OpenShift Container Platform 4.19

硬件加速器

硬件加速器

Last Updated: 2025-12-09

OpenShift Container Platform 4.19 硬件加速器

硬件加速器

Legal Notice

Abstract

本文档提供了有关安装和配置 Red Hat OpenShift AI 支持的 GPU Operator 的说明，以提供硬件加速功能，以创建人工智能和机器学习(AI/ML)应用程序。

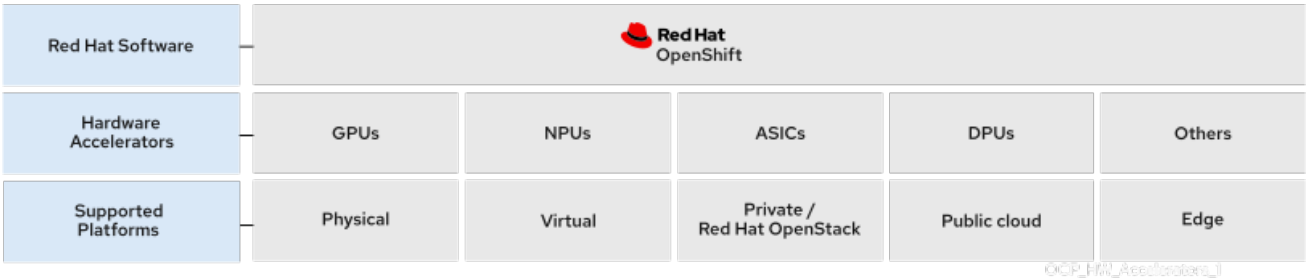
Table of Contents

第 1 章 关于硬件加速器

专用硬件加速器在新兴人工智能和机器学习(AI/ML)行业中发挥了关键作用。具体来说，硬件加速器对于培训以及支持这种新技术的大型语言和其他基础模型至关重要。数据科学家、数据工程师、ML 工程师和开发人员可以利用专门的硬件加速数据密集型转换和模型开发及服务。其中大多数生态系统都是开源的，有大量贡献合作伙伴和开源基础。

Red Hat OpenShift Container Platform 支持卡和外围硬件，添加组成硬件加速器的处理单元：

- 图形处理单元(GPU)
- Neural processing units (NPUs)
- 特定于应用程序的集成电路(ASIC)
- 数据处理单元(DPU)



专用硬件加速器为 AI/ML 开发提供了丰富的优点：

一个平台适用于所有功能

面向开发人员、数据工程师、数据科学家和 DevOps 的协作环境

使用 Operator 扩展功能

Operator 允许将 AI/ML 功能引入到 OpenShift Container Platform

混合云支持

对模型开发、交付和部署的内部支持

支持 AI/ML 工作负载

模型测试、迭代、集成、提升，并作为服务提供给生产环境中

红帽提供了一个优化的平台，可在 Linux (kernel 和 userspace) 和 Kubernetes 层的 Red Hat Enterprise Linux (RHEL)和 OpenShift Container Platform 平台中启用这些专用硬件加速器。为此，红帽在单个企业级 AI 应用平台中结合了 Red Hat OpenShift AI 和 Red Hat OpenShift Container Platform 的成熟功能。

硬件 Operator 使用 Kubernetes 集群的操作框架来启用所需的加速器资源。您还可以手动部署提供的设备插件或守护进程集。此插件在集群中注册 GPU。

某些专用硬件加速器设计为在必须维护安全环境以进行开发和测试的断开连接的环境中工作。

1.1. 硬件加速器

Red Hat OpenShift Container Platform 启用以下硬件加速器：

- NVIDIA GPU
- AMD Instinct® GPU

- Intel® Gaudi®

其他资源

- Red Hat OpenShift AI 简介
- NVIDIA GPU Operator on Red Hat OpenShift Container Platform
- AMD Instinct 加速器
- Intel Gaudi AI Accelerators

第 2 章 NVIDIA GPU 架构

NVIDIA 支持在 OpenShift Container Platform 上使用图形处理单元 (GPU) 资源。OpenShift Container Platform 是一个以安全为中心的、强化的 Kubernetes 平台，由红帽开发并提供支持，用于大规模部署和管理 Kubernetes 集群。OpenShift Container Platform 包括对 Kubernetes 的增强，以便用户可以轻松地配置和使用 NVIDIA GPU 资源来加快工作负载。

NVIDIA GPU Operator 利用 OpenShift Container Platform 中的 Operator 框架来管理运行 GPU 加速工作负载所需的 NVIDIA 软件组件的完整生命周期。

这些组件包括 NVIDIA 驱动程序（为了启用 CUDA）、GPU 的 Kubernetes 设备插件、NVID Container Toolkit、使用 GPU 特性发现(GFD)、基于 DCGM 的监控等的自动节点标记。

注意

NVIDIA GPU Operator 的支持仅由 NVIDIA 提供。有关从 NVIDIA 获取支持的更多信息，请参阅 NVIDIA 支持。

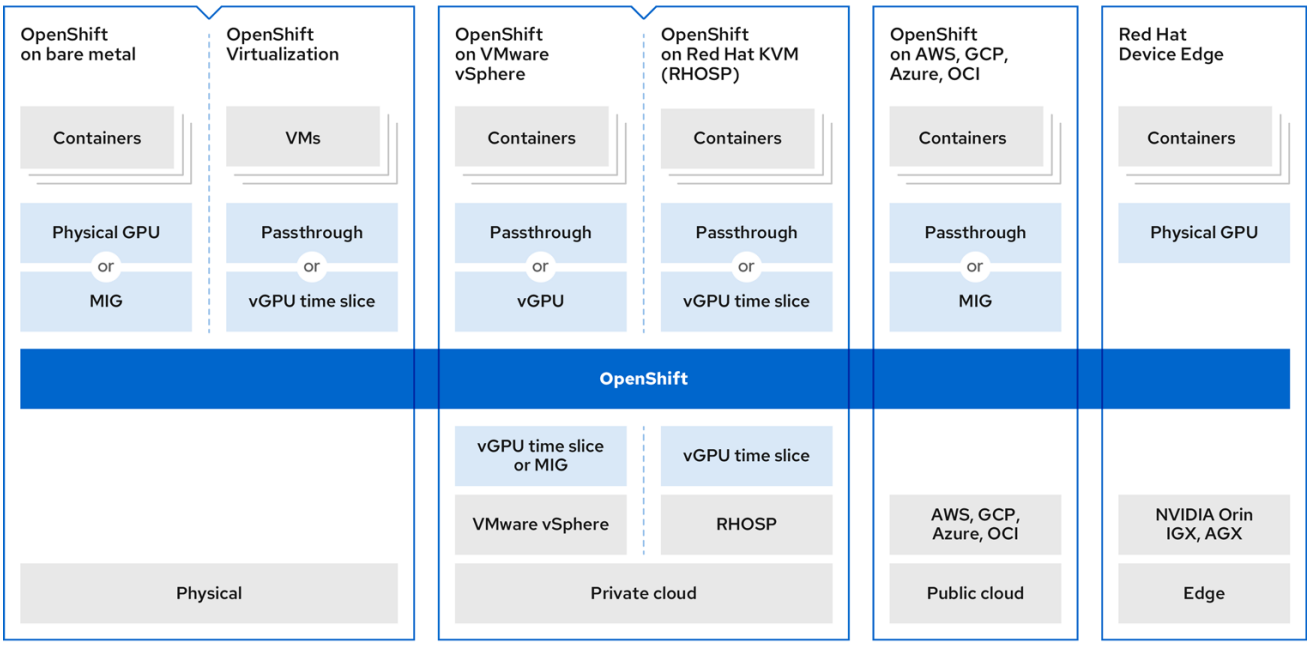
2.1. NVIDIA GPU 先决条件

- 包括至少一个 GPU worker 节点的，可正常工作的 OpenShift 集群。
- 以 **cluster-admin** 身份访问 OpenShift 集群，以执行必要的步骤。
- 已安装 OpenShift CLI (**oc**)。
- 已安装节点功能发现 (NFD) Operator 并创建了 **nodefeaturediscovery** 实例。

2.2. NVIDIA GPU 启用

下图显示了如何为 OpenShift 启用 GPU 架构：

图 2.1. NVIDIA GPU 启用



注意

从 NVIDIA Ampere 系列开始，在 GPU 上支持 MIG。有关支持 MIG 的 GPU 列表，请参阅 NVIDIA MIG 用户指南。

2.2.1. GPU 和裸机

您可以在 NVIDIA 认证的裸机服务器上部署 OpenShift Container Platform，但有一些限制：

- control plane 节点可以是 CPU 节点。
- Worker 节点必须是 GPU 节点，只要 AI/ML 工作负载在这些 worker 节点上执行。
  另外，worker 节点可以托管一个或多个 GPU，但它们必须是相同的类型。例如，一个节点可以有两个 NVIDIA A100 GPU，但不支持在一个节点中带有一个 A100 GPU 和一个 T4 GPU。
  Kubernetes 的 NVIDIA 设备插件不支持在同一节点上混合不同的 GPU 模型。
- 在使用 OpenShift 时，请注意，需要一个、三个或更多个服务器。不支持带有两个服务器的集群。单一服务器部署称为单一节点 openShift (SNO)，使用此配置的 OpenShift 环境不具有高可用性。

您可以选择以下方法之一来访问容器化 GPU：

- GPU passthrough（GPU 透传）
- 多实例 GPU (MIG)

其他资源

- Red Hat OpenShift on Bare Metal Stack

2.2.2. GPU 和虚拟化

虽然许多开发人员和企业都在转型到容器化应用程序和无服务器基础架构，但仍然有大量对在虚拟机(VM)上运行的应用程序进行开发和维护的需求。Red Hat OpenShift Virtualization 提供此功能，使企业能够将虚拟机合并到集群中的容器化工作流中。

您可以选择以下方法之一将 worker 节点连接到 GPU：

- 用于访问和使用虚拟机(VM)中的 GPU 硬件的 GPU 透传。
- 当 GPU 计算的容量没有因为工作负载而饱和时，可以进行 GPU (vGPU) 时间分片。

其他资源

- 带有 OpenShift Virtualization 的 NVIDIA GPU Operator

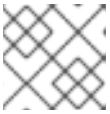
2.2.3. GPU 和 vSphere

您可以在可托管不同 GPU 类型的 NVIDIA 认证的 VMware vSphere 服务器上部署 OpenShift Container Platform。

如果虚拟机使用了 vGPU 实例，必须在 hypervisor 中安装 NVIDIA GPU 驱动程序。对于 VMware vSphere，此主机驱动程序以 VIB 文件的形式提供。

可分配给 worker 节点虚拟机的最大 vGPU 数量取决于 vSphere 的版本：

- vSphere 7.0：每个虚拟机最多 4 个 vGPU
- vSphere 8.0：每个虚拟机最大 8 个 vGPU



注意

vSphere 8.0 引入了对与一个虚拟机关联的多个完整或部分同配置集的支持。

您可以选择以下方法之一将 worker 节点附加到 GPU：

- 用于访问和使用虚拟机(VM)中的 GPU 硬件的 GPU 透传
- 当不需要所有 GPU 时，可以使用 GPU (vGPU) 时间分片

与裸机部署类似，需要一个或多个服务器。不支持带有两个服务器的集群。

其他资源

- 带有 NVIDIA vGPU 的 VMware vSphere 上的 OpenShift Container Platform

2.2.4. GPU 和 Red Hat KVM

您可以在基于 NVIDIA 认证的虚拟机 (KVM) 服务器上使用 OpenShift Container Platform。

与裸机部署类似，需要一个或多个服务器。不支持带有两个服务器的集群。

但是，与裸机部署不同，您可以在服务器中使用不同类型的 GPU。这是因为您可以将这些 GPU 分配给作为 Kubernetes 节点的不同虚拟机。唯一的限制是，一个 Kubernetes 节点在自己本身上必须具有相同的 GPU 类型。

您可以选择以下方法之一来访问容器化 GPU：

- 用于访问和使用虚拟机(VM)中的 GPU 硬件的 GPU 透传
- 当不需要所有 GPU 时，可以使用 GPU (vGPU) 时间分片

要启用 vGPU 功能，必须在主机级别安装特殊驱动程序。这个驱动程序作为 RPM 软件包提供。对于 GPU 透传分配，不需要这个主机驱动程序。

2.2.5. GPU 和 CSP

您可以将 OpenShift Container Platform 部署到主要的云服务供应商 (CSP) 之一：Amazon Web Services (AWS)、Google Cloud Platform 或 Microsoft Azure。

有两种操作模式：完全管理的部署和自我管理的部署。

- 在完全管理的部署中，一切都由红帽与 CSP 合作实现自动化。您可以通过 CSP Web 控制台请求 OpenShift 实例，集群由红帽自动创建并完全管理。您不必担心环境中节点故障或错误。红帽完全负责维护集群的正常运行时间。完全管理的服务在 AWS、Azure 和 Google Cloud 上提供。对于 AWS，OpenShift 服务名为 (Red Hat OpenShift Service on AWS)。对于 Azure，该服务称为 Azure Red Hat OpenShift。对于 Google Cloud，该服务在 Google Cloud 上称为 OpenShift Dedicated。
- 在自我管理的部署中，您需要自行实例化和维护 OpenShift 集群。红帽提供了 OpenShift-install 工具来支持 OpenShift 集群的部署。自我管理的服务可全局提供给所有 CSP。

重要的是，此计算实例是一个 GPU 加速的计算实例，并且 GPU 类型与 NVIDIA AI Enterprise 支持的 GPU 列表匹配。例如，T4、V100 和 A100 是此列表的一部分。

您可以选择以下方法之一来访问容器化 GPU：

- 用于访问和使用虚拟机(VM)中的 GPU 硬件的 GPU 透传。
- 当不需要整个 GPU 时，可以进行 GPU (vGPU) 时间分片。

其他资源

- 云中的 Red Hat Openshift

2.2.6. GPU 和 Red Hat Device Edge

Red Hat Device Edge 提供对 MicroShift 的访问。MicroShift 提供了单节点部署的简单性和资源约束（边缘）计算所需的功能和服务。Red Hat Device Edge 满足在资源受限环境中部署的裸机、虚拟、容器化或 Kubernetes 工作负载的需求。

您可以在 Red Hat Device Edge 环境中的容器上启用 NVIDIA GPU。

您可以使用 GPU 透传来访问容器化 GPU。

其他资源

- 如何在 Red Hat Device Edge 中使用 NVIDIA GPU 加速工作负载

2.3. GPU 共享方法

红帽和 NVIDIA 已开发了 GPU 并发和共享机制，以简化企业级 OpenShift Container Platform 集群上的 GPU 加速计算。

应用程序通常会有不同的计算要求，这可能会使 GPU 使用率不足。为每个工作负载提供正确的计算资源数量对于降低部署成本和最大化 GPU 使用率至关重要。

用于提高 GPU 使用率的并发机制，范围从编程模型 API 到系统软件和硬件分区，包括虚拟化。以下列表显示了 GPU 并发机制：

- 计算统一设备架构 (CUDA) 流
- Time-slicing（时间分片）
- CUDA 多进程服务 (MPS)
- 多实例 GPU (MIG)
- 使用 vGPU 的虚拟化

在将 GPU 并发机制用于不同的 OpenShift Container Platform 场景时，请考虑以下 GPU 共享建议：

裸机

vGPU 不可用。考虑使用支持 MIG 的卡。

虚拟机

vGPU 是最佳选择。

裸机中的较旧的 NVIDIA 卡中没有 MIG

考虑使用 time-slicing。

具有多个 GPU 的虚拟机，您需要透传和 vGPU

考虑使用单独的虚拟机。

使用 OpenShift Virtualization 和多个 GPU 的裸机

对于托管的虚拟机，考虑使用透传，对呀容器，考虑使用时间分片。

其他资源

- 提高 GPU 利用率

2.3.1. CUDA 流

Compute Unified Device Architecture (CUDA) 是由 NVIDIA 开发的并行计算平台和编程模型，用于 GPU 上的常规计算。

流是 GPU 上问题顺序执行的操作序列。CUDA 命令通常在默认流中按顺序执行，任务在前面的任务完成后才会启动。

跨不同流的异步处理操作允许并行执行任务。在一个流中发布的任务之前、期间或另一个任务签发到另一个流后运行。这允许 GPU 以任何规定的顺序同时运行多个任务，从而提高性能。

其他资源

- 异步并发执行

2.3.2. Time-slicing（时间分片）

当您运行多个 CUDA 应用程序时，在超载 GPU 上调度 GPU 时间的交集工作负载。

您可以通过为 GPU 定义一组副本来启用 Kubernetes 上的 GPU 的时间，每个副本都可以独立分发到 pod 来运行工作负载。与多实例 GPU (MIG) 不同，在副本之间不存在内存或故障隔离，但对于某些工作负载，这优于根本不进行共享。在内部，GPU 时间分片用于从同一底层 GPU 将多个工作负载分配到不同的副本。

对于时间分片，您可以应用一个集群范围的默认配置。您还可以应用特定于节点的配置。例如，您只能将时间分片配置应用到具有 Tesla T4 GPU 的节点，而不能将节点改为使用其他 GPU 模型。

您可以通过应用集群范围的默认配置来组合这两种方法，然后标记节点以授予这些节点接收特定于节点的配置。

2.3.3. CUDA 多进程服务

CUDA 多进程服务 (MPS) 允许单个 GPU 使用多个 CUDA 进程。进程在 GPU 上并行运行，消除了 GPU 计算资源的饱和。MPS 还支持并发执行或重叠内核操作和从不同进程复制的内存，以增强利用率。

其他资源

- CUDA MPS

2.3.4. 多实例 GPU

使用多实例 GPU (MIG)，您可以将 GPU 计算单元和内存分成多个 MIG 实例。每个实例都代表了从系统的角度来看的独立 GPU 设备，并可连接到节点上运行的任何应用程序、容器或虚拟机。使用 GPU 的软件将每个 MIG 实例视为单独的 GPU。

当您有一个不需要整个 GPU 完全功能的应用程序时，MIG 很有用。新的 NVIDIA Ampere 架构的 MIG 功能允许您将硬件资源分成多个 GPU 实例，每个实例都可作为独立的 CUDA GPU 提供给操作系统。

NVIDIA GPU Operator 版本 1.7.0 及更高版本为 A100 和 A30 Ampere 卡提供 MIG 支持。这些 GPU 实例旨在支持最多 7 个独立的 CUDA 应用程序，以便它们与专用硬件资源完全隔离。

其他资源

- NVIDIA 多实例 GPU 用户指南

2.3.5. 使用 vGPU 的虚拟化

虚拟机 (VM) 可以使用 NVIDIA vGPU 直接访问单个物理 GPU。您可以创建虚拟 GPU，供虚拟机在企业之间共享，并由其他设备访问。

此功能将 GPU 性能与 vGPU 提供的管理和安全优势相结合。vGPU 提供的其他好处包括虚拟机环境的主动管理和监控、混合 VDI 和计算工作负载的工作负载平衡以及多个虚拟机之间的资源共享。

其他资源

- 虚拟 GPU

2.4. OPENSHIFT CONTAINER PLATFORM 的 NVIDIA GPU 功能

NVIDIA Container Toolkit

NVIDIA Container Toolkit 可让您创建并运行 GPU 加速容器。工具包包括容器运行时库和工具，用于自动配置容器以使用 NVIDIA GPU。

NVIDIA AI Enterprise

NVIDIA AI Enterprise 是端到端的云原生 AI 和数据分析软件套件，由 NVIDIA 认证系统进行优化、认证和支持。
NVIDIA AI Enterprise 包括对 Red Hat OpenShift Container Platform 的支持。支持以下安装方法：

- 带有 GPU Passthrough 的裸机或 VMware vSphere 上的 OpenShift Container Platform。
- 带有 NVIDIA vGPU 的 VMware vSphere 上的 OpenShift Container Platform。

GPU 功能发现

NVIDIA GPU Feature Discovery for Kubernetes 是一个软件组件，可让您为节点上可用的 GPU 自动生成标签。GPU 功能发现使用节点功能发现(NFD)来执行此标记。
Node Feature Discovery Operator (NFD)通过使用硬件特定信息标记节点来管理 OpenShift Container Platform 集群中硬件功能和配置的发现。NFD 使用特定于节点的属性标记主机，如 PCI 卡、内核、操作系统版本等。

您可以通过搜索 "Node Feature Discovery" 在 Operator Hub 中找到 NFD Operator。

带有 OpenShift Virtualization 的 NVIDIA GPU Operator

到目前为止，GPU Operator 只置备了 worker 节点来运行 GPU 加速的容器。现在，GPU Operator 也可以用来置备 worker 节点来运行 GPU 加速的虚拟机 (VM)。

您可以根据将 GPU 工作负载配置为在这些节点上运行，将 GPU Operator 配置为将不同的软件组件部署到 worker 节点。

GPU 监控仪表板

您可以安装监控仪表板，在 OpenShift Container Platform Web 控制台的集群 **Observe** 页面中显示 GPU 用量信息。GPU 使用率信息包括可用 GPU 数、功耗（watts）、温度（Celsius）、利用率（百分比）以及其他每个 GPU 的指标。

其他资源

- NVIDIA 认证系统
- NVIDIA AI Enterprise
- NVIDIA Container Toolkit
- 启用 GPU 监控仪表板
- OpenShift Container Platform 中的 MIG 支持
- OpenShift 中的 NVIDIA GPU 时间分片
- 在断开连接的或 airgapped 环境中部署 GPU Operator
- Node Feature Discovery Operator

第 3 章 AMD GPU OPERATOR

AMD Instinct GPU 加速器与 OpenShift Container Platform 集群中的 AMD GPU Operator 相结合，可让您无缝利用计算功能进行机器学习、生成 AI 和 GPU 加速的应用程序。

本文档提供了启用、配置和测试 AMD GPU Operator 所需的信息。如需更多信息，请参阅 AMD Instinct™ 加速器。

3.1. 关于 AMD GPU OPERATOR

AMD GPU Operator 的硬件加速功能为数据科学家和开发人员提供增强的性能和成本效率，使用 Red Hat OpenShift AI 创建人工智能和机器学习(AI/ML)应用程序。加快 GPU 功能的特定区域可以最小化 CPU 处理和内存用量，提高整体应用程序速度、内存消耗和带宽限制。

3.2. 安装 AMD GPU OPERATOR

作为集群管理员，您可以使用 OpenShift CLI 和 Web 控制台安装 AMD GPU Operator。这是一个多步骤的过程，需要安装 Node Feature Discovery Operator、内核模块管理 Operator，然后是 AMD GPU Operator。使用以下步骤安装 Operator 的 AMD 社区版本。

后续步骤

1. 安装 Node Feature Discovery Operator。
2. 安装内核模块管理 Operator。
3. 安装和配置 AMD GPU Operator。

3.3. 测试 AMD GPU OPERATOR

使用以下步骤测试 ROCmInfo 安装并查看 AMD MI210 GPU 的日志。

流程

1. 创建测试 ROCmInfo 的 YAML 文件：

```
$ 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. 创建 **rocminfo** pod:

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

输出示例

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

3. 使用一个 MI210 GPU 检查 **rocmnfo** 日志：

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

输出示例

```
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. 删除 Pod：

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

输出示例

```
pod "rocminfo" deleted
```

第 4 章 INTEL GAUDI AI ACCELERATORS

您可以为 OpenShift Container Platform generative AI 和机器学习(AI/ML)应用程序使用 Intel Gaudi AI 加速器。Intel Gaudi AI 加速器为优化深度学习工作负载提供了一个具有成本效益、灵活且可扩展的解决方案。

红帽支持 Intel Gaudi 2 和 Intel Gaudi 3 设备。Intel Gaudi 3 设备在培训速度和能源效率方面提供了显著改进。

4.1. INTEL GAUDI AI ACCELERATORS 先决条件

- 您有一个正常工作的 OpenShift Container Platform 集群，至少有一个 GPU worker 节点。
- 您可以使用 cluster-admin 访问 OpenShift Container Platform 集群，以执行必要的步骤。
- 已安装 OpenShift CLI (**oc**)。
- 已安装 Node Feature Discovery (NFD) Operator，并创建了 **NodeFeatureDiscovery** 实例。

其他资源

- OpenShift 安装 (Intel Gaudi 文档)
- Intel Gaudi AI 加速器集成

第 5 章 NVIDIA GPUDIRECT REMOTE DIRECT MEMORY ACCESS (RDMA)

NVIDIA GPUDirect Remote Direct Memory Access (RDMA) 允许一个计算机中的应用程序直接访问另一个计算机的内存，而无需通过操作系统访问。这提供了一个在进程中绕过内核干预的功能，释放资源并大大减少了处理网络通信所需的 CPU 开销。这可用于在集群中分发 GPU 加速的工作负载。因为 RDMA 非常适合高带宽和低延迟应用程序，因此这使其成为大数据和机器学习应用程序的理想选择。

目前，NVIDIA GPUDirect RDMA 有三种配置方法：

共享设备

这个方法允许 NVIDIA GPUDirect RDMA 设备在公开设备的 OpenShift Container Platform worker 节点上的多个 pod 共享。

主机设备

此方法通过在 pod 上创建额外的主机网络，在 worker 节点上提供直接物理以太网访问。插件允许将网络设备从主机网络命名空间移到 pod 上的网络命名空间。

SR-IOV 传统设备

Single Root IO 虚拟化(SR-IOV)方法可以在多个 pod 间共享一个单个网络设备，如以太网适配器。SR-IOV 会将设备（在主机节点上被识别为物理功能(PF)）分段为多个虚拟功能(VF)。VF 和其它网络设备一样使用。

每种方法都可以在 NVIDIA GPUDirect RDMA over Converged Ethernet (RoCE)或 Infiniband 基础架构中使用，提供总计 6 种配置方法。

5.1. NVIDIA GPUDIRECT RDMA 的先决条件

NVIDIA GPUDirect RDMA 配置的所有方法都需要安装特定的 Operator。使用以下步骤安装 Operator：

- 安装 Node Feature Discovery Operator。
- 安装 SR-IOV Operator。
- 安装 NVIDIA Network Operator (NVIDIA 文档)。
- 安装 NVIDIA GPU Operator (NVIDIA 文档)。

5.2. 禁用 IRDMA 内核模块

在一些系统中，包括 DellR750xa，IRDMA 内核模块在卸载和加载 DOCA 驱动程序时为 NVIDIA Network Operator 造成问题。使用以下步骤禁用该模块。

流程

1. 运行以下命令生成以下机器配置文件：

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

输出示例

```
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. 运行以下命令，在集群中创建机器配置并等待节点重新引导：

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

输出示例

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

3. 运行以下命令，在每个节点上验证模块尚未加载的 debug pod:

```
$ 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. 创建持久性命名规则

在某些情况下，设备名称在重启后不会保留。例如，在 R760xa 系统上 Mellanox 设备重启后可能会重命名。您可以使用 **MachineConfig** 设置持久性来避免此问题。

流程

1. 将 worker 节点的 MAC 地址名称收集到文件中，并为需要保留的接口提供名称。这个示例使用文件 **70-persistent-net.rules** 来存储详情。

```
$ 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. 将该文件转换为没有换行符的 base64 字符串，并将输出设置为变量 **PERSIST** ：

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

```
$ echo $PERSIST
```

```
U1VCU1lTVEVNPT0ibmV0IixBQ1RJT049PSJhZGQiLEFUVFJ7YWRkcmVzc309PSJiODozZjpkMjozYjo1MToyOCIsQVRUUnt0eXBlfT09IjEiLE5BTUU9ImliczJmMCIKU1VCU1lTVEVNPT0ibmV0IixBQ1RJT049PSJhZGQiLEFUVFJ7YWRkcmVzc309PSJiODozZjpkMjozYjo1MToyOSIsQVRUUnt0eXBlfT09IjEiLE5BTUU9ImVuczhmMG5wMCIKU1VCU1lTVEVNPT0ibmV0IixBQ1RJT049PSJhZGQiLEFUVFJ7YWRkcmVzc309PSJiODozZjpkMjpmMDozNjpkMCIsQVRUUnt0eXBlfT09IjEiLE5BTUU9ImliczJmMCIKU1VCU1lTVEVNPT0ibmV0IixBQ1RJT049PSJhZGQiLEFUVFJ7YWRkcmVzc309PSJiODozZjpkMjpmMDozNjpkMSIsQVRUUnt0eXBlfT09IjEiLE5BTUU9ImVuczhmMG5wMCIK
```

3. 运行以下命令，创建机器配置并在自定义资源文件中设置 base64 编码：

```
$ 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. 运行以下命令在集群中创建机器配置。运行命令后，预期的输出显示 **machineconfig.machineconfiguration.openshift.io/99-machine-config-udev-network created**。

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

5. 使用 **get mcp** 命令查看机器配置状态：

```
$ oc get mcp
```

输出示例

```
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
```

节点将重新引导，并在 update 字段返回到 **false** 时，您可以通过查看 debug pod 中的设备来验证节点上的设备。

5.4. 配置 NFD OPERATOR

Node Feature Discovery（NFD）Operator通过将节点标记为硬件特定信息来管理 OpenShift Container Platform 集群中硬件功能和配置的检测。NFD 使用特定于节点的属性标记主机，如 PCI 卡、内核、操作系统版本等。

先决条件

- 已安装 NFD Operator。

流程

1. 运行以下命令，验证 Operator 是否已安装并运行 **openshift-nfd** 命名空间中的 pod：

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

输出示例

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

2. 在 NFD 控制器运行时，生成 **NodeFeatureDiscovery** 实例，并将其添加到集群中。NFD Operator 的 **ClusterServiceVersion** 规格提供默认值，包括作为 Operator 有效负载一部分的 NFD 操作对象镜像。运行以下命令来检索其值：

```
$ 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. 可选：在默认的 **deviceClassWhiteList** 字段中添加条目来支持更多网络适配器，如 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. 运行以下命令来创建 'NodeFeatureDiscovery' 实例：

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

输出示例

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

5. 运行以下命令，通过查看 **openshift-nfd** 命名空间中的 pod 来验证实例是否已启动并正在运行：

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

输出示例

```
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. 等待短暂的时间，然后验证 NFD 是否已向节点添加标签。NFD 标签使用 **feature.node.kubernetes.io** 前缀，以便您可以轻松过滤它们。

```
$ 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. 确认发现了一个网络设备：

```
$ 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. 配置 SR-IOV OPERATOR

单根 I/O 虚拟化(SR-IOV)通过从单一设备在多个 pod 之间提供共享来提高 NVIDIA GPUDirect RDMA 的性能。

先决条件

- 已安装 SR-IOV Operator。

流程

1. 运行以下命令，验证 Operator 是否已安装并运行 **openshift-sriov-network-operator** 命名空间中的 pod：

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

输出示例

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

2. 对于默认的 **SriovOperatorConfig** CR 以用于 MLNX_OFED 容器，请运行这个命令来更新以下值：

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

3. 运行以下命令在集群中创建资源：

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

输出示例

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

4. 运行以下命令修补 sriov-operator，以便 MOFED 容器可以使用它：

```
$ 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" } } }'
```

输出示例

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

5.6. 配置 NVIDIA NETWORK OPERATOR

NVIDIA network Operator 管理 NVIDIA 网络资源和网络相关组件，如驱动程序和设备插件来启用 NVIDIA GPUDirect RDMA 工作负载。

先决条件

- 已安装 NVIDIA network Operator。

流程

1. 运行以下命令，验证 network Operator 是否已安装并运行，确认控制器是否在 **nvidia-network-operator** 命名空间中运行：

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

输出示例

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

2. 在 Operator 运行时，创建 **NicClusterPolicy** 自定义资源文件。您选择的设备取决于您的系统配置。在这个示例中，Infiniband 接口 **ibs2f0** 是硬编码的，用作共享的 NVIDIA GPUDirect RDMA 设备。

```
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. 运行以下命令在集群中创建 **NicClusterPolicy** 自定义资源：

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

输出示例

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

4. 在 DOCA/MOFED 容器中运行以下命令来验证 **NicClusterPolicy** ：

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

输出示例

```
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** 到 **mofed** 容器，运行以下命令来检查状态：

```
$ 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
```

输出示例

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

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

输出示例

```
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. 创建 **IPoIBNetwork** 自定义资源文件：

```
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. 运行以下命令在集群中创建 **IPoIBNetwork** 资源：

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

输出示例

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

8. 为其他接口创建一个 **MacvlanNetwork** 自定义资源文件：

```
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. 运行以下命令在集群中创建资源：

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

输出示例

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

5.7. 配置 GPU OPERATOR

GPU Operator 自动管理 NVIDIA 驱动程序、GPU 的设备插件、NVIDIA Container Toolkit 和 GPU 置备所需的其他组件。

先决条件

- 已安装 GPU Operator。

流程

1. 运行以下命令，检查 Operator pod 是否正在运行以查看命名空间中的 pod：

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

输出示例

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

2. 创建 GPU 集群策略自定义资源文件，如下例所示：

```
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. 当生成 GPU **ClusterPolicy** 自定义资源时，运行以下命令在集群中创建资源：

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

输出示例

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

4. 运行以下命令验证 Operator 是否已安装并正在运行：

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

输出示例

```
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. 可选：验证 pod 正在运行时，远程 shell 到 NVIDIA 驱动程序 daemonset pod，并确认已载入 NVIDIA 模块。具体来说，请确保载入 **nvidia_peermem**。

```
$ 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
```

输出示例

```
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. 可选：运行 **nvidia-smi** 工具来显示驱动程序和硬件的详情：

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

+ .输出示例

```
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. 当您仍然在驱动程序 pod 中时，使用 **nvidia-smi** 命令将 GPU 时钟设置为最大值：

```
$ 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)
```

输出示例

```
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)
```

输出示例

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

2. 运行以下命令，从节点描述视角验证资源是否可用：

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

输出示例

```
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. 创建机器配置

在创建资源 pod 之前，您需要创建 **machineconfig.yaml** 自定义资源 (CR) 来提供 GPU 和网络资源的访问权限，而无需用户权限。

流程

1. 生成 **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. 创建工作负载 POD

使用本节中的步骤为共享和主机设备创建工作负载 pod。

5.9.1. 在 RoCE 上创建共享设备 RDMA

为 NVIDIA Network Operator 在 RDMA 上为共享设备 RDMA 创建工作负载 pod，并测试 pod 配置。

NVIDIA GPUDirect RDMA 设备在公开设备的 OpenShift Container Platform worker 节点上的 pod 共享。

先决条件

- 确保 Operator 正在运行。
- 删除 **NicClusterPolicy** 自定义资源 (CR) （如果存在）。

流程

1. 生成自定义 pod 资源：

```
$ 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. 使用以下命令在集群中创建 pod：

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

输出示例

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

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

输出示例

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

3. 使用以下命令验证 pod 是否正在运行：

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

输出示例

```
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. 在 RoCE 上创建主机设备 RDMA

为 NVIDIA Network Operator 为主机设备 Remote Direct Memory Access (RDMA) 创建工作负载 pod，并测试 pod 配置。

先决条件

- 确保 Operator 正在运行。
- 删除 **NicClusterPolicy** 自定义资源 (CR) （如果存在）。

流程

1. 生成一个新的主机设备 **NicClusterPolicy** (CR)，如下所示：

```
$ 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. 使用以下命令在集群中创建 **NicClusterPolicy** CR：

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

输出示例

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

3. 在 DOCA/MOFED 容器中使用以下命令验证主机设备 **NicClusterPolicy** CR:

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

输出示例

```
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. 使用以下命令确认资源出现在集群 **oc describe node** 部分中：

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

输出示例

```
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. 创建 **HostDeviceNetwork** CR 文件：

```
$ 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. 使用以下命令在集群中创建 **HostDeviceNetwork** 资源：

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

输出示例

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

7. 使用以下命令确认资源出现在集群 **oc describe node** 部分中：

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

输出示例

```
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. 在 RoCE 上创建 SR-IOV 旧模式 RDMA

在 RoCE 上配置单根 I/O 虚拟化 (SR-IOV) 旧模式主机设备 RDMA。

流程

1. 生成一个新的主机设备 **NicClusterPolicy** 自定义资源 (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. 使用以下命令在集群中创建策略：

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

输出示例

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

3. 在 DOCA/MOFED 容器中使用以下命令验证 pod：

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

输出示例

```
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. 为您要在 SR-IOV 传统模式下运行的设备生成虚拟功能 (VF) 的 **SriovNetworkNodePolicy** CR。请参见以下示例：

```
$ 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. 使用以下命令在集群中创建 CR：

注意

确保启用了 SR-IOV 全局启用。如需更多信息，请参阅无法启用 SR-IOV，并在 Red Hat Enterprise Linux 中接收消息 "not enough MMIO resources for SR-IOV "。

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

输出示例

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

6. 每个节点都被禁用调度。节点重新引导以应用配置。您可以使用以下命令查看节点：

```
$ oc get nodes
```

输出示例

```
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. 节点重启后，通过在每个节点中打开 debug pod 来验证 VF 接口是否存在。运行以下命令:

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

输出示例

```
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. 如有必要，在第二个节点上重复前面的步骤。

9. 可选：使用以下命令确认资源出现在集群 **oc describe node** 部分中：

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

输出示例

```
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. SR-IOV 传统模式的 VF 就绪后，生成 **SriovNetwork** CR 文件。请参见以下示例：

```
$ 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. 使用以下命令在集群中创建自定义资源：

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

输出示例

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

5.9.4. 在 Infiniband 上创建共享设备 RDMA

为 Infiniband 安装为共享设备远程内存访问 (RDMA) 创建工作负载 pod。

流程

1. 生成自定义 pod 资源：

```
$ 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. 使用以下命令在集群中创建 pod：

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

–

输出示例

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

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

输出示例

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

3. 使用以下命令验证 pod 是否正在运行：

```
$ oc get pods
```

输出示例

```
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. 验证 RDMA 连接

确认远程直接内存访问 (RDMA) 连接在系统间正常工作，特别是传统的单根 I/O 虚拟化 (SR-IOV) 以太网。

流程

1. 使用以下命令连接到每个 **rdma-workload-client** pod：

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

输出示例

```
sh-5.1#
```

2. 使用以下命令，检查分配给第一个工作负载 pod 的 IP 地址。在本例中，第一个工作负载 pod 是 RDMA 测试服务器。

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

输出示例

```
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#
```

分配给此 pod 的 RDMA 服务器的 IP 地址是 **net1** 接口。在本例中，IP 地址为 **192.168.4.225**。

3. 运行 **ibstatus** 命令以获取与每个 RDMA 设备 **mlx5_x** 关联的 **link_layer** 类型、以太网或 Infiniband。输出中还通过检查 **state** 字段来显示所有 RDMA 设备的状态，该字段会显示 **ACTIVE** 或 **DOWN**。

```
sh-5.1# ibstatus
```

输出示例

```
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. 要获取 worker 节点上每个 RDMA **mlx5** 设备的 **link_layer**，请运行 **ibstat** 命令：

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

输出示例

```
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. 对于 RDMA 共享设备或主机设备工作负载 pod，名为 **mlx5_x** 的 RDMA 设备已经已知，通常是 **mlx5_0** 或 **mlx5_1**。对于 RDMA 传统 SR-IOV 工作负载 pod，您需要确定哪个 RDMA 设备与哪些虚拟功能 (VF) 子接口相关联。使用以下命令提供此信息：

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

输出示例

```
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
```

在这个示例中，RDMA 设备名称 **mlx5_7** 与 **net1** 接口相关联。下一个命令中使用此输出来执行 RDMA 带宽测试，该测试还会验证 worker 节点之间的 RDMA 连接。

6. 运行以下 **ib_write_bw** RDMA 带宽测试命令：

```
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
```

其中：

- **mlx5_7** RDMA 设备在 **-d** 交换机中传递。
- 源 IP 地址为 **192.168.4.225** 来启动 RDMA 服务器。
- **--use_cuda=0**, **--use_cuda_dmabuf** 交换表示使用 GPUDirect RDMA。

输出示例

```
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. 打开另一个终端窗口，并在作为 RDMA 测试客户端 pod 的第二个工作负载 pod 上运行 **oc rsh** 命令：

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

输出示例

```
sh-5.1#
```

8. 使用以下命令，从 **net1** 接口获取 RDMA 测试客户端 pod IP 地址：

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

输出示例

```
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. 使用以下命令，获取与每个 RDMA 设备 **mlx5_x** 关联的 **link_layer** 类型：

```
sh-5.1# ibstatus
```

输出示例

```
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. 可选：使用 **ibstat** 命令获取 Mellanox 卡的固件版本：

```
sh-5.1# ibstat
```

输出示例

```
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. 要确定哪个 RDMA 设备与客户端工作负载 pod 使用的 Virtual Function 子接口相关联，请运行以下命令。在本例中，**net1** 接口使用 RDMA 设备 **mlx5_2**。

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

输出示例

```
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. 运行以下 **ib_write_bw** RDMA 带宽测试命令：

```
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
```

其中：

- **mlx5_2** RDMA 设备在 **-d** 交换机中传递。
- 源 IP 地址 **192.168.4.226** 以及 RDMA 服务器 **192.168.4.225** 的目标 IP 地址。
- **--use_cuda=0**, **--use_cuda_dmabuf** 交换表示使用 GPUDirect RDMA。

输出示例

```
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#
```

一个正的测试会在 Mpps 中看到一个预期的 BW 平均和 MsgRate。

在完成 **ib_write_bw** 命令后，服务器端输出也会出现在服务器 pod 中。请参见以下示例：

输出示例

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

```
 remote address: LID 0000 QPN 0x0139 PSN 0x6cf53d
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
 remote address: LID 0000 QPN 0x013a PSN 0xcaaf6f
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
 remote address: LID 0000 QPN 0x013b PSN 0x346437
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
 remote address: LID 0000 QPN 0x013c PSN 0xcc5865
 GID: 00:00:00:00:00:00:00:00:00:00:255:255:192:168:04:226
---------------------------------------------------------------------------------------
 #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
```

第 6 章 DYNAMIC ACCELERATOR SLICER (DAS) OPERATOR

重要

Dynamic Accelerator Slicer Operator 只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议（SLA）支持，且功能可能并不完整。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关红帽技术预览功能支持范围的更多信息，请参阅以下链接：

- 技术预览功能支持范围

Dynamic Accelerator Slicer (DAS) Operator 允许您在 OpenShift Container Platform 中动态分片 GPU 加速器，而不依赖在节点引导时定义的静态分片 GPU。这可让您根据特定工作负载需求动态分片 GPU，确保有效的资源利用率。

如果您不知道集群中每个节点前需要的所有加速器分区，则动态分片很有用。

DAS Operator 目前包含 NVIDIA Multi-Instance GPU (MIG) 的参考实施，并在以后支持其他供应商（如 NVIDIA MPS 或 GPU）的其它技术。

限制：

使用 Dynamic Accelerator Slicer Operator 时有以下限制：

- 您需要识别潜在的不兼容性，并确保系统可以与各种 GPU 驱动程序和操作系统无缝配合工作。
- Operator 只适用于特定的 MIG 兼容 NVIDIA GPU 和驱动程序，如 H100 和 A100。
- Operator 无法只使用节点的 GPU 的子集。
- NVIDIA 设备插件无法与 Dynamic Accelerator Slicer Operator 一起使用来管理集群的 GPU 资源。

注意

DAS Operator 设计为使用启用 MIG 的 GPU。它分配 MIG 片段而不是整个 GPU。安装 DAS Operator 可防止通过 NVIDIA 设备插件（如 **nvidia.com/gpu: "1"** ）使用标准资源请求来分配整个 GPU。

6.1. 安装 DYNAMIC ACCELERATOR SLICER OPERATOR

作为集群管理员，您可以使用 OpenShift Container Platform Web 控制台或 OpenShift CLI 安装 Dynamic Accelerator Slicer (DAS) Operator。

6.1.1. 使用 Web 控制台安装 Dynamic Accelerator Slicer Operator

作为集群管理员，您可以使用 OpenShift Container Platform Web 控制台安装 Dynamic Accelerator Slicer (DAS) Operator。

先决条件

- 可以使用具有 **cluster-admin** 权限的账户访问 OpenShift Container Platform 集群。

- 已安装所需的先决条件：
  - cert-manager Operator for Red Hat OpenShift
  - Node Feature Discovery (NFD) Operator
  - NVIDIA GPU Operator
  - NodeFeatureDiscovery CR

流程

1. 为 MIG 支持配置 NVIDIA GPU Operator：
   a. 在 OpenShift Container Platform web 控制台中导航至 **Operators → Installed Operators**。
   b. 从安装的 Operator 列表中选择 **NVIDIA GPU Operator**。
   c. 点 **ClusterPolicy** 选项卡，然后点 **Create ClusterPolicy**。
   d. 在 YAML 编辑器中，将默认内容替换为以下集群策略配置，以禁用默认的 NVIDIA 设备插件并启用 MIG 支持：

```
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. 点 **Create** 应用集群策略。

f. 进入到 **Workloads → Pods**，再选择 **nvidia-gpu-operator** 命名空间来监控集群策略部署。

g. 等待 NVIDIA GPU Operator 集群策略达到 **Ready** 状态。您可以通过以下方法监控它：

   i. 进入到 **Operators → Installed Operators → NVIDIA GPU Operator**。

   ii. 点 **ClusterPolicy** 选项卡，检查状态是否显示 **ready**。

h. 选择 **nvidia-gpu-operator** 命名空间并进入到 **Workloads → Pods** 来验证 NVIDIA GPU Operator 命名空间中的所有 pod 是否正在运行。

i. 使用支持 MIG 的 GPU 标签节点以启用 MIG 模式：

   i. 进入 **Compute → Nodes**。

   ii. 选择具有 MIG 功能 GPU 的节点。

   iii. 点 **Actions → Edit Labels**。

   iv. 添加标签 **nvidia.com/mig.config=all-enabled**。

   v. 点击 **Save**。

   vi. 对具有 MIG 功能 GPU 的每个节点重复此操作。

重要

应用 MIG 标签后，标记的节点将重新引导以启用 MIG 模式。等待节点恢复在线，然后继续。

j. 通过检查 **nvidia.com/mig.config=all-enabled** 标签出现在 **Labels** 部分中，验证 GPU 节点上的 MIG 模式是否已成功启用。要找到标签，进入到 **Compute → Nodes**，选择 GPU 节点，然后点 **Details** 选项卡。

2. 在 OpenShift Container Platform Web 控制台中，点击 **Operators → OperatorHub**。

3. 在过滤器框中搜索 **Dynamic Accelerator Slicer** 或 **DAS**，以查找 DAS Operator。

4. 选择 **Dynamic Accelerator Slicer**，再点 **Install**。

5. 在 **Install Operator** 页中：

a. 为安装模式选择 **All namespaces on the cluster (default)**。

b. 选择 **Installed Namespace → Operator recommended Namespace: Project das-operator**。

c. 如果创建新命名空间，请输入 **das-operator** 作为命名空间名称。

d. 选择一个更新频道。

e. 为批准策略选择 **Automatic** 或 **Manual**。

6. 点 **Install**。

7. 在 OpenShift Container Platform web 控制台中，点击 **Operators → Installed Operators**。

8. 从列表中选择 **DAS Operator**。

9. 在 **Provided APIs** 表列中，点 **DASOperator**。这会进入 **Operator 详情**页的 **DAS Operator** 选项卡。

10. 点 **Create DASOperator**。这会进入 **Create DASOperator** YAML 视图。

11. 在 YAML 编辑器中，粘贴以下示例：

DASOperator CR 示例

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

(1) **DASOperator** CR 的名称必须是 **cluster**。

12. 点 **Create**。

验证

验证 DAS Operator 是否已成功安装：

1. 导航到 **Operators → Installed Operators** 页面。

2. 确保 **das-operator** 命名空间中列出了 **Dynamic Accelerator Slicer**，**Status** 为 **Succeeded**。

验证 **DASOperator** CR 是否已成功安装：

- 创建 **DASOperator** CR 后，Web 控制台会进入 **DASOperator 列表视图**。当所有组件都运行时，CR 的 **Status** 字段将变为 **Available**。

- 可选。您可以通过在 OpenShift CLI 中运行以下命令来验证 **DASOperator** CR 是否已成功安装：

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

输出示例

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

注意

在安装过程中，Operator 可能会显示 **Failed** 状态。如果安装后安装成功并显示 **Succeeded** 信息，您可以忽略 **Failed** 信息。

您还可以通过检查 pod 来验证安装：

1. 进入到 **Workloads → Pods** 页，再选择 **das-operator** 命名空间。
2. 验证所有 DAS Operator 组件 pod 是否正在运行：
   - **DAS-operator** pod （主 operator 控制器）
   - **dAS-operator-webhook** pod (webhook 服务器)
   - **dAS-scheduler** pod (scheduler 插件)
   - **DAS-daemonset** pod （仅在具有 MIG 兼容 GPU 的节点上）

注意

das-daemonset pod 只会出现在具有 MIG 兼容的 GPU 硬件的节点上。如果没有看到任何 daemonset pod，请验证集群是否有带有支持的 GPU 硬件的节点，并且 NVIDIA GPU Operator 是否已正确配置。

故障排除

如果没有安装 Operator，请使用以下步骤：

1. 导航到 **Operators → Installed Operators** 页面，检查 **Operator Subscriptions** 和 **Install Plans** 选项卡中的 **Status** 项中是否有任何错误。
2. 进入到 **Workloads → Pods** 页，在 **das-operator** 命名空间中检查 pod 的日志。

其他资源

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

6.1.2. 使用 CLI 安装动态加速器 Slicer Operator

作为集群管理员，您可以使用 OpenShift CLI 安装 Dynamic Accelerator Slicer (DAS) Operator。

先决条件

- 可以使用具有 **cluster-admin** 权限的账户访问 OpenShift Container Platform 集群。
- 已安装 OpenShift CLI(**oc**)。
- 已安装所需的先决条件：
  - cert-manager Operator for Red Hat OpenShift
  - Node Feature Discovery (NFD) Operator
  - NVIDIA GPU Operator
  - NodeFeatureDiscovery CR

流程

1. 为 MIG 支持配置 NVIDIA GPU Operator：
   a. 应用以下集群策略来禁用默认的 NVIDIA 设备插件并启用 MIG 支持。使用以下内容创建名为 **gpu-cluster-policy.yaml** 的文件：

```
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. 运行以下命令来应用集群策略：

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

c. 运行以下命令，验证 NVIDIA GPU Operator 集群策略是否达到 **Ready** 状态：

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

等待 **STATUS** 列显示 **ready**。

输出示例

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

d. 运行以下命令，验证 NVIDIA GPU Operator 命名空间中的所有 pod 是否正在运行：

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

所有 pod 都应该会显示 **Running** 或 **Completed** 状态。

e. 运行以下命令，使用 MIG 功能 GPU 标记节点以启用 MIG 模式：

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

将 **$NODE_NAME** 替换为具有 MIG 功能 GPU 的每个节点的名称。

重要

应用 MIG 标签后，标记的节点重新引导以启用 MIG 模式。等待节点恢复在线，然后继续。

f. 运行以下命令验证节点是否已成功启用 MIG 模式：

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

2. 为 DAS Operator 创建命名空间：

a. 创建定义 **das-operator** 命名空间的以下 **Namespace** 自定义资源(CR)，并将 YAML 保存到 **das-namespace.yaml** 文件中：

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

b. 运行以下命令创建命名空间：

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

3. 通过创建以下对象，在您上一步中创建的命名空间中安装 DAS Operator：

a. 创建以下 **OperatorGroup** CR，并在 **das-operatorgroup.yaml** 文件中保存 YAML：

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

b. 运行以下命令来创建 **OperatorGroup** CR:

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

c. 创建以下 **Subscription** CR，并将 YAML 保存到 **das-sub.yaml** 文件中：

订阅示例

```
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. 运行以下命令来创建订阅对象：

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

e. 进入 **das-operator** 项目：

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

f. 创建以下 **DASOperator** CR，并在 **das-dasoperator.yaml** 文件中保存 YAML：

DASOperator CR 示例

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

(1) **DASOperator** CR 的名称必须是 **cluster**。

g. 运行以下命令来创建 **dasoperator** CR：

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

验证

- 运行以下命令验证 Operator 部署是否成功：

```
$ oc get pods
```

输出示例

```
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
```

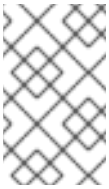
成功部署会显示状态为 **Running** 的所有 Pod。部署包括：

das-operator
主 Operator 控制器 pod

das-operator-webhook
用于变异 pod 请求的 Webhook 服务器 pod

das-scheduler
用于 MIG 分片分配的调度程序插件 pod

das-daemonset
仅在具有 MIG 兼容 GPU 的节点上运行的 DaemonSet pod



注意

das-daemonset pod 只会出现在具有 MIG 兼容的 GPU 硬件的节点上。如果没有看到任何 daemonset pod，请验证集群是否有带有支持的 GPU 硬件的节点，并且 NVIDIA GPU Operator 是否已正确配置。

其他资源

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

6.2. 卸载动态加速器 SLICER OPERATOR

根据 Operator 的安装方式，使用以下流程之一卸载 Dynamic Accelerator Slicer (DAS) Operator。

6.2.1. 使用 Web 控制台卸载 Dynamic Accelerator Slicer Operator

您可以使用 OpenShift Container Platform Web 控制台卸载 Dynamic Accelerator Slicer (DAS) Operator。

先决条件

- 可以使用具有 **cluster-admin** 权限的账户访问 OpenShift Container Platform 集群。
- DAS Operator 已安装在集群中。

流程

1. 在 OpenShift Container Platform web 控制台中导航至 **Operators → Installed Operators**。
2. 在安装的 Operator 列表中找到 **Dynamic Accelerator Slicer**。
3. 点 DAS Operator 的 **Options** 菜单 并选择 **Uninstall Operator**。
4. 在确认对话框中，点 **Uninstall** 确认删除。
5. 浏览至 **Home → Project**。
6. 在搜索框中搜索 **das-operator** 以查找 DAS Operator 项目。
7. 点 das-operator 项目 旁边的 **Options** 菜单 ，然后选择 **Delete Project**。
8. 在确认对话框中，在对话框中键入 **das-operator**，然后点 **Delete** 确认删除。

验证

1. 导航到 **Operators → Installed Operators** 页面。
2. 验证 Dynamic Accelerator Slicer (DAS) Operator 是否不再被列出。
3. 可选。运行以下命令，验证 **das-operator** 命名空间及其资源是否已移除：

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

该命令应该会返回指示未找到命名空间的错误。

警告

卸载 DAS Operator 会删除所有 GPU 分片分配，并可能导致运行依赖于 GPU 分片的工作负载失败。在继续卸载前，请确保没有关键工作负载使用 GPU 片段。

6.2.2. 使用 CLI 卸载动态加速器 Slicer Operator

您可以使用 OpenShift CLI 卸载 Dynamic Accelerator Slicer (DAS) Operator。

先决条件

- 可以使用具有 **cluster-admin** 权限的账户访问 OpenShift Container Platform 集群。
- 已安装 OpenShift CLI(**oc**)。
- DAS Operator 已安装在集群中。

流程

1. 运行以下命令，列出已安装的 Operator 以查找 DAS Operator 订阅：

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

输出示例

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

2. 运行以下命令来删除订阅：

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

3. 运行以下命令，列出并删除集群服务版本 (CSV)：

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

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

4. 运行以下命令来删除 operator 组：

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

5. 运行以下命令来删除任何剩余的 **AllocationClaim** 资源：

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

6. 运行以下命令来删除 DAS Operator 命名空间：

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

验证

1. 运行以下命令验证 DAS Operator 资源是否已移除：

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

该命令应该会返回指示未找到命名空间的错误。

2. 运行以下命令验证没有剩余的 **AllocationClaim** 自定义资源定义：

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

该命令应该会返回指示没有找到自定义资源定义的错误。

警告

卸载 DAS Operator 会删除所有 GPU 分片分配，并可能导致运行依赖于 GPU 分片的工作负载失败。在继续卸载前，请确保没有关键工作负载使用 GPU 片段。

6.3. 使用 DYNAMIC ACCELERATOR SLICER OPERATOR 部署 GPU 工作负载

您可以部署请求 GPU 片段由 Dynamic Accelerator Slicer (DAS) Operator 管理的工作负载。Operator 动态分区 GPU 加速器，并将工作负载调度到可用的 GPU 分片。

先决条件

- 在集群中有 MIG 支持的 GPU 硬件。
- 安装 NVIDIA GPU Operator，**ClusterPolicy** 会显示 **Ready** 状态。
- 已安装 DAS Operator。

流程

1. 运行以下命令来创建命名空间：

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

2. 创建使用 NVIDIA MIG 资源请求 GPU 资源的部署：

```
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. 运行以下命令来应用部署配置：

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

4. 运行以下命令验证部署是否已创建并调度 pod：

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

输出示例

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

5. 运行以下命令，检查 pod 的状态：

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

输出示例

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

验证

1. 运行以下命令，检查是否为部署 pod 创建 **AllocationClaim** 资源：

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

输出示例

```
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. 运行以下命令，验证 GPU 片段是否已正确分配 pod 的资源分配：

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

3. 运行以下命令，检查日志以验证 CUDA 示例应用程序是否已成功运行：

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

输出示例

```
[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. 运行以下命令，检查环境变量以验证 GPU 设备是否已正确公开给容器：

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

输出示例

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

这些环境变量表示 GPU MIG 片段已被正确分配，并出现在容器内的 CUDA 运行时。

6.4. 对动态加速器 SLICER OPERATOR 进行故障排除

如果您在 Dynamic Accelerator Slicer (DAS) Operator 时遇到问题，请使用以下故障排除步骤诊断和解决问题。

先决条件

- 已安装 DAS Operator。
- 您可以使用具有 cluster-admin 角色的用户访问 OpenShift Container Platform 集群。

6.4.1. 调试 DAS Operator 组件

流程

1. 运行以下命令，检查所有 DAS Operator 组件的状态：

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

输出示例

```
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. 运行以下命令，检查 DAS Operator 控制器的日志：

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

3. 运行以下命令，检查 webhook 服务器的日志：

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

4. 运行以下命令，检查调度程序插件的日志：

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

5. 运行以下命令，检查设备插件 daemonset 的日志：

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

6.4.2. Monitoring AllocationClaims

流程

1. 运行以下命令检查活跃的 **AllocationClaim** 资源：

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

输出示例

```
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. 运行以下命令，查看有关特定 **AllocationClaim** 的详细信息：

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

输出示例（截断）

```
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. 运行以下命令，检查不同状态中的声明：

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

输出示例

```
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. 运行以下命令，查看与 **AllocationClaim** 资源相关的事件：

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

5. 运行以下命令，检查 **NodeAccelerator** 资源以验证 GPU 硬件检测：

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

输出示例

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

NodeAccelerator 资源代表 DAS Operator 检测到的 GPU 功能的节点。

其他信息

AllocationClaim 自定义资源跟踪以下信息：

GPU UUID

GPU 设备的唯一标识符。

分片位置

GPU 上的 MIG 片段的位置。

Pod 参考

请求 GPU 片段的 pod。

状态

当前的声明状态 (**staged**, **created**, 或 **released**)。

声明以 **staged** 状态开始，然后当所有请求都满足后会转变为 **created**。删除 pod 时，会自动清理关联的声明。

6.4.3. 验证 GPU 设备可用性

流程

1. 在使用 GPU 硬件的节点上，运行以下命令来验证 CDI 设备是否已创建：

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

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

2. 运行以下命令检查 NVIDIA GPU Operator 状态：

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

输出应显示 **ready**。

6.4.4. 增加日志详细程度

流程

获取更详细的调试信息：

1. 运行以下命令来编辑 **DASOperator** 资源来提高日志详细程度：

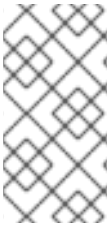
```
$ oc edit dasoperator -n das-operator
```

2. 将 **operatorLogLevel** 字段设置为 **Debug** 或 **Trace** ：

```
spec:
  operatorLogLevel: Debug
```

3. 保存更改，并验证 Operator pod 重启并增加详细程度。

6.4.5. 常见问题和解决方案



POD 处于 UNEXPECTEDADMISSIONERROR 状态

由于 kubernetes/kubernetes vmcore8043，如果准入失败，pod 可能会进入 **UnexpectedAdmissionError** 状态。由更高级别的控制器管理的 Pod 会自动重新创建。但是，naked pod 必须使用 **oc delete pod** 手动清理。建议使用控制器，直到上游问题解决为止。

未满足先决条件

如果 DAS Operator 无法启动或正常工作，请验证是否已安装所有先决条件：

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

其他资源

- Kubernetes issue #128043
- Node Feature Discovery Operator
- NVIDIA GPU Operator 故障排除