OpenShift Container Platform 4.18

Hardware accelerators

Hardware accelerators

Last Updated: 2025-06-09

OpenShift Container Platform 4.18 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.

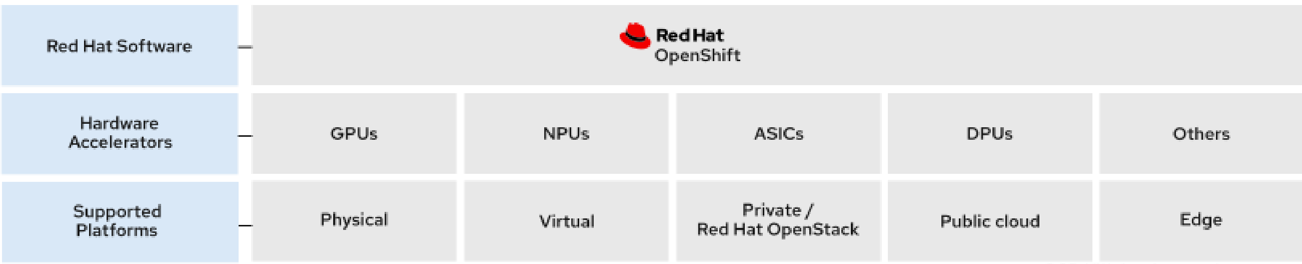
Table of Contents

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 .

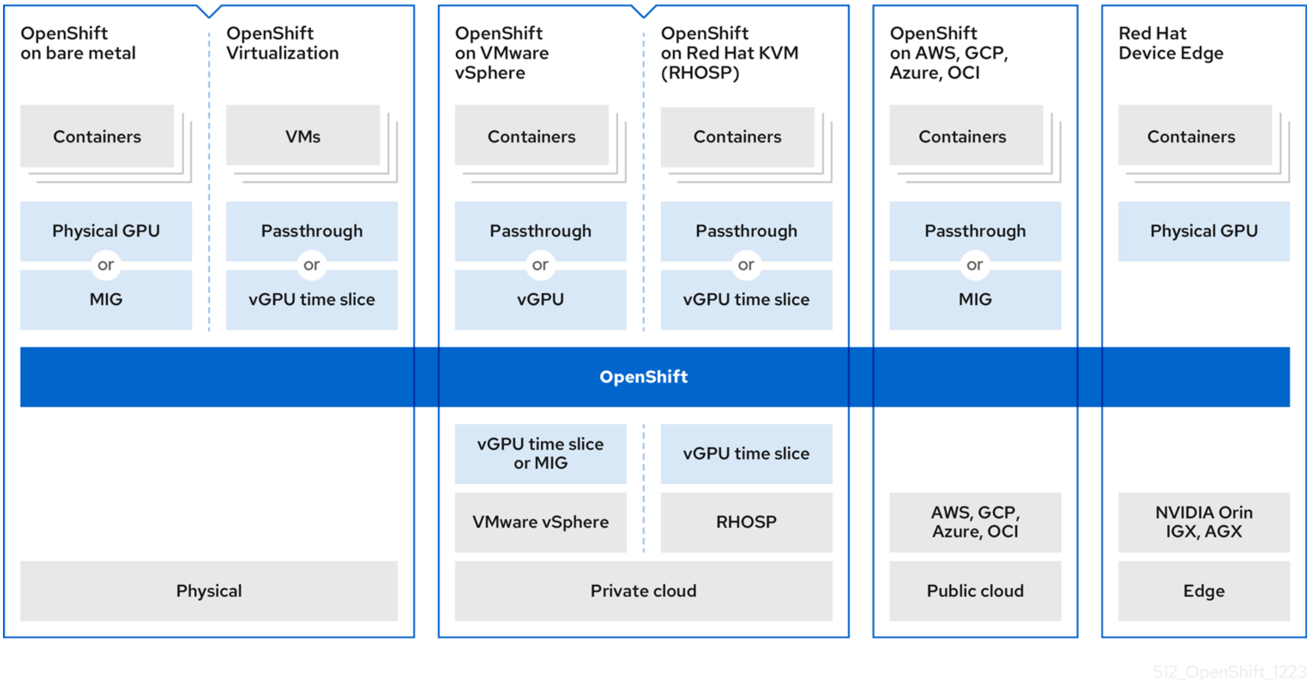
2.1. NVIDIA GPU PREREQUISITES

- 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 Platform (GCP), 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 GCP. For AWS, the OpenShift service is called ROSA (Red Hat OpenShift Service on AWS). For Azure, the service is called Azure Red Hat OpenShift. For GCP, the service is called OpenShift Dedicated on GCP.
- 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 the memory 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
```

```
U1VCU1lTVEVNPT0ibmV0IixBQ1RJT049PSJhZGQiLEFUVFJ7YWRkcmVzc309PSJiODozZjp
kMjozYjo1MToyOCIsQVRUUnt0eXBlfT09IjEiLE5BTUU9ImliczJmMCIKU1VCU1lTVEVNPT0ibm
V0IixBQ1RJT049PSJhZGQiLEFUVFJ7YWRkcmVzc309PSJiODozZjpkMjozYjo1MToyOSIsQV
RUUnt0eXBlfT09IjEiLE5BTUU9ImVuczhmMG5wMCIKU1VCU1lTVEVNPT0ibmV0IixBQ1RJT0
49PSJhZGQiLEFUVFJ7YWRkcmVzc309PSJiODozZjpkMjpmMDozNjpkMCIsQVRUUnt0eXBlfT
09IjEiLE5BTUU9ImliczJmMCIKU1VCU1lTVEVNPT0ibmV0IixBQ1RJT049PSJhZGQiLEFUVFJ
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:

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

Example output

```
machineconfig.machineconfiguration.openshift.io/99-machine-config-udev-network created
```

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 **deviceClasseWhiteList** 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
```