/gpu:              2
  pods:                    250
  rdma/rdma_shared_device_eth:  63
  rdma/rdma_shared_device_ib:   63
Allocatable:
  cpu:                   127500m
  ephemeral-storage:           1438028263499
  hugepages-1Gi:               0
  hugepages-2Mi:               0
  memory:                  262445696Ki
  nvidia.com/gpu:              2
  pods:                    250
  rdma/rdma_shared_device_eth:  63
  rdma/rdma_shared_device_ib:   63
```

5.8. CREATING THE MACHINE CONFIGURATION

Before you create the resource pods, you need to create the **machineconfig.yaml** custom resource (CR) that provides access to the GPU and networking resources without the need for user privileges.

Procedure

1. Generate a **Machineconfig** CR:

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  labels:
    machineconfiguration.openshift.io/role: worker
  name: 02-worker-container-runtime
spec:
  config:
    ignition:
      version: 3.2.0
    storage:
      files:
      - contents:
          source: data:text/plain;charset=utf-
8;base64,W2NyaW8ucnVudGltZV0KZGVmYXVsdF91bGltaXRzID0gWwoibWVtbG9jaz0tMTot
MSIKXQo=
        mode: 420
        overwrite: true
        path: /etc/crio/crio.conf.d/10-custom
```

5.9. CREATING THE WORKLOAD PODS

Use the procedures in this section to create the workload pods for the shared and host devices.

5.9.1. Creating a shared device RDMA on RoCE

Create the workload pods for a shared device RDMA on RDMA over Converged Ethernet (RoCE) for the NVIDIA Network Operator and test the pod configuration.

The NVIDIA GPUDirect RDMA device is shared among pods on the OpenShift Container Platform worker node where the device is exposed.

Prerequisites

- Ensure that the Operator is running.
- Delete the **NicClusterPolicy** custom resource (CR), if it exists.

Procedure

1. Generate custom pod resources:

```
$ cat <<EOF > rdma-eth-32-workload.yaml
apiVersion: v1
kind: Pod
metadata:
  name: rdma-eth-32-workload
  namespace: default
  annotations:
    k8s.v1.cni.cncf.io/networks: rdmashared-net
spec:
  nodeSelector:
    kubernetes.io/hostname: nvd-srv-32.nvidia.eng.rdu2.dc.redhat.com
  containers:
  - image: quay.io/edge-infrastructure/nvidia-tools:0.1.5
    name: rdma-eth-32-workload
    resources:
      limits:
        nvidia.com/gpu: 1
        rdma/rdma_shared_device_eth: 1
      requests:
        nvidia.com/gpu: 1
        rdma/rdma_shared_device_eth: 1

EOF

$ cat <<EOF > rdma-eth-33-workload.yaml
apiVersion: v1
kind: Pod
metadata:
  name: rdma-eth-33-workload
  namespace: default
  annotations:
    k8s.v1.cni.cncf.io/networks: rdmashared-net
spec:
```

```
  nodeSelector:
    kubernetes.io/hostname: nvd-srv-33.nvidia.eng.rdu2.dc.redhat.com
  containers:
  - image: quay.io/edge-infrastructure/nvidia-tools:0.1.5
    name: rdma-eth-33-workload
    securityContext:
      capabilities:
        add: [ "IPC_LOCK" ]
    resources:
      limits:
        nvidia.com/gpu: 1
        rdma/rdma_shared_device_eth: 1
      requests:
        nvidia.com/gpu: 1
        rdma/rdma_shared_device_eth: 1
EOF
```

2. Create the pods on the cluster by using the following commands:

```
$ oc create -f rdma-eth-32-workload.yaml
```

Example output

```
pod/rdma-eth-32-workload created
```

```
$ oc create -f rdma-eth-33-workload.yaml
```

Example output

```
pod/rdma-eth-33-workload created
```

3. Verify that the pods are running by using the following command:

```
$ oc get pods -n default
```

Example output

```
NAME                READY   STATUS    RESTARTS   AGE
rdma-eth-32-workload   1/1     Running   0          25s
rdma-eth-33-workload   1/1     Running   0          22s
```

5.9.2. Creating a host device RDMA on RoCE

Create the workload pods for a host device Remote Direct Memory Access (RDMA) for the NVIDIA Network Operator and test the pod configuration.

Prerequisites

- Ensure that the Operator is running.
- Delete the **NicClusterPolicy** custom resource (CR), if it exists.

Procedure

1. Generate a new host device **NicClusterPolicy** (CR), as shown below:

```
$ cat <<EOF > network-hostdev-nic-cluster-policy.yaml
apiVersion: mellanox.com/v1alpha1
kind: NicClusterPolicy
metadata:
  name: nic-cluster-policy
spec:
  ofedDriver:
    image: doca-driver
    repository: nvcr.io/nvidia/mellanox
    version: 24.10-0.7.0.0-0
    startupProbe:
      initialDelaySeconds: 10
      periodSeconds: 20
    livenessProbe:
      initialDelaySeconds: 30
      periodSeconds: 30
    readinessProbe:
      initialDelaySeconds: 10
      periodSeconds: 30
    env:
    - name: UNLOAD_STORAGE_MODULES
      value: "true"
    - name: RESTORE_DRIVER_ON_POD_TERMINATION
      value: "true"
    - name: CREATE_IFNAMES_UDEV
      value: "true"
  sriovDevicePlugin:
      image: sriov-network-device-plugin
      repository: ghcr.io/k8snetworkplumbingwg
      version: v3.7.0
      config: |
        {
          "resourceList": [
              {
                  "resourcePrefix": "nvidia.com",
                  "resourceName": "hostdev",
                  "selectors": {
                      "vendors": ["15b3"],
                      "isRdma": true
                  }
              }
          ]
        }
EOF
```

2. Create the **NicClusterPolicy** CR on the cluster by using the following command:

```
$ oc create -f network-hostdev-nic-cluster-policy.yaml
```

Example output

```
nicclusterpolicy.mellanox.com/nic-cluster-policy created
```

3. Verify that the host device **NicClusterPolicy** CR by using the following command in the DOCA/MOFED container:

```
$ oc get pods -n nvidia-network-operator
```

Example output

```
NAME                                                  READY   STATUS    RESTARTS   AGE
mofed-rhcos4.16-696886fcb4-ds-9sgvd                   2/2     Running   0          2m37s
mofed-rhcos4.16-696886fcb4-ds-lkjd4                   2/2     Running   0          2m37s
nvidia-network-operator-controller-manager-68d547dbbd-qsdkf   1/1     Running   0
141m
sriov-device-plugin-6v2nz                             1/1     Running   0          2m14s
sriov-device-plugin-hc4t8                             1/1     Running   0          2m14s
```

4. Confirm that the resources appear in the cluster **oc describe node** section by using the following command:

```
$ oc describe node -l node-role.kubernetes.io/worker=| grep -E 'Capacity:|Allocatable:' -A7
```

Example output

```
Capacity:
  cpu:                128
  ephemeral-storage:   1561525616Ki
  hugepages-1Gi:       0
  hugepages-2Mi:       0
  memory:             263596708Ki
  nvidia.com/hostdev:  2
  pods:               250
Allocatable:
  cpu:                127500m
  ephemeral-storage:   1438028263499
  hugepages-1Gi:       0
  hugepages-2Mi:       0
  memory:             262445732Ki
  nvidia.com/hostdev:  2
  pods:               250
--
Capacity:
  cpu:                128
  ephemeral-storage:   1561525616Ki
  hugepages-1Gi:       0
  hugepages-2Mi:       0
  memory:             263596704Ki
  nvidia.com/hostdev:  2
  pods:               250
Allocatable:
  cpu:                127500m
  ephemeral-storage:   1438028263499
  hugepages-1Gi:       0
  hugepages-2Mi:       0
```

```
  memory:             262445728Ki
  nvidia.com/hostdev:  2
  pods:               250
```

5. Create a **HostDeviceNetwork** CR file:

```
$ cat <<EOF >  hostdev-network.yaml
apiVersion: mellanox.com/v1alpha1
kind: HostDeviceNetwork
metadata:
  name: hostdev-net
spec:
  networkNamespace: "default"
  resourceName: "hostdev"
  ipam: |
    {
      "type": "whereabouts",
      "range": "192.168.3.225/28",
      "exclude": [
       "192.168.3.229/30",
       "192.168.3.236/32"
      ]
    }
EOF
```

6. Create the **HostDeviceNetwork** resource on the cluster by using the following command:

```
$ oc create -f hostdev-network.yaml
```

Example output

```
hostdevicenetwork.mellanox.com/hostdev-net created
```

7. Confirm that the resources appear in the cluster **oc describe node** section by using the following command:

```
$ oc describe node -l node-role.kubernetes.io/worker=| grep -E 'Capacity:|Allocatable:' -A8
```

Example output

```
Capacity:
  cpu:                128
  ephemeral-storage:   1561525616Ki
  hugepages-1Gi:       0
  hugepages-2Mi:       0
  memory:             263596708Ki
  nvidia.com/gpu:      2
  nvidia.com/hostdev:  2
  pods:               250
Allocatable:
  cpu:                127500m
  ephemeral-storage:   1438028263499
  hugepages-1Gi:       0
  hugepages-2Mi:       0
```

```
  memory:             262445732Ki
  nvidia.com/gpu:     2
  nvidia.com/hostdev: 2
  pods:               250
--
Capacity:
  cpu:                128
  ephemeral-storage:  1561525616Ki
  hugepages-1Gi:      0
  hugepages-2Mi:      0
  memory:             263596680Ki
  nvidia.com/gpu:     2
  nvidia.com/hostdev: 2
  pods:               250
Allocatable:
  cpu:                127500m
  ephemeral-storage:  1438028263499
  hugepages-1Gi:      0
  hugepages-2Mi:      0
  memory:             262445704Ki
  nvidia.com/gpu:     2
  nvidia.com/hostdev: 2
  pods:               250
```

5.9.3. Creating a SR-IOV legacy mode RDMA on RoCE

Configure a Single Root I/O Virtualization (SR-IOV) legacy mode host device RDMA on RoCE.

Procedure

1. Generate a new host device **NicClusterPolicy** custom resource (CR):

```
$ cat <<EOF > network-sriovleg-nic-cluster-policy.yaml
apiVersion: mellanox.com/v1alpha1
kind: NicClusterPolicy
metadata:
  name: nic-cluster-policy
spec:
  ofedDriver:
    image: doca-driver
    repository: nvcr.io/nvidia/mellanox
    version: 24.10-0.7.0.0-0
    startupProbe:
      initialDelaySeconds: 10
      periodSeconds: 20
    livenessProbe:
      initialDelaySeconds: 30
      periodSeconds: 30
    readinessProbe:
      initialDelaySeconds: 10
      periodSeconds: 30
    env:
    - name: UNLOAD_STORAGE_MODULES
      value: "true"
    - name: RESTORE_DRIVER_ON_POD_TERMINATION
```

```
      value: "true"
    - name: CREATE_IFNAMES_UDEV
      value: "true"
EOF
```

2. Create the policy on the cluster by using the following command:

```
$ oc create -f network-sriovleg-nic-cluster-policy.yaml
```

Example output

```
nicclusterpolicy.mellanox.com/nic-cluster-policy created
```

3. Verify the pods by using the following command in the DOCA/MOFED container:

```
$ oc get pods -n nvidia-network-operator
```

Example output

```
NAME                                                  READY   STATUS    RESTARTS      AGE
mofed-rhcos4.16-696886fcb4-ds-4mb42                     2/2     Running   0             40s
mofed-rhcos4.16-696886fcb4-ds-8knwq                     2/2     Running   0             40s
nvidia-network-operator-controller-manager-68d547dbbd-qsdkf   1/1     Running   13 (4d ago)
4d21h
```

4. Create an **SriovNetworkNodePolicy** CR that generates the Virtual Functions (VFs) for the device you want to operate in SR-IOV legacy mode. See the following example:

```
$ cat <<EOF > sriov-network-node-policy.yaml
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetworkNodePolicy
metadata:
  name: sriov-legacy-policy
  namespace:  openshift-sriov-network-operator
spec:
  deviceType: netdevice
  mtu: 1500
  nicSelector:
    vendor: "15b3"
    pfNames: ["ens8f0np0#0-7"]
  nodeSelector:
    feature.node.kubernetes.io/pci-15b3.present: "true"
  numVfs: 8
  priority: 90
  isRdma: true
  resourceName: sriovlegacy
EOF
```

5. Create the CR on the cluster by using the following command:

NOTE

Ensure that SR-IOV Global Enable is enabled. For more information, see Unable to enable SR-IOV and receiving the message "not enough MMIO resources for SR-IOV" in Red Hat Enterprise Linux.

```
$ oc create -f sriov-network-node-policy.yaml
```

Example output

```
sriovnetworknodepolicy.sriovnetwork.openshift.io/sriov-legacy-policy created
```

6. Each node has scheduling disabled. The nodes reboot to apply the configuration. You can view the nodes by using the following command:

```
$ oc get nodes
```

Example output

```
NAME                                   STATUS                     ROLES                  AGE
VERSION
edge-19.edge.lab.eng.rdu2.redhat.com   Ready                      control-
plane,master,worker   5d      v1.29.8+632b078
nvd-srv-32.nvidia.eng.rdu2.dc.redhat.com   Ready                      worker
4d22h   v1.29.8+632b078
nvd-srv-33.nvidia.eng.rdu2.dc.redhat.com   NotReady,SchedulingDisabled   worker
4d22h   v1.29.8+632b078
```

7. After the nodes have rebooted, verify that the VF interfaces exist by opening up a debug pod on each node. Run the following command:

```
a$ oc debug node/nvd-srv-33.nvidia.eng.rdu2.dc.redhat.com
```

Example output

```
Starting pod/nvd-srv-33nvidiaengrdu2dcredhatcom-debug-cqfjz ...
To use host binaries, run `chroot /host`
Pod IP: 10.6.135.12
If you don't see a command prompt, try pressing enter.
sh-5.1# chroot /host
sh-5.1# ip link show | grep ens8
26: ens8f0np0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP
mode DEFAULT group default qlen 1000
42: ens8f0v0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode
DEFAULT group default qlen 1000
43: ens8f0v1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode
DEFAULT group default qlen 1000
44: ens8f0v2: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode
DEFAULT group default qlen 1000
45: ens8f0v3: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode
DEFAULT group default qlen 1000
46: ens8f0v4: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode
DEFAULT group default qlen 1000
```

```
47: ens8f0v5: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode
DEFAULT group default qlen 1000
48: ens8f0v6: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode
DEFAULT group default qlen 1000
49: ens8f0v7: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode
DEFAULT group default qlen 1000
```

8. Repeat the previous steps on the second node, if necessary.

9. Optional: Confirm that the resources appear in the cluster **oc describe node** section by using the following command:

```
$ oc describe node -l node-role.kubernetes.io/worker=| grep -E 'Capacity:|Allocatable:' -A8
```

Example output

```
Capacity:
  cpu:                128
  ephemeral-storage:        1561525616Ki
  hugepages-1Gi:            0
  hugepages-2Mi:            0
  memory:                 263596692Ki
  nvidia.com/gpu:           2
  nvidia.com/hostdev:       0
  openshift.io/sriovlegacy:  8
--
Allocatable:
  cpu:                127500m
  ephemeral-storage:        1438028263499
  hugepages-1Gi:            0
  hugepages-2Mi:            0
  memory:                 262445716Ki
  nvidia.com/gpu:           2
  nvidia.com/hostdev:       0
  openshift.io/sriovlegacy:  8
--
Capacity:
  cpu:                128
  ephemeral-storage:        1561525616Ki
  hugepages-1Gi:            0
  hugepages-2Mi:            0
  memory:                 263596688Ki
  nvidia.com/gpu:           2
  nvidia.com/hostdev:       0
  openshift.io/sriovlegacy:  8
--
Allocatable:
  cpu:                127500m
  ephemeral-storage:        1438028263499
  hugepages-1Gi:            0
  hugepages-2Mi:            0
  memory:                 262445712Ki
  nvidia.com/gpu:           2
  nvidia.com/hostdev:       0
  openshift.io/sriovlegacy:  8
```

10. After the VFs for SR-IOV legacy mode are in place, generate the **SriovNetwork** CR file. See the following example:

```
$ cat <<EOF > sriov-network.yaml
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetwork
metadata:
  name: sriov-network
  namespace:  openshift-sriov-network-operator
spec:
  vlan: 0
  networkNamespace: "default"
  resourceName: "sriovlegacy"
  ipam: |
    {
      "type": "whereabouts",
      "range": "192.168.3.225/28",
      "exclude": [
       "192.168.3.229/30",
       "192.168.3.236/32"
      ]
    }
EOF
```

11. Create the custom resource on the cluster by using the following command:

```
$ oc create -f sriov-network.yaml
```

Example output

```
sriovnetwork.sriovnetwork.openshift.io/sriov-network created
```

5.9.4. Creating a shared device RDMA on Infiniband

Create the workload pods for a shared device Remote Direct Memory Access (RDMA) for an Infiniband installation.

Procedure

1. Generate custom pod resources:

```
$ cat <<EOF > rdma-ib-32-workload.yaml
apiVersion: v1
kind: Pod
metadata:
  name: rdma-ib-32-workload
  namespace: default
  annotations:
    k8s.v1.cni.cncf.io/networks: example-ipoibnetwork
spec:
  nodeSelector:
    kubernetes.io/hostname: nvd-srv-32.nvidia.eng.rdu2.dc.redhat.com
  containers:
  - image: quay.io/edge-infrastructure/nvidia-tools:0.1.5
```

```
    name: rdma-ib-32-workload
    resources:
      limits:
        nvidia.com/gpu: 1
        rdma/rdma_shared_device_ib: 1
      requests:
        nvidia.com/gpu: 1
        rdma/rdma_shared_device_ib: 1
EOF

$ cat <<EOF > rdma-ib-32-workload.yaml
apiVersion: v1
kind: Pod
metadata:
  name: rdma-ib-33-workload
  namespace: default
  annotations:
    k8s.v1.cni.cncf.io/networks: example-ipoibnetwork
spec:
  nodeSelector:
    kubernetes.io/hostname: nvd-srv-33.nvidia.eng.rdu2.dc.redhat.com
  containers:
  - image: quay.io/edge-infrastructure/nvidia-tools:0.1.5
    name: rdma-ib-33-workload
    securityContext:
      capabilities:
        add: [ "IPC_LOCK" ]
    resources:
      limits:
        nvidia.com/gpu: 1
        rdma/rdma_shared_device_ib: 1
      requests:
        nvidia.com/gpu: 1
        rdma/rdma_shared_device_ib: 1
EOF
```

2. Create the pods on the cluster by using the following commands:

```
$ oc create -f rdma-ib-32-workload.yaml
```

Example output

```
pod/rdma-ib-32-workload created
```

```
$ oc create -f rdma-ib-33-workload.yaml
```

Example output

```
pod/rdma-ib-33-workload created
```

3. Verify that the pods are running by using the following command:

```
$ oc get pods
```

Example output

```
NAME                  READY   STATUS    RESTARTS   AGE
rdma-ib-32-workload   1/1     Running   0          10s
rdma-ib-33-workload   1/1     Running   0          3s
```

5.10. VERIFYING RDMA CONNECTIVITY

Confirm Remote Direct Memory Access (RDMA) connectivity is working between the systems, specifically for Legacy Single Root I/O Virtualization (SR-IOV) Ethernet.

Procedure

1. Connect to each **rdma-workload-client** pod by using the following command:

```
$ oc rsh -n default rdma-sriov-32-workload
```

Example output

```
sh-5.1#
```

2. Check the IP address assigned to the first workload pod by using the following command. In this example, the first workload pod is the RDMA test server.

```
sh-5.1# ip a
```

Example output

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0@if3970: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1400 qdisc noqueue
state UP group default
    link/ether 0a:58:0a:80:02:a7 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.128.2.167/23 brd 10.128.3.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::858:aff:fe80:2a7/64 scope link
       valid_lft forever preferred_lft forever
3843: net1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP
group default qlen 1000
    link/ether 26:34:fd:53:a6:ec brd ff:ff:ff:ff:ff:ff
    altname enp55s0f0v5
    inet 192.168.4.225/28 brd 192.168.4.239 scope global net1
       valid_lft forever preferred_lft forever
    inet6 fe80::2434:fdff:fe53:a6ec/64 scope link
       valid_lft forever preferred_lft forever
sh-5.1#
```

The IP address of the RDMA server assigned to this pod is the **net1** interface. In this example, the IP address is **192.168.4.225**.

3. Run the **ibstatus** command to get the **link_layer** type, Ethernet or Infiniband, associated with each RDMA device **mlx5_x**. The output also shows the status of all of the RDMA devices by checking the **state** field, which shows either **ACTIVE** or **DOWN**.

```
sh-5.1# ibstatus
```

Example output

```
Infiniband device 'mlx5_0' port 1 status:
 default gid:  0000:0000:0000:0000:0000:0000:0000:0000
 base lid:  0x0
 sm lid:   0x0
 state:   4: ACTIVE
 phys state:  5: LinkUp
 rate:   200 Gb/sec (4X HDR)
 link_layer:  Ethernet

Infiniband device 'mlx5_1' port 1 status:
 default gid:  fe80:0000:0000:0000:e8eb:d303:0072:1415
 base lid:  0xc
 sm lid:   0x1
 state:   4: ACTIVE
 phys state:  5: LinkUp
 rate:   200 Gb/sec (4X HDR)
 link_layer:  InfiniBand

Infiniband device 'mlx5_2' port 1 status:
 default gid:  0000:0000:0000:0000:0000:0000:0000:0000
 base lid:  0x0
 sm lid:   0x0
 state:   1: DOWN
 phys state:  3: Disabled
 rate:   200 Gb/sec (4X HDR)
 link_layer:  Ethernet

Infiniband device 'mlx5_3' port 1 status:
 default gid:  0000:0000:0000:0000:0000:0000:0000:0000
 base lid:  0x0
 sm lid:   0x0
 state:   1: DOWN
 phys state:  3: Disabled
 rate:   200 Gb/sec (4X HDR)
 link_layer:  Ethernet

Infiniband device 'mlx5_4' port 1 status:
 default gid:  0000:0000:0000:0000:0000:0000:0000:0000
 base lid:  0x0
 sm lid:   0x0
 state:   1: DOWN
 phys state:  3: Disabled
 rate:   200 Gb/sec (4X HDR)
 link_layer:  Ethernet
```

```
Infiniband device 'mlx5_5' port 1 status:
 default gid:  0000:0000:0000:0000:0000:0000:0000:0000
 base lid:  0x0
 sm lid:   0x0
 state:   1: DOWN
 phys state:  3: Disabled
 rate:   200 Gb/sec (4X HDR)
 link_layer:  Ethernet

Infiniband device 'mlx5_6' port 1 status:
 default gid:  0000:0000:0000:0000:0000:0000:0000:0000
 base lid:  0x0
 sm lid:   0x0
 state:   1: DOWN
 phys state:  3: Disabled
 rate:   200 Gb/sec (4X HDR)
 link_layer:  Ethernet

Infiniband device 'mlx5_7' port 1 status:
 default gid:  fe80:0000:0000:0000:2434:fdff:fe53:a6ec
 base lid:  0x0
 sm lid:   0x0
 state:   4: ACTIVE
 phys state:  5: LinkUp
 rate:   200 Gb/sec (4X HDR)
 link_layer:  Ethernet

Infiniband device 'mlx5_8' port 1 status:
 default gid:  0000:0000:0000:0000:0000:0000:0000:0000
 base lid:  0x0
 sm lid:   0x0
 state:   1: DOWN
 phys state:  3: Disabled
 rate:   200 Gb/sec (4X HDR)
 link_layer:  Ethernet

Infiniband device 'mlx5_9' port 1 status:
 default gid:  0000:0000:0000:0000:0000:0000:0000:0000
 base lid:  0x0
 sm lid:   0x0
 state:   1: DOWN
 phys state:  3: Disabled
 rate:   200 Gb/sec (4X HDR)
 link_layer:  Ethernet

sh-5.1#
```

4. To get the **link_layer** for each RDMA **mlx5** device on your worker node, run the **ibstat** command:

```
sh-5.1# ibstat | egrep "Port|Base|Link"
```

Example output

```
Port 1:
  Physical state: LinkUp
  Base lid: 0
  Port GUID: 0x0000000000000000
  Link layer: Ethernet
 Port 1:
  Physical state: LinkUp
  Base lid: 12
  Port GUID: 0xe8ebd30300721415
  Link layer: InfiniBand
 Port 1:
  Base lid: 0
  Port GUID: 0x0000000000000000
  Link layer: Ethernet
 Port 1:
  Base lid: 0
  Port GUID: 0x0000000000000000
  Link layer: Ethernet
 Port 1:
  Base lid: 0
  Port GUID: 0x0000000000000000
  Link layer: Ethernet
 Port 1:
  Base lid: 0
  Port GUID: 0x0000000000000000
  Link layer: Ethernet
 Port 1:
  Base lid: 0
  Port GUID: 0x0000000000000000
  Link layer: Ethernet
 Port 1:
  Physical state: LinkUp
  Base lid: 0
  Port GUID: 0x2434fdfffe53a6ec
  Link layer: Ethernet
 Port 1:
  Base lid: 0
  Port GUID: 0x0000000000000000
  Link layer: Ethernet
 Port 1:
  Base lid: 0
  Port GUID: 0x0000000000000000
  Link layer: Ethernet
sh-5.1#
```

5. For RDMA Shared Device or Host Device workload pods, the RDMA device named **mlx5_x** is already known and is typically **mlx5_0** or **mlx5_1**. For RDMA Legacy SR-IOV workload pods, you need to determine which RDMA device is associated with which Virtual Function (VF) subinterface. Provide this information by using the following command:

```
sh-5.1# rdma link show
```

Example output

```
link mlx5_0/1 state ACTIVE physical_state LINK_UP
```

```
link mlx5_1/1 subnet_prefix fe80:0000:0000:0000 lid 12 sm_lid 1 lmc 0 state ACTIVE
physical_state LINK_UP
link mlx5_2/1 state DOWN physical_state DISABLED
link mlx5_3/1 state DOWN physical_state DISABLED
link mlx5_4/1 state DOWN physical_state DISABLED
link mlx5_5/1 state DOWN physical_state DISABLED
link mlx5_6/1 state DOWN physical_state DISABLED
link mlx5_7/1 state ACTIVE physical_state LINK_UP netdev net1
link mlx5_8/1 state DOWN physical_state DISABLED
link mlx5_9/1 state DOWN physical_state DISABLED
```

In this example, the RDMA device names **mlx5_7** is associated with the **net1** interface. This output is used in the next command to perform the RDMA bandwidth test, which also verifies RDMA connectivity between worker nodes.

6. Run the following **ib_write_bw** RDMA bandwidth test command:

```
sh-5.1# /root/perftest/ib_write_bw -R -T 41 -s 65536 -F -x 3 -m 4096 --report_gbits -q 16 -D
60  -d mlx5_7 -p 10000 --source_ip  192.168.4.225 --use_cuda=0 --use_cuda_dmabuf
```

where:

- The **mlx5_7** RDMA device is passed in the **-d** switch.
- The source IP address is **192.168.4.225** to start the RDMA server.
- The **--use_cuda=0**, **--use_cuda_dmabuf** switches indicate that the use of GPUDirect RDMA.

Example output

```
WARNING: BW peak won't be measured in this run.
Perftest doesn't supports CUDA tests with inline messages: inline size set to 0

************************************
* Waiting for client to connect... *
************************************
```

7. Open another terminal window and run **oc rsh** command on the second workload pod that acts as the RDMA test client pod:

```
$ oc rsh -n default rdma-sriov-33-workload
```

Example output

```
sh-5.1#
```

8. Obtain the RDMA test client pod IP address from the **net1** interface by using the following command:

```
sh-5.1# ip a
```

Example output

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0@if4139: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1400 qdisc noqueue
state UP group default
    link/ether 0a:58:0a:83:01:d5 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.131.1.213/23 brd 10.131.1.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::858:aff:fe83:1d5/64 scope link
       valid_lft forever preferred_lft forever
4076: net1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP
group default qlen 1000
    link/ether 56:6c:59:41:ae:4a brd ff:ff:ff:ff:ff:ff
    altname enp55s0f0v0
    inet 192.168.4.226/28 brd 192.168.4.239 scope global net1
       valid_lft forever preferred_lft forever
    inet6 fe80::546c:59ff:fe41:ae4a/64 scope link
       valid_lft forever preferred_lft forever
sh-5.1#
```

9. Obtain the **link_layer** type associated with each RDMA device **mlx5_x** by using the following command:

```
sh-5.1# ibstatus
```

Example output

```
Infiniband device 'mlx5_0' port 1 status:
 default gid:  0000:0000:0000:0000:0000:0000:0000:0000
 base lid:  0x0
 sm lid:   0x0
 state:   4: ACTIVE
 phys state:  5: LinkUp
 rate:   200 Gb/sec (4X HDR)
 link_layer:  Ethernet

Infiniband device 'mlx5_1' port 1 status:
 default gid:  fe80:0000:0000:0000:e8eb:d303:0072:09f5
 base lid:  0xd
 sm lid:   0x1
 state:   4: ACTIVE
 phys state:  5: LinkUp
 rate:   200 Gb/sec (4X HDR)
 link_layer:  InfiniBand

Infiniband device 'mlx5_2' port 1 status:
 default gid:  fe80:0000:0000:0000:546c:59ff:fe41:ae4a
 base lid:  0x0
 sm lid:   0x0
 state:   4: ACTIVE
 phys state:  5: LinkUp
```

```
 rate:   200 Gb/sec (4X HDR)
 link_layer:  Ethernet

Infiniband device 'mlx5_3' port 1 status:
 default gid:  0000:0000:0000:0000:0000:0000:0000:0000
 base lid:  0x0
 sm lid:   0x0
 state:   1: DOWN
 phys state:  3: Disabled
 rate:   200 Gb/sec (4X HDR)
 link_layer:  Ethernet

Infiniband device 'mlx5_4' port 1 status:
 default gid:  0000:0000:0000:0000:0000:0000:0000:0000
 base lid:  0x0
 sm lid:   0x0
 state:   1: DOWN
 phys state:  3: Disabled
 rate:   200 Gb/sec (4X HDR)
 link_layer:  Ethernet

Infiniband device 'mlx5_5' port 1 status:
 default gid:  0000:0000:0000:0000:0000:0000:0000:0000
 base lid:  0x0
 sm lid:   0x0
 state:   1: DOWN
 phys state:  3: Disabled
 rate:   200 Gb/sec (4X HDR)
 link_layer:  Ethernet

Infiniband device 'mlx5_6' port 1 status:
 default gid:  0000:0000:0000:0000:0000:0000:0000:0000
 base lid:  0x0
 sm lid:   0x0
 state:   1: DOWN
 phys state:  3: Disabled
 rate:   200 Gb/sec (4X HDR)
 link_layer:  Ethernet

Infiniband device 'mlx5_7' port 1 status:
 default gid:  0000:0000:0000:0000:0000:0000:0000:0000
 base lid:  0x0
 sm lid:   0x0
 state:   1: DOWN
 phys state:  3: Disabled
 rate:   200 Gb/sec (4X HDR)
 link_layer:  Ethernet

Infiniband device 'mlx5_8' port 1 status:
 default gid:  0000:0000:0000:0000:0000:0000:0000:0000
 base lid:  0x0
 sm lid:   0x0
 state:   1: DOWN
 phys state:  3: Disabled
 rate:   200 Gb/sec (4X HDR)
 link_layer:  Ethernet
```

```
Infiniband device 'mlx5_9' port 1 status:
 default gid:  0000:0000:0000:0000:0000:0000:0000:0000
 base lid:  0x0
 sm lid:   0x0
 state:   1: DOWN
 phys state:  3: Disabled
 rate:   200 Gb/sec (4X HDR)
 link_layer:  Ethernet
```

10. Optional: Obtain the firmware version of Mellanox cards by using the **ibstat** command:

```
sh-5.1# ibstat
```

Example output

```
CA 'mlx5_0'
 CA type: MT4123
 Number of ports: 1
 Firmware version: 20.43.1014
 Hardware version: 0
 Node GUID: 0xe8ebd303007209f4
 System image GUID: 0xe8ebd303007209f4
 Port 1:
  State: Active
  Physical state: LinkUp
  Rate: 200
  Base lid: 0
  LMC: 0
  SM lid: 0
  Capability mask: 0x00010000
  Port GUID: 0x0000000000000000
  Link layer: Ethernet
CA 'mlx5_1'
 CA type: MT4123
 Number of ports: 1
 Firmware version: 20.43.1014
 Hardware version: 0
 Node GUID: 0xe8ebd303007209f5
 System image GUID: 0xe8ebd303007209f4
 Port 1:
  State: Active
  Physical state: LinkUp
  Rate: 200
  Base lid: 13
  LMC: 0
  SM lid: 1
  Capability mask: 0xa651e848
  Port GUID: 0xe8ebd303007209f5
  Link layer: InfiniBand
CA 'mlx5_2'
 CA type: MT4124
 Number of ports: 1
 Firmware version: 20.43.1014
 Hardware version: 0
```

```
  Node GUID: 0x566c59fffe41ae4a
  System image GUID: 0xe8ebd303007209f4
  Port 1:
   State: Active
   Physical state: LinkUp
   Rate: 200
   Base lid: 0
   LMC: 0
   SM lid: 0
   Capability mask: 0x00010000
   Port GUID: 0x546c59fffe41ae4a
   Link layer: Ethernet
 CA 'mlx5_3'
  CA type: MT4124
  Number of ports: 1
  Firmware version: 20.43.1014
  Hardware version: 0
  Node GUID: 0xb2ae4bfffe8f3d02
  System image GUID: 0xe8ebd303007209f4
  Port 1:
   State: Down
   Physical state: Disabled
   Rate: 200
   Base lid: 0
   LMC: 0
   SM lid: 0
   Capability mask: 0x00010000
   Port GUID: 0x0000000000000000
   Link layer: Ethernet
 CA 'mlx5_4'
  CA type: MT4124
  Number of ports: 1
  Firmware version: 20.43.1014
  Hardware version: 0
  Node GUID: 0x2a9967fffe8bf272
  System image GUID: 0xe8ebd303007209f4
  Port 1:
   State: Down
   Physical state: Disabled
   Rate: 200
   Base lid: 0
   LMC: 0
   SM lid: 0
   Capability mask: 0x00010000
   Port GUID: 0x0000000000000000
   Link layer: Ethernet
 CA 'mlx5_5'
  CA type: MT4124
  Number of ports: 1
  Firmware version: 20.43.1014
  Hardware version: 0
  Node GUID: 0x5aff2ffffe2e17e8
  System image GUID: 0xe8ebd303007209f4
  Port 1:
   State: Down
   Physical state: Disabled
```

```
    Rate: 200
    Base lid: 0
    LMC: 0
    SM lid: 0
    Capability mask: 0x00010000
    Port GUID: 0x0000000000000000
    Link layer: Ethernet
  CA 'mlx5_6'
   CA type: MT4124
   Number of ports: 1
   Firmware version: 20.43.1014
   Hardware version: 0
   Node GUID: 0x121bf1fffe074419
   System image GUID: 0xe8ebd303007209f4
   Port 1:
    State: Down
    Physical state: Disabled
    Rate: 200
    Base lid: 0
    LMC: 0
    SM lid: 0
    Capability mask: 0x00010000
    Port GUID: 0x0000000000000000
    Link layer: Ethernet
  CA 'mlx5_7'
   CA type: MT4124
   Number of ports: 1
   Firmware version: 20.43.1014
   Hardware version: 0
   Node GUID: 0xb22b16fffed03dd7
   System image GUID: 0xe8ebd303007209f4
   Port 1:
    State: Down
    Physical state: Disabled
    Rate: 200
    Base lid: 0
    LMC: 0
    SM lid: 0
    Capability mask: 0x00010000
    Port GUID: 0x0000000000000000
    Link layer: Ethernet
  CA 'mlx5_8'
   CA type: MT4124
   Number of ports: 1
   Firmware version: 20.43.1014
   Hardware version: 0
   Node GUID: 0x523800fffe16d105
   System image GUID: 0xe8ebd303007209f4
   Port 1:
    State: Down
    Physical state: Disabled
    Rate: 200
    Base lid: 0
    LMC: 0
    SM lid: 0
    Capability mask: 0x00010000
```

```
  Port GUID: 0x0000000000000000
  Link layer: Ethernet
CA 'mlx5_9'
 CA type: MT4124
 Number of ports: 1
 Firmware version: 20.43.1014
 Hardware version: 0
 Node GUID: 0xd2b4a1fffebdc4a9
 System image GUID: 0xe8ebd303007209f4
 Port 1:
  State: Down
  Physical state: Disabled
  Rate: 200
  Base lid: 0
  LMC: 0
  SM lid: 0
  Capability mask: 0x00010000
  Port GUID: 0x0000000000000000
  Link layer: Ethernet
sh-5.1#
```

11. To determine which RDMA device is associated with the Virtual Function subinterface that the client workload pod uses, run the following command. In this example, the **net1** interface is using the RDMA device **mlx5_2**.

```
sh-5.1# rdma link show
```

Example output

```
link mlx5_0/1 state ACTIVE physical_state LINK_UP
link mlx5_1/1 subnet_prefix fe80:0000:0000:0000 lid 13 sm_lid 1 lmc 0 state ACTIVE
physical_state LINK_UP
link mlx5_2/1 state ACTIVE physical_state LINK_UP netdev net1
link mlx5_3/1 state DOWN physical_state DISABLED
link mlx5_4/1 state DOWN physical_state DISABLED
link mlx5_5/1 state DOWN physical_state DISABLED
link mlx5_6/1 state DOWN physical_state DISABLED
link mlx5_7/1 state DOWN physical_state DISABLED
link mlx5_8/1 state DOWN physical_state DISABLED
link mlx5_9/1 state DOWN physical_state DISABLED
sh-5.1#
```

12. Run the following **ib_write_bw** RDMA bandwidth test command:

```
sh-5.1# /root/perftest/ib_write_bw -R -T 41 -s 65536 -F -x 3 -m 4096 --report_gbits -q 16 -D
60  -d mlx5_2 -p 10000 --source_ip  192.168.4.226 --use_cuda=0 --use_cuda_dmabuf
192.168.4.225
```

where:

- The **mlx5_2** RDMA device is passed in the **-d** switch.
- The source IP address **192.168.4.226** and the destination IP address of the RDMA server **192.168.4.225**.

- The **--use_cuda=0**, **--use_cuda_dmabuf** switches indicate that the use of GPUDirect RDMA.

Example output

```
WARNING: BW peak won't be measured in this run.
Perftest doesn't supports CUDA tests with inline messages: inline size set to 0
Requested mtu is higher than active mtu
Changing to active mtu - 3
initializing CUDA
Listing all CUDA devices in system:
CUDA device 0: PCIe address is 61:00

Picking device No. 0
[pid = 8909, dev = 0] device name = [NVIDIA A40]
creating CUDA Ctx
making it the current CUDA Ctx
CUDA device integrated: 0
using DMA-BUF for GPU buffer address at 0x7f8738600000 aligned at 0x7f8738600000
with aligned size 2097152
allocated GPU buffer of a 2097152 address at 0x23a7420 for type CUDA_MEM_DEVICE
Calling ibv_reg_dmabuf_mr(offset=0, size=2097152, addr=0x7f8738600000, fd=40) for
QP #0
---------------------------------------------------------------------------------------
                    RDMA_Write BW Test
 Dual-port       : OFF  Device         : mlx5_2
 Number of qps   : 16  Transport type : IB
 Connection type : RC  Using SRQ      : OFF
 PCIe relax order: ON  Lock-free      : OFF
 ibv_wr* API     : ON  Using DDP      : OFF
 TX depth        : 128
 CQ Moderation   : 1
 CQE Poll Batch  : 16
 Mtu             : 1024[B]
 Link type       : Ethernet
 GID index       : 3
 Max inline data : 0[B]
 rdma_cm QPs  : ON
 Data ex. method : rdma_cm  TOS    : 41
---------------------------------------------------------------------------------------
 local address: LID 0000 QPN 0x012d PSN 0x3cb6d7
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
 local address: LID 0000 QPN 0x012e PSN 0x90e0ac
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
 local address: LID 0000 QPN 0x012f PSN 0x153f50
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
 local address: LID 0000 QPN 0x0130 PSN 0x5e0128
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
 local address: LID 0000 QPN 0x0131 PSN 0xd89752
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
 local address: LID 0000 QPN 0x0132 PSN 0xe5fc16
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
 local address: LID 0000 QPN 0x0133 PSN 0x236787
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
 local address: LID 0000 QPN 0x0134 PSN 0xd9273e
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
```

```
 local address: LID 0000 QPN 0x0135 PSN 0x37cfd4
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
 local address: LID 0000 QPN 0x0136 PSN 0x3bff8f
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
 local address: LID 0000 QPN 0x0137 PSN 0x81f2bd
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
 local address: LID 0000 QPN 0x0138 PSN 0x575c43
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
 local address: LID 0000 QPN 0x0139 PSN 0x6cf53d
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
 local address: LID 0000 QPN 0x013a PSN 0xcaaf6f
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
 local address: LID 0000 QPN 0x013b PSN 0x346437
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
 local address: LID 0000 QPN 0x013c PSN 0xcc5865
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
 remote address: LID 0000 QPN 0x026d PSN 0x359409
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
 remote address: LID 0000 QPN 0x026e PSN 0xe387bf
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
 remote address: LID 0000 QPN 0x026f PSN 0x5be79d
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
 remote address: LID 0000 QPN 0x0270 PSN 0x1b4b28
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
 remote address: LID 0000 QPN 0x0271 PSN 0x76a61b
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
 remote address: LID 0000 QPN 0x0272 PSN 0x3d50e1
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
 remote address: LID 0000 QPN 0x0273 PSN 0x1b572c
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
 remote address: LID 0000 QPN 0x0274 PSN 0x4ae1b5
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
 remote address: LID 0000 QPN 0x0275 PSN 0x5591b5
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
 remote address: LID 0000 QPN 0x0276 PSN 0xfa2593
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
 remote address: LID 0000 QPN 0x0277 PSN 0xd9473b
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
 remote address: LID 0000 QPN 0x0278 PSN 0x2116b2
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
 remote address: LID 0000 QPN 0x0279 PSN 0x9b83b6
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
 remote address: LID 0000 QPN 0x027a PSN 0xa0822b
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
 remote address: LID 0000 QPN 0x027b PSN 0x6d930d
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
 remote address: LID 0000 QPN 0x027c PSN 0xb1a4d
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
---------------------------------------------------------------------------------------
 #bytes     #iterations    BW peak[Gb/sec]    BW average[Gb/sec]   MsgRate[Mpps]
 65536      10329004         0.00               180.47         0.344228
---------------------------------------------------------------------------------------
deallocating GPU buffer 00007f8738600000
destroying current CUDA Ctx
sh-5.1#
```

A positive test is seeing an expected BW average and MsgRate in Mpps.

Upon completion of the **ib_write_bw** command, the server side output also appears on the server pod. See the following example:

Example output

```
WARNING: BW peak won't be measured in this run.
Perftest doesn't supports CUDA tests with inline messages: inline size set to 0

************************************
* Waiting for client to connect... *
************************************
Requested mtu is higher than active mtu
Changing to active mtu - 3
initializing CUDA
Listing all CUDA devices in system:
CUDA device 0: PCIe address is 61:00

Picking device No. 0
[pid = 9226, dev = 0] device name = [NVIDIA A40]
creating CUDA Ctx
making it the current CUDA Ctx
CUDA device integrated: 0
using DMA-BUF for GPU buffer address at 0x7f447a600000 aligned at 0x7f447a600000
with aligned size 2097152
allocated GPU buffer of a 2097152 address at 0x2406400 for type CUDA_MEM_DEVICE
Calling ibv_reg_dmabuf_mr(offset=0, size=2097152, addr=0x7f447a600000, fd=40) for
QP #0
---------------------------------------------------------------------------------------
                    RDMA_Write BW Test
 Dual-port       : OFF  Device         : mlx5_7
 Number of qps   : 16  Transport type : IB
 Connection type : RC  Using SRQ      : OFF
 PCIe relax order: ON  Lock-free      : OFF
 ibv_wr* API     : ON  Using DDP      : OFF
 CQ Moderation   : 1
 CQE Poll Batch  : 16
 Mtu             : 1024[B]
 Link type       : Ethernet
 GID index       : 3
 Max inline data : 0[B]
 rdma_cm QPs  : ON
 Data ex. method : rdma_cm  TOS    : 41
---------------------------------------------------------------------------------------
 Waiting for client rdma_cm QP to connect
 Please run the same command with the IB/RoCE interface IP
---------------------------------------------------------------------------------------
 local address: LID 0000 QPN 0x026d PSN 0x359409
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
 local address: LID 0000 QPN 0x026e PSN 0xe387bf
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
 local address: LID 0000 QPN 0x026f PSN 0x5be79d
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
 local address: LID 0000 QPN 0x0270 PSN 0x1b4b28
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
```

```
local address: LID 0000 QPN 0x0271 PSN 0x76a61b
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
local address: LID 0000 QPN 0x0272 PSN 0x3d50e1
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
local address: LID 0000 QPN 0x0273 PSN 0x1b572c
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
local address: LID 0000 QPN 0x0274 PSN 0x4ae1b5
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
local address: LID 0000 QPN 0x0275 PSN 0x5591b5
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
local address: LID 0000 QPN 0x0276 PSN 0xfa2593
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
local address: LID 0000 QPN 0x0277 PSN 0xd9473b
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
local address: LID 0000 QPN 0x0278 PSN 0x2116b2
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
local address: LID 0000 QPN 0x0279 PSN 0x9b83b6
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
local address: LID 0000 QPN 0x027a PSN 0xa0822b
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
local address: LID 0000 QPN 0x027b PSN 0x6d930d
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
local address: LID 0000 QPN 0x027c PSN 0xb1a4d
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:225
remote address: LID 0000 QPN 0x012d PSN 0x3cb6d7
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
remote address: LID 0000 QPN 0x012e PSN 0x90e0ac
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
remote address: LID 0000 QPN 0x012f PSN 0x153f50
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
remote address: LID 0000 QPN 0x0130 PSN 0x5e0128
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
remote address: LID 0000 QPN 0x0131 PSN 0xd89752
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
remote address: LID 0000 QPN 0x0132 PSN 0xe5fc16
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
remote address: LID 0000 QPN 0x0133 PSN 0x236787
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
remote address: LID 0000 QPN 0x0134 PSN 0xd9273e
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
remote address: LID 0000 QPN 0x0135 PSN 0x37cfd4
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
remote address: LID 0000 QPN 0x0136 PSN 0x3bff8f
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
remote address: LID 0000 QPN 0x0137 PSN 0x81f2bd
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
remote address: LID 0000 QPN 0x0138 PSN 0x575c43
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
remote address: LID 0000 QPN 0x0139 PSN 0x6cf53d
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
remote address: LID 0000 QPN 0x013a PSN 0xcaaf6f
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
remote address: LID 0000 QPN 0x013b PSN 0x346437
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
remote address: LID 0000 QPN 0x013c PSN 0xcc5865
GID: 00:00:00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
```

```
---------------------------------------------------------------------------------------
 #bytes     #iterations    BW peak[Gb/sec]    BW average[Gb/sec]   MsgRate[Mpps]
 65536      10329004         0.00               180.47         0.344228
---------------------------------------------------------------------------------------
deallocating GPU buffer 00007f447a600000
destroying current CUDA Ctx
```

CHAPTER 6. DYNAMIC ACCELERATOR SLICER (DAS) OPERATOR

IMPORTANT

Dynamic Accelerator Slicer Operator is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

The Dynamic Accelerator Slicer (DAS) Operator allows you to dynamically slice GPU accelerators in OpenShift Container Platform, instead of relying on statically sliced GPUs defined when the node is booted. This allows you to dynamically slice GPUs based on specific workload demands, ensuring efficient resource utilization.

Dynamic slicing is useful if you do not know all the accelerator partitions needed in advance on every node on the cluster.

The DAS Operator currently includes a reference implementation for NVIDIA Multi-Instance GPU (MIG) and is designed to support additional technologies such as NVIDIA MPS or GPUs from other vendors in the future.

Limitations

The following limitations apply when using the Dynamic Accelerator Slicer Operator:

- You need to identify potential incompatibilities and ensure the system works seamlessly with various GPU drivers and operating systems.
- The Operator only works with specific MIG compatible NVIDIA GPUs and drivers, such as H100 and A100.
- The Operator cannot use only a subset of the GPUs of a node.
- The NVIDIA device plugin cannot be used together with the Dynamic Accelerator Slicer Operator to manage the GPU resources of a cluster.

NOTE

The DAS Operator is designed to work with MIG-enabled GPUs. It allocates MIG slices instead of whole GPUs. Installing the DAS Operator prevents the use of the standard resource request through the NVIDIA device plugin such as **nvidia.com/gpu: "1"**, for allocating the entire GPU.

6.1. INSTALLING THE DYNAMIC ACCELERATOR SLICER OPERATOR

As a cluster administrator, you can install the Dynamic Accelerator Slicer (DAS) Operator by using the OpenShift Container Platform web console or the OpenShift CLI.

6.1.1. Installing the Dynamic Accelerator Slicer Operator using the web console

As a cluster administrator, you can install the Dynamic Accelerator Slicer (DAS) Operator using the OpenShift Container Platform web console.

Prerequisites

- You have access to an OpenShift Container Platform cluster using an account with **cluster-admin** permissions.
- You have installed the required prerequisites:
  - cert-manager Operator for Red Hat OpenShift
  - Node Feature Discovery (NFD) Operator
  - NVIDIA GPU Operator
  - NodeFeatureDiscovery CR

Procedure

1. Configure the NVIDIA GPU Operator for MIG support:
   a. In the OpenShift Container Platform web console, navigate to **Operators → Installed Operators**.
   b. Select the **NVIDIA GPU Operator** from the list of installed operators.
   c. Click the **ClusterPolicy** tab and then click **Create ClusterPolicy**.
   d. In the YAML editor, replace the default content with the following cluster policy configuration to disable the default NVIDIA device plugin and enable MIG support:

```
apiVersion: nvidia.com/v1
kind: ClusterPolicy
metadata:
  name: gpu-cluster-policy
spec:
  daemonsets:
    rollingUpdate:
      maxUnavailable: "1"
    updateStrategy: RollingUpdate
  dcgm:
    enabled: true
  dcgmExporter:
    config:
      name: ""
    enabled: true
    serviceMonitor:
      enabled: true
  devicePlugin:
    config:
      default: ""
      name: ""
    enabled: false
    mps:
```

```
    root: /run/nvidia/mps
driver:
  certConfig:
    name: ""
  enabled: true
  kernelModuleConfig:
    name: ""
  licensingConfig:
    configMapName: ""
    nlsEnabled: true
  repoConfig:
    configMapName: ""
  upgradePolicy:
    autoUpgrade: true
    drain:
      deleteEmptyDir: false
      enable: false
      force: false
      timeoutSeconds: 300
    maxParallelUpgrades: 1
    maxUnavailable: 25%
    podDeletion:
      deleteEmptyDir: false
      force: false
      timeoutSeconds: 300
    waitForCompletion:
      timeoutSeconds: 0
  useNvidiaDriverCRD: false
  useOpenKernelModules: false
  virtualTopology:
    config: ""
gdrcopy:
  enabled: false
gds:
  enabled: false
gfd:
  enabled: true
mig:
  strategy: mixed
migManager:
  config:
    default: ""
    name: default-mig-parted-config
  enabled: true
  env:
    - name: WITH_REBOOT
      value: 'true'
    - name: MIG_PARTED_MODE_CHANGE_ONLY
      value: 'true'
nodeStatusExporter:
  enabled: true
operator:
  defaultRuntime: crio
  initContainer: {}
  runtimeClass: nvidia
  use_ocp_driver_toolkit: true
```

```
sandboxDevicePlugin:
  enabled: true
sandboxWorkloads:
  defaultWorkload: container
  enabled: false
toolkit:
  enabled: true
  installDir: /usr/local/nvidia
validator:
  plugin:
    env:
    - name: WITH_WORKLOAD
      value: "false"
  cuda:
    env:
    - name: WITH_WORKLOAD
      value: "false"
vfioManager:
  enabled: true
vgpuDeviceManager:
  enabled: true
vgpuManager:
  enabled: false
```

e. Click **Create** to apply the cluster policy.

f. Navigate to **Workloads → Pods** and select the **nvidia-gpu-operator** namespace to monitor the cluster policy deployment.

g. Wait for the NVIDIA GPU Operator cluster policy to reach the **Ready** state. You can monitor this by:

   i. Navigating to **Operators → Installed Operators → NVIDIA GPU Operator**.

   ii. Clicking the **ClusterPolicy** tab and checking that the status shows **ready**.

h. Verify that all pods in the NVIDIA GPU Operator namespace are running by selecting the **nvidia-gpu-operator** namespace and navigating to **Workloads → Pods**.

i. Label nodes with MIG-capable GPUs to enable MIG mode:

   i. Navigate to **Compute → Nodes**.

   ii. Select a node that has MIG-capable GPUs.

   iii. Click **Actions → Edit Labels**.

   iv. Add the label **nvidia.com/mig.config=all-enabled**.

   v. Click **Save**.

   vi. Repeat for each node with MIG-capable GPUs.

IMPORTANT

After applying the MIG label, the labeled nodes will reboot to enable MIG mode. Wait for the nodes to come back online before proceeding.

j. Verify that MIG mode is successfully enabled on the GPU nodes by checking that the **nvidia.com/mig.config=all-enabled** label appears in the **Labels** section. To locate the label, navigate to **Compute → Nodes**, select the GPU node, and click the **Details** tab.

2. In the OpenShift Container Platform web console, click **Operators → OperatorHub**.
3. Search for **Dynamic Accelerator Slicer** or **DAS** in the filter box to locate the DAS Operator.
4. Select the **Dynamic Accelerator Slicer** and click **Install**.
5. On the **Install Operator** page:
   a. Select **All namespaces on the cluster (default)**for the installation mode.
   b. Select **Installed Namespace → Operator recommended Namespace: Project das-operator**.
   c. If creating a new namespace, enter **das-operator** as the namespace name.
   d. Select an update channel.
   e. Select **Automatic** or **Manual** for the approval strategy.
6. Click **Install**.
7. In the OpenShift Container Platform web console, click **Operators → Installed Operators**.
8. Select **DAS Operator** from the list.
9. In the **Provided APIs** table column, click **DASOperator**. This takes you to the **DASOperator** tab of the **Operator details** page.
10. Click **Create DASOperator**. This takes you to the **Create DASOperator** YAML view.
11. In the YAML editor, paste the following example:

Example DASOperator CR

```
apiVersion: inference.redhat.com/v1alpha1
kind: DASOperator
metadata:
  name: cluster (1)
  namespace: das-operator
spec:
  logLevel: Normal
  operatorLogLevel: Normal
  managementState: Managed
```

(1) The name of the **DASOperator** CR must be **cluster**.

12. Click **Create**.

Verification

To verify that the DAS Operator installed successfully:

1. Navigate to the **Operators → Installed Operators** page.
2. Ensure that **Dynamic Accelerator Slicer** is listed in the **das-operator** namespace with a **Status** of **Succeeded**.

To verify that the **DASOperator** CR installed successfully:

- After you create the **DASOperator** CR, the web console brings you to the **DASOperator list view**. The **Status** field of the CR changes to **Available** when all of the components are running.
- Optional. You can verify that the **DASOperator** CR installed successfully by running the following command in the OpenShift CLI:

```
$ oc get dasoperator -n das-operator
```

Example output

```
NAME      STATUS     AGE
cluster   Available 3m
```

NOTE

During installation an Operator might display a **Failed** status. If the installation later succeeds with an **Succeeded** message, you can ignore the **Failed** message.

You can also verify the installation by checking the pods:

1. Navigate to the **Workloads → Pods** page and select the **das-operator** namespace.
2. Verify that all DAS Operator component pods are running:
   - **das-operator** pods (main operator controllers)
   - **das-operator-webhook** pods (webhook servers)
   - **das-scheduler** pods (scheduler plugins)
   - **das-daemonset** pods (only on nodes with MIG-compatible GPUs)

NOTE

The **das-daemonset** pods will only appear on nodes that have MIG-compatible GPU hardware. If you do not see any daemonset pods, verify that your cluster has nodes with supported GPU hardware and that the NVIDIA GPU Operator is properly configured.

Troubleshooting

Use the following procedure if the Operator does not appear to be installed:

1. Navigate to the **Operators → Installed Operators** page and inspect the **Operator Subscriptions** and **Install Plans** tabs for any failure or errors under **Status**.
2. Navigate to the **Workloads → Pods** page and check the logs for pods in the **das-operator** namespace.

Additional resources

- cert-manager Operator for Red Hat OpenShift
- Node Feature Discovery (NFD) Operator
- NVIDIA GPU Operator
- NodeFeatureDiscovery CR

6.1.2. Installing the Dynamic Accelerator Slicer Operator using the CLI

As a cluster administrator, you can install the Dynamic Accelerator Slicer (DAS) Operator using the OpenShift CLI.

Prerequisites

- You have access to an OpenShift Container Platform cluster using an account with **cluster-admin** permissions.
- You have installed the OpenShift CLI (**oc**).
- You have installed the required prerequisites:
  - cert-manager Operator for Red Hat OpenShift
  - Node Feature Discovery (NFD) Operator
  - NVIDIA GPU Operator
  - NodeFeatureDiscovery CR

Procedure

1. Configure the NVIDIA GPU Operator for MIG support:
   a. Apply the following cluster policy to disable the default NVIDIA device plugin and enable MIG support. Create a file named **gpu-cluster-policy.yaml** with the following content:

```
apiVersion: nvidia.com/v1
kind: ClusterPolicy
metadata:
  name: gpu-cluster-policy
spec:
  daemonsets:
    rollingUpdate:
      maxUnavailable: "1"
    updateStrategy: RollingUpdate
  dcgm:
    enabled: true
```

```
dcgmExporter:
  config:
    name: ""
  enabled: true
  serviceMonitor:
    enabled: true
devicePlugin:
  config:
    default: ""
    name: ""
  enabled: false
  mps:
    root: /run/nvidia/mps
driver:
  certConfig:
    name: ""
  enabled: true
  kernelModuleConfig:
    name: ""
  licensingConfig:
    configMapName: ""
    nlsEnabled: true
  repoConfig:
    configMapName: ""
  upgradePolicy:
    autoUpgrade: true
    drain:
      deleteEmptyDir: false
      enable: false
      force: false
      timeoutSeconds: 300
    maxParallelUpgrades: 1
    maxUnavailable: 25%
    podDeletion:
      deleteEmptyDir: false
      force: false
      timeoutSeconds: 300
    waitForCompletion:
      timeoutSeconds: 0
  useNvidiaDriverCRD: false
  useOpenKernelModules: false
  virtualTopology:
    config: ""
gdrcopy:
  enabled: false
gds:
  enabled: false
gfd:
  enabled: true
mig:
  strategy: mixed
migManager:
  config:
    default: ""
    name: default-mig-parted-config
  enabled: true
```

```
    env:
      - name: WITH_REBOOT
        value: 'true'
      - name: MIG_PARTED_MODE_CHANGE_ONLY
        value: 'true'
  nodeStatusExporter:
    enabled: true
  operator:
    defaultRuntime: crio
    initContainer: {}
    runtimeClass: nvidia
    use_ocp_driver_toolkit: true
  sandboxDevicePlugin:
    enabled: true
  sandboxWorkloads:
    defaultWorkload: container
    enabled: false
  toolkit:
    enabled: true
    installDir: /usr/local/nvidia
  validator:
    plugin:
      env:
      - name: WITH_WORKLOAD
        value: "false"
    cuda:
      env:
      - name: WITH_WORKLOAD
        value: "false"
  vfioManager:
    enabled: true
  vgpuDeviceManager:
    enabled: true
  vgpuManager:
    enabled: false
```

b. Apply the cluster policy by running the following command:

```
$ oc apply -f gpu-cluster-policy.yaml
```

c. Verify the NVIDIA GPU Operator cluster policy reaches the **Ready** state by running the following command:

```
$ oc get clusterpolicies.nvidia.com gpu-cluster-policy -w
```

Wait until the **STATUS** column shows **ready**.

Example output

```
NAME                STATUS   AGE
gpu-cluster-policy  ready    2025-08-14T08:56:45Z
```

d. Verify that all pods in the NVIDIA GPU Operator namespace are running by running the following command:

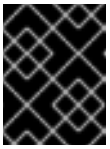
```
$ oc get pods -n nvidia-gpu-operator
```

All pods should show a **Running** or **Completed** status.

e. Label nodes with MIG-capable GPUs to enable MIG mode by running the following command:

```
$ oc label node $NODE_NAME nvidia.com/mig.config=all-enabled --overwrite
```

Replace **$NODE_NAME** with the name of each node that has MIG-capable GPUs.



IMPORTANT

After applying the MIG label, the labeled nodes reboot to enable MIG mode. Wait for the nodes to come back online before proceeding.

f. Verify that the nodes have successfully enabled MIG mode by running the following command:

```
$ oc get nodes -l nvidia.com/mig.config=all-enabled
```

2. Create a namespace for the DAS Operator:

a. Create the following **Namespace** custom resource (CR) that defines the **das-operator** namespace, and save the YAML in the **das-namespace.yaml** file:

```
apiVersion: v1
kind: Namespace
metadata:
  name: das-operator
  labels:
    name: das-operator
    openshift.io/cluster-monitoring: "true"
```

b. Create the namespace by running the following command:

```
$ oc create -f das-namespace.yaml
```

3. Install the DAS Operator in the namespace you created in the previous step by creating the following objects:

a. Create the following **OperatorGroup** CR and save the YAML in the **das-operatorgroup.yaml** file:

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  generateName: das-operator-
  name: das-operator
  namespace: das-operator
```

b. Create the **OperatorGroup** CR by running the following command:

```
$ oc create -f das-operatorgroup.yaml
```

c. Create the following **Subscription** CR and save the YAML in the **das-sub.yaml** file:

Example Subscription

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: das-operator
  namespace: das-operator
spec:
  channel: "stable"
  installPlanApproval: Automatic
  name: das-operator
  source: redhat-operators
  sourceNamespace: openshift-marketplace
```

d. Create the subscription object by running the following command:

```
$ oc create -f das-sub.yaml
```

e. Change to the **das-operator** project:

```
$ oc project das-operator
```

f. Create the following **DASOperator** CR and save the YAML in the **das-dasoperator.yaml** file:

Example DASOperator CR

```
apiVersion: inference.redhat.com/v1alpha1
kind: DASOperator
metadata:
  name: cluster (1)
  namespace: das-operator
spec:
  managementState: Managed
  logLevel: Normal
  operatorLogLevel: Normal
```

(1) The name of the **DASOperator** CR must be **cluster**.

g. Create the **dasoperator** CR by running the following command:

```
oc create -f das-dasoperator.yaml
```

Verification

- Verify that the Operator deployment is successful by running the following command:

```
$ oc get pods
```

Example output

```
NAME                                  READY   STATUS    RESTARTS   AGE
das-daemonset-6rsfd                   1/1     Running   0          5m16s
das-daemonset-8qzgf                   1/1     Running   0          5m16s
das-operator-5946478b47-cjfcp         1/1     Running   0          5m18s
das-operator-5946478b47-npwmn         1/1     Running   0          5m18s
das-operator-webhook-59949d4f85-5n9qt 1/1     Running   0          68s
das-operator-webhook-59949d4f85-nbtdl 1/1     Running   0          68s
das-scheduler-6cc59dbf96-4r85f        1/1     Running   0          68s
das-scheduler-6cc59dbf96-bf6ml        1/1     Running   0          68s
```

A successful deployment shows all pods with a **Running** status. The deployment includes:

das-operator
: Main Operator controller pods

das-operator-webhook
: Webhook server pods for mutating pod requests

das-scheduler
: Scheduler plugin pods for MIG slice allocation

das-daemonset
: Daemonset pods that run only on nodes with MIG-compatible GPUs

NOTE

The **das-daemonset** pods only appear on nodes that have MIG-compatible GPU hardware. If you do not see any daemonset pods, verify that your cluster has nodes with supported GPU hardware and that the NVIDIA GPU Operator is properly configured.

Additional resources

- cert-manager Operator for Red Hat OpenShift
- Node Feature Discovery (NFD) Operator
- NVIDIA GPU Operator
- NodeFeatureDiscovery CR

6.2. UNINSTALLING THE DYNAMIC ACCELERATOR SLICER OPERATOR

Use one of the following procedures to uninstall the Dynamic Accelerator Slicer (DAS) Operator, depending on how the Operator was installed.

6.2.1. Uninstalling the Dynamic Accelerator Slicer Operator using the web console

You can uninstall the Dynamic Accelerator Slicer (DAS) Operator using the OpenShift Container Platform web console.

Prerequisites

- You have access to an OpenShift Container Platform cluster using an account with **cluster-admin** permissions.
- The DAS Operator is installed in your cluster.

Procedure

1. In the OpenShift Container Platform web console, navigate to **Operators → Installed Operators**.
2. Locate the **Dynamic Accelerator Slicer** in the list of installed Operators.
3. Click the **Options** menu  for the DAS Operator and select **Uninstall Operator**.
4. In the confirmation dialog, click **Uninstall** to confirm the removal.
5. Navigate to **Home → Projects**.
6. Search for **das-operator** in the search box to locate the DAS Operator project.
7. Click the **Options** menu  next to the das-operator project, and select **Delete Project**.
8. In the confirmation dialog, type **das-operator** in the dialog box, and click **Delete** to confirm the deletion.

Verification

1. Navigate to the **Operators → Installed Operators** page.
2. Verify that the Dynamic Accelerator Slicer (DAS) Operator is no longer listed.
3. Optional. Verify that the **das-operator** namespace and its resources have been removed by running the following command:

   ```
   $ oc get namespace das-operator
   ```

   The command should return an error indicating that the namespace is not found.

WARNING

Uninstalling the DAS Operator removes all GPU slice allocations and might cause running workloads that depend on GPU slices to fail. Ensure that no critical workloads are using GPU slices before proceeding with the uninstallation.

6.2.2. Uninstalling the Dynamic Accelerator Slicer Operator using the CLI

You can uninstall the Dynamic Accelerator Slicer (DAS) Operator using the OpenShift CLI.

Prerequisites

- You have access to an OpenShift Container Platform cluster using an account with **cluster-admin** permissions.
- You have installed the OpenShift CLI (**oc**).
- The DAS Operator is installed in your cluster.

Procedure

1. List the installed operators to find the DAS Operator subscription by running the following command:

```
$ oc get subscriptions -n das-operator
```

Example output

```
NAME           PACKAGE        SOURCE             CHANNEL
das-operator   das-operator   redhat-operators   stable
```

2. Delete the subscription by running the following command:

```
$ oc delete subscription das-operator -n das-operator
```

3. List and delete the cluster service version (CSV) by running the following commands:

```
$ oc get csv -n das-operator
```

```
$ oc delete csv <csv-name> -n das-operator
```

4. Remove the operator group by running the following command:

```
$ oc delete operatorgroup das-operator -n das-operator
```

5. Delete any remaining **AllocationClaim** resources by running the following command:

```
$ oc delete allocationclaims --all -n das-operator
```

6. Remove the DAS Operator namespace by running the following command:

```
$ oc delete namespace das-operator
```

Verification

1. Verify that the DAS Operator resources have been removed by running the following command:

```
$ oc get namespace das-operator
```

   The command should return an error indicating that the namespace is not found.

2. Verify that no **AllocationClaim** custom resource definitions remain by running the following command:

```
$ oc get crd | grep allocationclaim
```

   The command should return an error indicating that no custom resource definitions are found.

WARNING

Uninstalling the DAS Operator removes all GPU slice allocations and might cause running workloads that depend on GPU slices to fail. Ensure that no critical workloads are using GPU slices before proceeding with the uninstallation.

6.3. DEPLOYING GPU WORKLOADS WITH THE DYNAMIC ACCELERATOR SLICER OPERATOR

You can deploy workloads that request GPU slices managed by the Dynamic Accelerator Slicer (DAS) Operator. The Operator dynamically partitions GPU accelerators and schedules workloads to available GPU slices.

Prerequisites

- You have MIG supported GPU hardware available in your cluster.
- The NVIDIA GPU Operator is installed and the **ClusterPolicy** shows a **Ready** state.
- You have installed the DAS Operator.

Procedure

1. Create a namespace by running the following command:

```
oc new-project cuda-workloads
```

2. Create a deployment that requests GPU resources using the NVIDIA MIG resource:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cuda-vectoradd
spec:
  replicas: 2
  selector:
    matchLabels:
      app: cuda-vectoradd
  template:
    metadata:
      labels:
        app: cuda-vectoradd
    spec:
      restartPolicy: Always
      containers:
      - name: cuda-vectoradd
        image: nvcr.io/nvidia/k8s/cuda-sample:vectoradd-cuda12.5.0-ubi8
        resources:
          limits:
            nvidia.com/mig-1g.5gb: "1"
        command:
          - sh
          - -c
          - |
            env && /cuda-samples/vectorAdd && sleep 3600
```

3. Apply the deployment configuration by running the following command:

```
$ oc apply -f cuda-vectoradd-deployment.yaml
```

4. Verify that the deployment is created and pods are scheduled by running the following command:

```
$ oc get deployment cuda-vectoradd
```

Example output

```
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
cuda-vectoradd   2/2     2          2          2m
```

5. Check the status of the pods by running the following command:

```
$ oc get pods -l app=cuda-vectoradd
```

Example output

```
NAME                          READY   STATUS    RESTARTS   AGE
cuda-vectoradd-6b8c7d4f9b-abc12   1/1     Running   0        2m
cuda-vectoradd-6b8c7d4f9b-def34   1/1     Running   0        2m
```

Verification

1. Check that **AllocationClaim** resources were created for your deployment pods by running the following command:

```
$ oc get allocationclaims -n das-operator
```

Example output

```
NAME                                                                                                AGE
13950288-57df-4ab5-82bc-6138f646633e-harpatil000034jma-qh5fm-worker-f-57md9-cuda-vectoradd-0   2m
ce997b60-a0b8-4ea4-9107-cf59b425d049-harpatil000034jma-qh5fm-worker-f-fl4wg-cuda-vectoradd-0   2m
```

2. Verify that the GPU slices are properly allocated by checking one of the pod's resource allocation by running the following command:

```
$ oc describe pod -l app=cuda-vectoradd
```

3. Check the logs to verify the CUDA sample application runs successfully by running the following command:

```
$ oc logs -l app=cuda-vectoradd
```

Example output

```
[Vector addition of 50000 elements]
Copy input data from the host memory to the CUDA device
CUDA kernel launch with 196 blocks of 256 threads
Copy output data from the CUDA device to the host memory
Test PASSED
```

4. Check the environment variables to verify that the GPU devices are properly exposed to the container by running the following command:

```
$ oc exec deployment/cuda-vectoradd -- env | grep -E "
(NVIDIA_VISIBLE_DEVICES|CUDA_VISIBLE_DEVICES)"
```

Example output

```
NVIDIA_VISIBLE_DEVICES=MIG-d8ac9850-d92d-5474-b238-0afeabac1652
CUDA_VISIBLE_DEVICES=MIG-d8ac9850-d92d-5474-b238-0afeabac1652
```

These environment variables indicate that the GPU MIG slice has been properly allocated and is visible to the CUDA runtime within the container.

6.4. TROUBLESHOOTING THE DYNAMIC ACCELERATOR SLICER OPERATOR

If you experience issues with the Dynamic Accelerator Slicer (DAS) Operator, use the following troubleshooting steps to diagnose and resolve problems.

Prerequisites

- You have installed the DAS Operator.
- You have access to the OpenShift Container Platform cluster as a user with the cluster-admin role.

6.4.1. Debugging DAS Operator components

Procedure

1. Check the status of all DAS Operator components by running the following command:

```
$ oc get pods -n das-operator
```

Example output

```
NAME                                  READY   STATUS    RESTARTS   AGE
das-daemonset-6rsfd                   1/1     Running   0          5m16s
das-daemonset-8qzgf                   1/1     Running   0          5m16s
das-operator-5946478b47-cjfcp         1/1     Running   0          5m18s
das-operator-5946478b47-npwmn         1/1     Running   0          5m18s
das-operator-webhook-59949d4f85-5n9qt 1/1     Running   0          68s
das-operator-webhook-59949d4f85-nbtdl 1/1     Running   0          68s
das-scheduler-6cc59dbf96-4r85f        1/1     Running   0          68s
das-scheduler-6cc59dbf96-bf6ml        1/1     Running   0          68s
```

2. Inspect the logs of the DAS Operator controller by running the following command:

```
$ oc logs -n das-operator deployment/das-operator
```

3. Check the logs of the webhook server by running the following command:

```
$ oc logs -n das-operator deployment/das-operator-webhook
```

4. Check the logs of the scheduler plugin by running the following command:

```
$ oc logs -n das-operator deployment/das-scheduler
```

5. Check the logs of the device plugin daemonset by running the following command:

```
$ oc logs -n das-operator daemonset/das-daemonset
```

6.4.2. Monitoring AllocationClaims

Procedure

1. Inspect active **AllocationClaim** resources by running the following command:

```
$ oc get allocationclaims -n das-operator
```

Example output

```
NAME                                                                AGE
13950288-57df-4ab5-82bc-6138f646633e-harpatil000034jma-qh5fm-worker-f-57md9-cuda-vectoradd-0   5m
ce997b60-a0b8-4ea4-9107-cf59b425d049-harpatil000034jma-qh5fm-worker-f-fl4wg-cuda-vectoradd-0   5m
```

2. View detailed information about a specific **AllocationClaim** by running the following command:

```
$ oc get allocationclaims -n das-operator -o yaml
```

Example output (truncated)

```
apiVersion: inference.redhat.com/v1alpha1
kind: AllocationClaim
metadata:
  name: 13950288-57df-4ab5-82bc-6138f646633e-harpatil000034jma-qh5fm-worker-f-57md9-cuda-vectoradd-0
  namespace: das-operator
spec:
  gpuUUID: GPU-9003fd9c-1ad1-c935-d8cd-d1ae69ef17c0
  migPlacement:
    size: 1
    start: 0
  nodename: harpatil000034jma-qh5fm-worker-f-57md9
  podRef:
    kind: Pod
    name: cuda-vectoradd-f4b84b678-l2m69
    namespace: default
    uid: 13950288-57df-4ab5-82bc-6138f646633e
  profile: 1g.5gb
status:
  conditions:
  - lastTransitionTime: "2025-08-06T19:28:48Z"
    message: Allocation is inUse
    reason: inUse
    status: "True"
    type: State
  state: inUse
```

3. Check for claims in different states by running the following command:

```
$ oc get allocationclaims -n das-operator -o jsonpath='{range .items[*]}{.metadata.name}{"\t"}{.status.state}{"\n"}{end}'
```

Example output

```
13950288-57df-4ab5-82bc-6138f646633e-harpatil000034jma-qh5fm-worker-f-57md9-cuda-vectoradd-0 inUse
ce997b60-a0b8-4ea4-9107-cf59b425d049-harpatil000034jma-qh5fm-worker-f-fl4wg-cuda-vectoradd-0 inUse
```

4. View events related to **AllocationClaim** resources by running the following command:

```
$ oc get events -n das-operator --field-selector involvedObject.kind=AllocationClaim
```

5. Check **NodeAccelerator** resources to verify GPU hardware detection by running the following command:

```
$ oc get nodeaccelerator -n das-operator
```

Example output

```
NAME                                  AGE
harpatil000034jma-qh5fm-worker-f-57md9   96m
harpatil000034jma-qh5fm-worker-f-fl4wg   96m
```

The **NodeAccelerator** resources represent the GPU-capable nodes detected by the DAS Operator.

Additional information

The **AllocationClaim** custom resource tracks the following information:

GPU UUID
: The unique identifier of the GPU device.

Slice position
: The position of the MIG slice on the GPU.

Pod reference
: The pod that requested the GPU slice.

State
: The current state of the claim (**staged**, **created**, or **released**).

Claims start in the **staged** state and transition to **created** when all requests are satisfied. When a pod is deleted, the associated claim is automatically cleaned up.

6.4.3. Verifying GPU device availability

Procedure

1. On a node with GPU hardware, verify that CDI devices were created by running the following command:

```
$ oc debug node/<node-name>
```

```
sh-4.4# chroot /host
sh-4.4# ls -l /var/run/cdi/
```

2. Check the NVIDIA GPU Operator status by running the following command:

```
$ oc get clusterpolicies.nvidia.com -o jsonpath='{.items[0].status.state}'
```

The output should show **ready**.

6.4.4. Increasing log verbosity

Procedure

To get more detailed debugging information:

1. Edit the **DASOperator** resource to increase log verbosity by running the following command:

```
$ oc edit dasoperator -n das-operator
```

2. Set the **operatorLogLevel** field to **Debug** or **Trace**:

```
spec:
  operatorLogLevel: Debug
```

3. Save the changes and verify that the operator pods restart with increased verbosity.

6.4.5. Common issues and solutions

PODS STUCK IN UNEXPECTEDADMISSIONERROR STATE

Due to kubernetes/kubernetes#128043, pods might enter an **UnexpectedAdmissionError** state if admission fails. Pods managed by higher level controllers such as Deployments are recreated automatically. Naked pods, however, must be cleaned up manually with **oc delete pod**. Using controllers is recommended until the upstream issue is resolved.

Prerequisites not met

If the DAS Operator fails to start or function properly, verify that all prerequisites are installed:

- Cert-manager
- Node Feature Discovery (NFD) Operator
- NVIDIA GPU Operator

Additional resources

- Kubernetes issue #128043
- Node Feature Discovery Operator
- NVIDIA GPU Operator troubleshooting