Menu

硬件加速器

概述

介绍

硬件加速器介绍

产品优势

应用场景

技术限制

功能概览

vGPU（基于开源 GPU-Manager）

pGPU（NVIDIA 设备插件）

MPS（NVIDIA 多进程服务插件）

安装

安装

安装 Kubernetes 硬件加速工具包

应用开发

介绍

应用开发简介

功能指南

故障排除

配置管理

介绍

配置管理概述

功能指南

资源监控

介绍

资源监控介绍

优势

应用场景

使用限制

功能指南

Menu

概述

介绍

硬件加速器介绍

产品优势

应用场景

技术限制

功能概览

vGPU（基于开源 GPU-Manager）

pGPU（NVIDIA 设备插件）

MPS（NVIDIA 多进程服务插件）

介绍

目录

硬件加速器介绍

Kubernetes 硬件加速器套件是一种企业级解决方案，旨在优化云原生环境中 GPU 资源的分配、隔离和共享。该套件基于 Kubernetes 设备插件和 NVIDIA 原生技术，提供三个核心模块：

1. **vGPU** 模块
   基于开源 GPU-Manager，此模块通过将物理 GPU 切分为可共享的虚拟单元（具备内存/计算配额）来实现细粒度的 GPU 虚拟化。非常适合需要动态资源分配的多租户环境。

2. **pGPU** 模块
   利用 NVIDIA 官方设备插件，提供具有 NUMA 感知调度的完整物理 GPU 隔离。专为需要专用 GPU 访问的高性能计算（HPC）工作负载设计。

3. **MPS** 模块
   实现 NVIDIA 的多进程服务，允许在资源限制下并发执行 GPU 上下文。通过 CUDA 内核融合优化对延迟敏感的应用程序。

产品优势

vGPU 模块

- 动态切片：将 GPU 切分以支持多进程使用一个物理 GPU
- **QoS** 强制执行：保证计算单元（vcuda-core）和内存配额（vcuda-memory）

pGPU 模块

- 硬件级隔离：直接 PCIe 直通，加上 IOMMU 保护
- **NUMA** 优化：通过自动 NUMA 节点绑定最小化跨套接字数据传输

MPS 模块

- 低延迟执行：通过 CUDA 上下文融合实现 30-50% 的延迟减少
- 资源限制：限制每个进程的 GPU 计算（0-100%）和内存使用
- 零代码更改：适用于未修改的 CUDA 应用程序

应用场景

vGPU 用例

- 多租户 **AI** 平台：在团队之间共享 A100/H100 GPU，并提供保底服务水平协议（SLA）
- 虚拟桌面基础环境（**VDI**）：为 CAD/3D 渲染提供 GPU 加速虚拟桌面
- 批量推理：通过分配部分 GPU 实现模型服务并行化

pGPU 用例

- **HPC** 集群：为天气模拟运行独占 GPU 访问的 MPI 作业
- 机器学习训练：充分利用 GPU 进行大语言模型的训练
- 医学影像：处理高分辨率 MRI 数据，而不发生资源争用

MPS 用例

- 实时推理：使用并发的 CUDA 流进行低延迟视频分析
- 微服务编排：在共享硬件上共同承载多个 GPU 微服务
- 高并发服务：为推荐系统提升 3 倍的每秒查询率（QPS）

技术限制

需要特权

硬件设备访问要求

设备文件权限
NVIDIA GPU 设备需要直接访问受保护的系统资源：

```
# 设备文件所有权和权限
$ ls -l /dev/nvidia*
crw-rw-rw- 1 root root 195,   0 Aug 1 10:00 /dev/nvidia0
crw-rw-rw- 1 root root 195, 255 Aug 1 10:00 /dev/nvidiactl
crw-rw-rw- 1 root root 195, 254 Aug 1 10:00 /dev/nvidia-uvm
```

- 要求：根用户访问以读/写设备文件
- 后果：非根容器会出现权限不足错误

内核级操作

NVIDIA 驱动程序的基本交互

操作	特权要求	目的
模块加载	CAP_SYS_MODULE	加载 NVIDIA 内核模块
内存管理	CAP_IPC_LOCK	GPU 内存分配
中断处理	CAP_SYS_RAWIO	处理 GPU 中断

K8s 设备插件架构要求

1. 创建套接字：写入 `/var/lib/kubelet/device-plugins`
2. 健康监测：访问 `nvidia-smi` 和内核日志
3. 资源分配：修改设备控制组

vGPU 限制

- 仅支持 CUDA 版本低于 12.4
- vGPU 启用时不支持 MIG

pGPU 限制

- 不具备 GPU 共享能力（1 Pod 到 GPU 映射）
- 需要 Kubernetes 1.25 及以上版本，并开启 SR-IOV
- 限制在 PCIe/NVSwitch 连接的 GPU 上

MPS 限制

- 在融合上下文之间可能出现故障传播
- 需要 CUDA 11.4 及以上版本以实现内存限制

- 不支持 MIG 切片的 GPU

功能概览

目录

vGPU（基于开源 GPU-Manager）

- 细粒度资源切割
  将物理 GPU 核心切分为 1-100 个配额。支持动态分配，适用于 AI 推理和虚拟桌面等多租户环境。
- 拓扑感知调度
  自动优先考虑 NVLink/C2C 连接的 GPU，以最小化跨插槽数据传输延迟。确保用于分布式训练工作负载的最佳 GPU 配对。

pGPU（NVIDIA 设备插件）

- **NUMA** 优化分配
  强制执行 1
  GPU 与 Pod 的映射，并绑定 NUMA 节点，减少高速计算（HPC）任务（如 LLM 训练）中的 PCIe 总线争用。
- 独占硬件访问
  通过 PCIe 直通提供完全的物理 GPU 隔离，非常适合需要确定性性能的关键任务应用程序

（如医疗影像处理）。

MPS（NVIDIA 多进程服务插件）

- 延迟优化执行
  实现跨进程的 CUDA 核心融合，减少实时应用（如视频分析）的推理延迟 30-50%。
- 带上限的资源共享
  允许并发 GPU 上下文执行，同时通过环境变量强制每个进程的计算（0-100%）和内存限制。

安装

目录

安装 **Kubernetes** 硬件加速工具包

先决条件

在安装之前，请确保满足以下要求：

1. **Kubernetes** 集群：版本 ≥1.25，并启用了 `DevicePlugins` 功能开关。
2. **NVIDIA** 驱动程序：已安装在所有 GPU 节点上。通过 `nvidia-smi` 验证。
3. 容器运行时：已配置 NVIDIA 容器工具包（≥1.7.0）以支持 GPU。

通过 **Web** 控制台安装

1. 导航到集群插件：
   - 前往 平台管理 → 目录 → 集群插件
   - 搜索 "gpu" 并点击 安装
2. 功能切换： 在安装过程中启用/禁用高级功能：

选项	功能	推荐场景
PGPU	物理 GPU 隔离，支持 NUMA 感知调度	AI 训练/高性能计算
vGPU	通过 GPU-Manager 进行虚拟 GPU 切片	多租户共享/资源配额
MPS	多进程服务用于计算/内存共享	低延迟推理/并行任务

Menu

应用开发

介绍

介绍

应用开发简介

功能指南

CUDA 驱动与运行时兼容性

分层架构与核心概念

版本兼容矩阵与约束

部署最佳实践

故障排查手册

使用 ConfigMap 添加自定义设备

介绍

特性

优势

功能模块 1：ConfigMap 编写规范

功能模块 2：资源值定义

故障排除

排查 **vLLM** 中仅支持计算能力至少为 **xx** 的 **float16** 错误

问题描述

根本原因

故障排查

解决方案

预防措施

相关内容

Paddle Autogrow 内存分配在 **GPU-Manager** 上崩溃

问题描述

根本原因

解决方案

验证方法

预防措施

相关内容

介绍

目录

应用开发简介

应用开发模块指导用户通过统一接口配置多个供应商（例如，AMD/Intel GPU、FPGA）的硬件加速器，从而在容器化环境中实现异构计算资源的协调和优化，以提升高性能工作负载，如人工智能训练和图像处理。

Menu

功能指南

CUDA 驱动与运行时兼容性

分层架构与核心概念

版本兼容矩阵与约束

部署最佳实践

故障排查手册

使用 ConfigMap 添加自定义设备

介绍

特性

优势

功能模块 1：ConfigMap 编写规范

功能模块 2：资源值定义

CUDA 驱动与运行时兼容性

目录

分层架构与核心概念

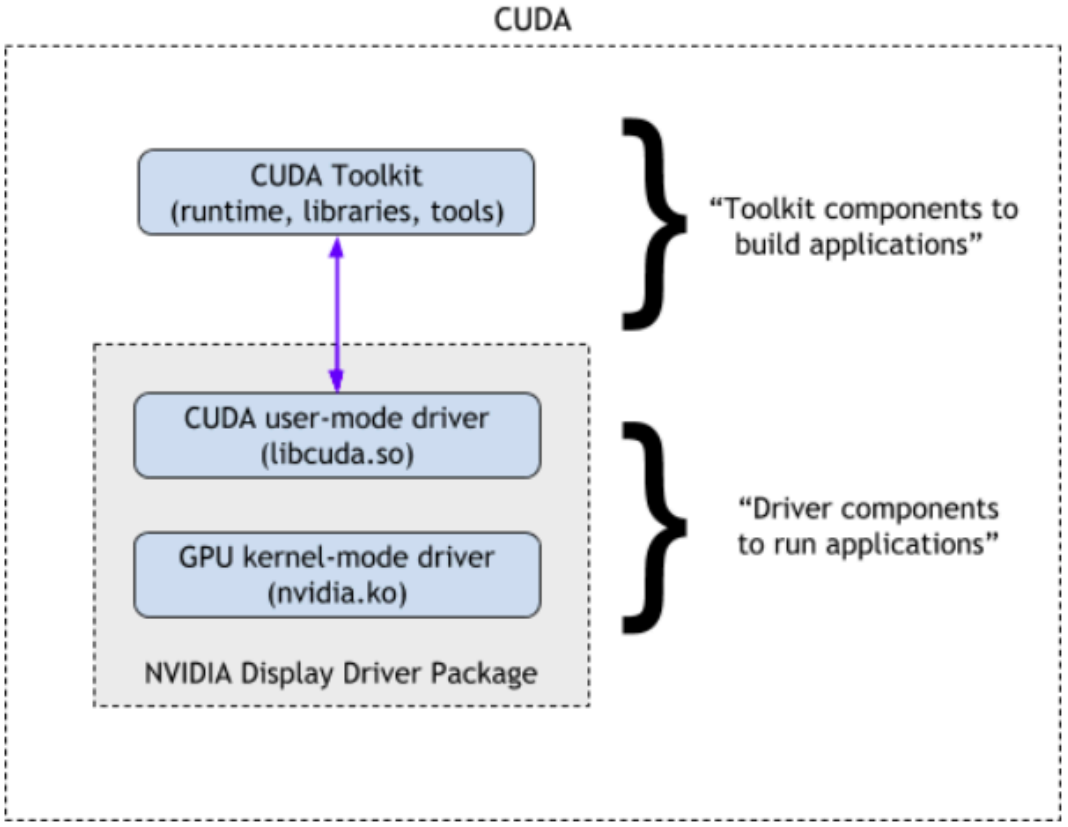

Figure 1: Components of CUDA #

1. CUDA Runtime API 层

技术定位

1. 功能范围：为开发者提供高层抽象接口，封装核心 GPU 操作（内存分配、流管理、内核启动等）
2. 版本绑定：由构建时使用的 CUDA Toolkit 版本决定（例如 CUDA 12.0.1）

版本检测方法

```
# Python 环境检测（推荐优先方法）
pip list | grep cuda
conda list |grep cuda
# 示例输出：cu121  # cu121 表示 CUDA 12.1 环境

# 系统级运行时库检测
find / -name "libcudart*"
# cudart 表示 cuda runtime
# 示例输出:
/usr/local/cuda-12.4/targets/x86_64-linux/lib/libcudart.so.12
/usr/local/cuda-12.4/targets/x86_64-linux/lib/libcudart.so.12.4.127
表示 CUDA 12.4
```

如果发现多个库版本，应检查程序实际使用的版本，如 PATH、LD_LIBRARY_PATH 或其他程序设置

```
env |grep PATH
# 示例输出：
LIBRARY_PATH=/usr/local/cuda/lib64/stubs
LD_LIBRARY_PATH=/usr/local/nvidia/lib:/usr/local/nvidia/lib64
PATH=/go/bin:/usr/local/go/bin:/usr/local/nvidia/bin:/usr/local/cuda/bin:/usr
# 表示你的 cuda 程序使用的是 lib 路径顺序中的第一个库
```

2. CUDA Driver API 层

技术定位

1. 功能范围：低层接口，直接与 GPU 硬件交互，负责指令翻译和硬件资源调度
2. 版本绑定：由 NVIDIA 驱动版本决定，遵循 SemVer 规范

版本检测方法

```
nvidia-smi
# 示例输出：
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 550.144.03             Driver Version: 550.144.03     CUDA Versi
|-----------------------------------------+------------------------+----------
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util
|                                         |                        |
|=========================================+========================+==========
|   0  NVIDIA A30                     Off |   00000000:00:0B.0 Off |
| N/A   31C    P0             28W /  165W |   10195MiB /  24576MiB |      0%
|                                         |                        |
+-----------------------------------------+------------------------+----------
```

版本兼容矩阵与约束

物理 **GPU** 部署 **-** 核心兼容原则

首先参考 NVIDIA 官方声明，基本约束为

1. 驱动版本必须始终 **≥** 运行时版本
2. NVIDIA 官方保证 向后兼容 **1** 个主版本（例如 CUDA Driver 12.x 支持 Runtime 11.x）
3. 跨两个主版本兼容（例如 Driver 12.x 与 Runtime 10.x）既不官方支持也不推荐

部署 cuda 程序时，请遵守基本约束

正式规则

```
+ 强制：驱动版本 ≥ 运行时版本
+ 推荐：驱动主版本 - 运行时主版本 ≤ 1
- 阻断：驱动版本 < 运行时版本 → 可能触发 CUDA_ERROR_UNKNOWN(999)
- 不稳定：驱动主版本 - 运行时主版本 > 1 → 应用可能异常
```

虚拟化场景增强（**HAMI/GPU-Manager**）

使用 GPU-Manager 或 HAMI 等虚拟 GPU 方案时，除遵守上述基本约束外，还必须满足以下额外约束：

版本要求

```
1. 虚拟 GPU 方案基线版本 ≥ 运行时版本
2. 运行时主版本 = 驱动主版本 = 基线主版本
```

GPU-Manager 特别说明： 我们实现了部分跨 1 主版本兼容（例如基线 12.4 支持 vLLM 11.8），但这需要针对每个应用进行 hook 调整，需逐案分析。

部署最佳实践

推荐策略

• 新 GPU 集群规划时，建议采用较新 CUDA 版本（如 CUDA 12.x）作为驱动和运行时版本

旧系统替代方案

1. 物理 GPU 调度或 GPU-Manager 整卡分配

整卡调度提供与物理 GPU 访问等效的原生兼容性
GPU-Manager 可通过设置 tencent.com/vcuda-core 为 100 的正整数倍（如 100、200、300）启用整卡模式

```yaml
resources:
  limits:
    tencent.com/vcuda-core: "100"
```

2. 节点标签策略

根据支持的驱动 CUDA 版本为节点打标签：

```
node_labels:
  cuda-major-version: "12"
  cuda-minor-version: "4"
```

表示该节点为 cuda 12.4

在部署时配置调度亲和性：
可根据程序所需的 cuda 运行时版本设置 cuda-major-version 和 cuda-minor-version

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cuda-app
spec:
  template:
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: cuda-major-version
                operator: In
                values: ["12"]
              - key: cuda-minor-version
                operator: Gt
                values: ["2"]
```

3. 运行时版本升级

旧版 CUDA Runtime 可能存在安全漏洞（CVE）且不支持新 GPU 特性，优先升级至 CUDA 12.x。

NVIDIA 推荐同时升级驱动和运行时

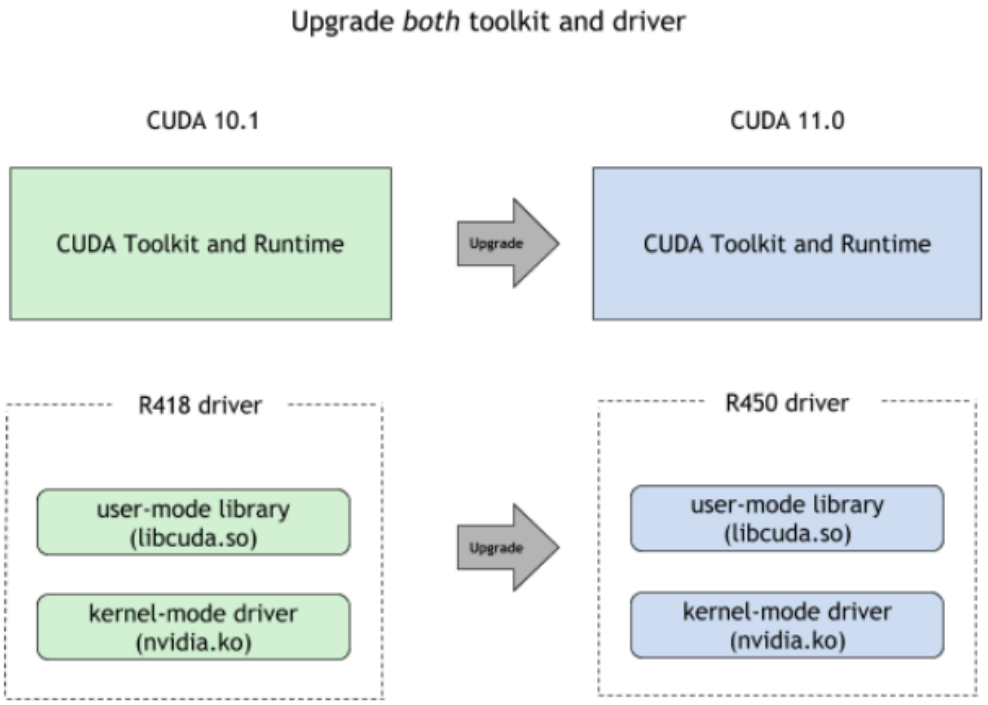

Figure 2: CUDA Upgrade Path

故障排查手册

常见错误代码

错误代码	描述	推荐操作
CUDA_ERROR_INVALID_IMAGE	驱动与运行时不兼容	使驱动版本与容器 CUDA 版本保持一致
CUDA_ERROR_ILLEGAL_ADDRESS	虚拟内存违规（版本不匹配常见）	核实运行时与基线约束是否符合
CUDA_ERROR_UNSUPPORTED_PTX_VERSION	PTX 指令集不匹配	使用显式 -arch=sm_xx 重新编译

使用 **ConfigMap** 添加自定义设备

目录

介绍

- 通过 ConfigMap 实现 Kubernetes 自定义资源的标准化定义和管理，解决以下问题：
- 自定义资源规范的统一管理，防止配置碎片化
- 标准化的资源定义格式，便于维护
- 支持多语言描述和默认值配置
- 适用于需要扩展 Kubernetes 资源模型的场景（例如，GPU 资源管理），提供标准化的资源定义框架

特性

- 单键资源定义规范
- 多键关联资源定义
- 标准化的资源请求接口
- 中英文双语描述支持
- 资源默认值配置机制

优势

- 扩展性：通过标签进行资源组管理
- 安全性：命名空间隔离（kube-public）
- 稳定性：强制格式验证规则
- 可维护性：统一的元数据标签规范

功能模块 1：**ConfigMap** 编写规范

核心规则

1. 单一职责原则：每个键定义对应一个 ConfigMap
2. 命名空间：固定为 `namespace=kube-public`
3. 命名约定：

```
cf-crl-{customName}-{keyName}
```

- `cf-crl`：固定前缀

- `customName`：自定义有效名称
- `keyName`：键标识符（特殊字符用 '-' 替换）

4. 标签要求：

```
labels:
  features.alauda.io/type: CustomResourceLimitation  # 固定值
  features.alauda.io/group: {resource-group}          # 例如，gpu-manager
  features.alauda.io/enabled: "true"                 # 启用标志
```

参数规范

参数	必填	描述
name format	是	遵循 cf-crl-{customName}-{keyName}
namespace	是	固定为 kube-public
label group	是	必须包含指定的 3 个特征标签

功能模块 2：资源值定义

单键示例

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: cf-crl-gpu-manager-vcuda-core
  namespace: kube-public
  labels:
    features.alauda.io/type: CustomResourceLimitation
    features.alauda.io/group: gpu-manager
    features.alauda.io/enabled: "true"
data:
  key: "tencent.com/vcuda-core"        # 资源键
  dataType: "integer"                  # 值类型
  defaultValue: "20"                   # 默认值
  descriptionZh: "" # 中文描述
  descriptionEn: "GPU vcore count, 100 virtual cores equal 1 physical GPU cor
  group: "gpu-manager"                 # 资源组
  limits: "optional"                   # 限制字段策略
  requests: "disabled"                 # 请求字段策略
```

多键关联

```yaml
metadata:
  name: cf-crl-gpu-manager-vcuda-core
  labels: [相同的组标签]

metadata:
  name: cf-crl-gpu-manager-vcuda-memory
  labels: [相同的组标签]  # 通过相同标签进行关联
```

策略规范

字段	允许值	描述
limits	disabled/required/optional	资源限制配置
requests	disabled/required/fromLimits	资源请求配置

Menu

故障排除

排查 **vLLM** 中仅支持计算能力至少为 **xx** 的 **float16** 错误

问题描述

根本原因

故障排查

解决方案

预防措施

相关内容

Paddle Autogrow 内存分配在 **GPU-Manager** 上崩溃

问题描述

根本原因

解决方案

验证方法

预防措施

相关内容

排查 **vLLM** 中仅支持计算能力至少为 **xx** 的 **float16** 错误

目录

问题描述

环境

- 硬件: NVIDIA GPU，计算能力 <8.0（例如，Tesla V100、T4）
- 模型类型: 需要 bfloat16/FP8 精度的 LLM（例如，LLaMA-2-70B、GPT-NeoX-20B）

症状

1. 明确的错误信息：

```
ValueError: float16/bfloat16 仅在计算能力至少为 8.0 的 GPU 上受支持
```

2. 在模型加载期间内核编译失败

相关日志

```
# vLLM 错误堆栈跟踪
File "/usr/local/lib/python3.10/site-packages/vllm/model_executor/layers/quan
    raise ValueError(
ValueError: bfloat16 仅在计算能力至少为 8.0 的 GPU 上受支持。当前 GPU: Tesla V100-P
```

根本原因

主要原因

GPU 计算能力不足 GPU 的计算能力（CC）未满足特定数据类型的最低要求：

- **bfloat16/FP8**: 需要 CC ≥8.0（安培架构及更新）
- **FP16** 张量核心优化: 需要 CC \≥7.0（伏打架构及更新）

技术分析

1. 架构限制：

   - 前安培 GPU（CC <8.0）缺乏用于 bfloat16 操作的专用矩阵数学单元
   - 伏打/图灵中的张量核心（CC 7.0-7.5）仅支持 FP16/FP32 混合精度

2. 框架强制性：

```
# vLLM 的能力检查（简化版）
def _verify_cuda_compute_capability():
    if device.compute_capability < MIN_REQUIRED_CC:
        raise ValueError(f"需要计算能力 ≥{MIN_REQUIRED_CC}")
```

故障排查

第一步：验证 **GPU** 计算能力

```
import torch
print(f"计算能力: {torch.cuda.get_device_capability()}")
```

第二步：检查模型精度要求

```
cat model/config.json | grep "torch_dtype"
# 预期输出: "bfloat16" 或 "float16"
```

第三步：验证框架兼容性

```
from vllm import _is_cuda_compute_capability_compatible as compat
print(f"支持 bfloat16: {compat((8,0))}")
```

解决方案

针对计算能力不足的解决方案

注意事项

- 降低精度时预计性能下降
- 不同精度类型可能导致模型准确性变化

前提条件

- CUDA 工具包 ≥11.8

步骤

1. 修改 **InferenceService yaml**： 添加参数如 --dtype=half

```yaml
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: llama-2-service
  annotations:
    serving.kserve.io/enable-prometheus-scraping: "true"
spec:
  predictor:
    containers:
    - name: kserve-container
      image: vllm/vllm-serving:0.3.2
      args:
        - --model=meta-llama/Llama-2-7b-chat-hf
        - --dtype=half  # 强制 FP16 精度
        - --tensor-parallel-size=1
      resources:
        limits:
          nvidia.com/gpu: "1"
```

2. 等待部署重启

预防措施

1. 前期检查：

```
from vllm import LLM
LLM.validate_environment(model_dtype="bfloat16")
```

2. 集群配置：

```
# NVIDIA 设备插件配置
helm upgrade -i nvidia-device-plugin \
  --set compatabilityPolicy=strict \
  --set computeCapabilities=8.0+
```

3. 模型优化：

```
# 应用 AWQ 量化
llm = LLM(model="codellama/CodeLlama-34b",
          quantization="awq",
          load_format="awq")
```

相关内容

GPU 计算能力参考

架构	CC 范围	支持的精度
伏打	7.0-7.2	FP16 张量核心
图灵	7.5	FP16/INT8
安培	8.0-8.9	bfloat16/TF32/FP8

架构	**CC** 范围	支持的精度
霍普尔	9.0+	FP4/FP8 动态缩放

官方参考

1. NVIDIA 计算能力表↗
2. vLLM 硬件要求↗

Paddle Autogrow 内存分配在 GPU-Manager 上崩溃

目录

问题描述

症状

当同时启用 PaddlePaddle 的 Autogrow 内存分配策略和 GPU-Manager 的虚拟化内存管理时，可能会发生以下异常：

1. 由于内存分配不连续导致的 OOM 错误
2. 异常的 GPU 利用率波动
3. 随机的训练过程崩溃
4. nvidia-smi 报告与框架统计之间的内存使用不一致

根本原因

根本原因分析

1. 内存分配策略冲突 Paddle 的 Autogrow 使用动态分段分配，而 GPU-Manager 的虚拟化要求连续的物理内存映射
2. 管理机制不兼容 Autogrow 的延迟释放机制与 GPU-Manager 的内存回收策略发生冲突
3. 元数据维护冲突 两个系统各自进行的元数据维护导致内存视图不一致

触发机制：

- Autogrow 在分配时尝试最优块大小
- GPU-Manager 虚拟化层拦截物理内存请求
- 非连续的分配导致虚拟地址映射失败
- 双重管理导致元数据一致性异常

解决方案

解决方案概述

通过环境变量强制 Paddle 使用传统的分配策略：

```
FLAGS_allocator_strategy=naive_best_fit
```

注意事项

1. 需要重启训练过程
2. 可能会降低 Paddle 的内存重用效率

实施步骤

Kubernetes 部署

1. 编辑部署配置

```
apiVersion: apps/v1
kind: Deployment
spec:
  template:
    spec:
      containers:
      ◦ name: paddle-container
        env:
        ▪ name: FLAGS_allocator_strategy
          value: "naive_best_fit"
```

1. 应用配置

```
kubectl apply -f updated_deployment.yaml
```

1. 验证配置

```
kubectl exec <pod-name> -- env | grep FLAGS
```

Bare Metal 部署

1. 在执行前设置环境变量

```
export FLAGS_allocator_strategy=naive_best_fit
python train.py
```

1. 或在 Python 代码中设置

```
import os
os.environ['FLAGS_allocator_strategy'] = 'naive_best_fit'
```

验证方法

1. 检查日志中的分配策略确认

```
I0715 14:25:17.112233 12345 allocator.cc:256]
Using Naive Best Fit allocation strategy
```

1. 监测内存分配的连续性

```
nvidia-smi --query-gpu=memory.used --format=csv -l 1
```

1. 压力测试验证

```
# 连续分配测试脚本
import paddle
for i in range(10):
    data = paddle.randn([1024, 1024, 100], dtype='float32')
    print(f"分配了 {i+1}GB")
```

预防措施

1. 版本兼容性检查 在升级时查看 Paddle 版本更新记录中的内存分配变更

2. 监测配置 添加 Prometheus 警报规则：

```
- alert: GPUAllocConflict
  expr: rate(paddle_gpu_malloc_failed_total[5m]) > 0
  labels:
    severity: critical
  annotations:
    summary: "GPU 内存分配冲突警报"
```

3. 基线测试 在新环境中进行内存分配基线测试：

```
python -c "import paddle; paddle.utils.run_check()"
```

相关内容

内存分配策略比较

策略	优势	劣势
autogrow	高效	大块性能差
naive_best_fit	稳定分配	可能导致碎片化

参考文献

Paddle 内存优化白皮书↗

Menu

配置管理

介绍

介绍

配置管理概述

功能指南

在 **GPU** 节点上配置硬件加速器

前提条件

物理 GPU 配置

NVIDIA MPS 配置（驱动支持 CUDA 版本必须 >= 11.5）

GPU 管理器配置

结果验证

介绍

目录

配置管理概述

配置管理是一个集中化的文档门户，用于在 Kubernetes 环境中配置 GPU 加速功能。该活文档为管理员提供了统一的指导，以便在混合基础设施上设置物理 GPU (pGPU)、虚拟 GPU (vGPU) 和多进程服务 (MPS) 配置。

Menu

功能指南

在 **GPU** 节点上配置硬件加速器

前提条件

物理 GPU 配置

NVIDIA MPS 配置（驱动支持 CUDA 版本必须 >= 11.5）

GPU 管理器配置

结果验证

在 GPU 节点上配置硬件加速器

随着业务数据的增加，尤其是在人工智能和数据分析等场景中，您可能希望在自建业务集群中利用 GPU 功能来加速数据处理。除了为集群节点准备 GPU 资源外，还需进行 GPU 配置。

本解决方案将集群中具有GPU计算能力的节点称为 **GPU** 节点。

注意：除非另有说明，操作步骤适用于两种类型的节点。有关驱动程序安装相关问题，请参考 NVIDIA 官方安装文档↗。

目录

前提条件

在操作节点上已准备好 GPU 资源，该节点属于本节提到的 GPU 节点。

安装 GPU 驱动

注意：如果 **GPU** 节点使用 **NVIDIA MPS** 插件，请确保该节点的 **GPU** 架构为 **Volta** 或更新版本（**Volta/Turing/Ampere/Hopper** 等），且驱动程序支持 **CUDA** 版本 **11.5** 或更高版本。

获取驱动下载地址

1. 登录到 GPU 节点，并运行命令 `lspci |grep -i NVIDIA` 检查该节点的 GPU 型号。

   在以下示例中，GPU 型号为 Tesla T4。

   ```
   lspci | grep NVIDIA
   00:08.0 3D controller: NVIDIA Corporation TU104GL [Tesla T4] (rev a1)
   ```

2. 访问 NVIDIA 官方网站↗ 获取驱动下载链接。

   1. 点击首页顶部导航栏中的 **Drivers**。
   2. 根据 GPU 节点型号填写下载驱动所需的信息。
   3. 点击 **Search**。
   4. 点击 **Download**。
   5. 右键点击 **Download** > **Copy Link Address** 复制驱动的下载链接。

3. 在 GPU 节点上执行以下命令，创建 `/home/gpu` 目录，并将驱动文件下载并保存到该目录。

```
# 创建 /home/gpu 目录
mkdir -p /home/gpu
cd /home/gpu/
# 将驱动文件下载到 /home/gpu 目录，示例：wget https://cn.download.nvidia.com/tes
wget <驱动下载地址>
# 验证驱动文件是否成功下载，如果返回驱动文件名（例如：NVIDIA-Linux-x86_64-515.65.01
ls <驱动文件名>
```

安装驱动

1. 在 GPU 节点上执行以下命令，以安装与当前操作系统对应的 gcc 和 kernel-devel 包。

```
sudo yum install dkms gcc kernel-devel-$(uname -r) -y
```

2. 执行以下命令以安装 GPU 驱动。

```
chmod a+x /home/gpu/<驱动文件名>
/home/gpu/<驱动文件名> --dkms
```

3. 安装完成后，执行 `nvidia-smi` 命令。如果返回类似以下示例的 GPU 信息，则表示驱动安装成功。

```
# nvidia-smi
Tue Sep 13 01:31:33 2022
+-----------------------------------------------------------------------------
| NVIDIA-SMI 515.65.01    Driver Version: 515.65.01    CUDA Version: 11.7
+-------------------------------+----------------------+----------------
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. E
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute
|                               |                      |               MIG
|===============================+======================+================
|   0  Tesla T4            Off  | 00000000:00:08.0 Off |
| N/A   55C    P0    28W /  70W |      2MiB / 15360MiB |      5%      Defau
|                               |                      |                  N
+-------------------------------+----------------------+----------------

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

安装 **NVIDIA** 容器运行时

1. 在 **GPU** 节点 上添加 NVIDIA yum 仓库。

```
distribution=$(. /etc/os-release;echo $ID$VERSION_ID) && curl -s -L https:/
yum makecache -y
```

当提示 "Metadata cache created." 出现时，表示添加成功。

2. 安装 NVIDIA 容器运行时。

```
yum install nvidia-container-toolkit -y
```

当提示 `Complete!` 出现时，表示安装成功。

3. 配置默认运行时。

   将以下配置添加到文件中。

   - Containerd: 修改 `/etc/containerd/config.toml` 文件。

```
[plugins]
 [plugins."io.containerd.grpc.v1.cri"]
   [plugins."io.containerd.grpc.v1.cri".containerd]
...
      default_runtime_name = "nvidia"
...
        [plugins."io.containerd.grpc.v1.cri".containerd.runtimes]
...
          [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc]
            runtime_type = "io.containerd.runc.v2"
            runtime_engine = ""
            runtime_root = ""
            privileged_without_host_devices = false
            base_runtime_spec = ""
            [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc
              SystemdCgroup = true
          [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.nvidia
            privileged_without_host_devices = false
            runtime_engine = ""
            runtime_root = ""
            runtime_type = "io.containerd.runc.v1"
            [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.nvid
              BinaryName = "/usr/bin/nvidia-container-runtime"
              SystemdCgroup = true
...
```

   - Docker: 修改 `/etc/docker/daemon.json` 文件。

```
{
...
    "default-runtime": "nvidia",
    "runtimes": {
    "nvidia": {
        "path": "/usr/bin/nvidia-container-runtime"
        }
    },
...
}
```

4. 重启 Containerd / Docker。

   - Containerd

   ```
   systemctl restart containerd   #重启

   crictl info |grep Runtime  #检查
   ```

   - Docker

   ```
   systemctl restart docker   #重启

   docker info |grep Runtime  #检查
   ```

物理 GPU 配置

在 GPU 业务集群上部署物理 GPU 插件

在 GPU 集群的管理界面上执行以下操作：

1. 在目录左侧栏中，选择 "集群插件" 子栏，单击部署 "ACP GPU 设备插件" 并打开 "pGPU" 选项；

2. 在 "节点" 选项卡中，选择需要部署物理 GPU 的节点，然后单击 "标签和污点管理器"，添加 "设备标签" 并选择 "pGPU"，然后单击 OK；
3. 在 "Pods" 选项卡中，检查与 nvidia-device-plugin-ds 对应的容器组运行状态，查看是否有异常，并确保它在指定节点上运行。

NVIDIA MPS 配置（驱动支持 CUDA 版本必须 >= 11.5）

在 GPU 业务集群上部署 NVIDIA MPS 插件

在 GPU 集群的管理界面上执行以下操作：

1. 在目录左侧栏中，选择 "集群插件" 子栏，单击部署 "ACP GPU 设备插件" 并打开 "MPS" 选项；
2. 在 "节点" 选项卡中，选择需要部署物理 GPU 的节点，然后单击 "标签和污点管理器"，添加 "设备标签" 并选择 "MPS"，然后单击 OK；
3. 在 "Pods" 选项卡中，检查与 nvidia-mps-device-plugin-daemonset 对应的容器组运行状态，查看是否有异常，并确保它在指定节点上运行。

配置 kube-scheduler（kubernetes >= 1.23）

1. 在 业务集群控制节点 上，检查调度程序是否正确引用调度策略。

```
cat /etc/kubernetes/manifests/kube-scheduler.yaml
```

检查是否有 –config 选项，并且值为 /etc/kubernetes/scheduler-config.yaml，如下

```yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    component: kube-scheduler
    tier: control-plane
  name: kube-scheduler
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-scheduler
    - --config=/etc/kubernetes/scheduler-config.yaml
```

注意：上述参数和配置为平台的默认配置。如果您已修改它们，请改回默认值。可以将原始自定义配置复制到调度策略文件中。

2. 检查调度策略文件的配置。

    1. 执行命令： `kubectl describe service kubernetes -n default |grep Endpoints` 。

        ```
        期望效果 Endpoints:         192.168.130.240:6443
        ```

    2. 将所有 Master 节点上 `/etc/kubernetes/scheduler-config.yaml` 文件的内容替换为以下内容，其中 `${kube-apiserver}` 应该替换为第一步的输出。

```
apiVersion: kubescheduler.config.k8s.io/v1beta2
kind: KubeSchedulerConfiguration
clientConnection:
  kubeconfig: /etc/kubernetes/scheduler.conf
extenders:
- enableHTTPS: true
  filterVerb: predicates
  managedResources:
  - ignoredByScheduler: false
    name: nvidia.com/mps-core
  nodeCacheCapable: false
  urlPrefix: https://${kube-apiserver}/api/v1/namespaces/kube-system/serv
  tlsConfig:
    insecure: false
    certFile: /etc/kubernetes/pki/apiserver-kubelet-client.crt
    keyFile: /etc/kubernetes/pki/apiserver-kubelet-client.key
    caFile: /etc/kubernetes/pki/ca.crt
```

如果 schedule-config.yaml文件中已存在 extenders，则将 yaml 附加到末尾

```
- enableHTTPS: true
  filterVerb: predicates
  managedResources:
  - ignoredByScheduler: false
    name: nvidia.com/mps-core
  nodeCacheCapable: false
  urlPrefix: https://${kube-apiserver}/api/v1/namespaces/kube-system/serv
  tlsConfig:
    insecure: false
    certFile: /etc/kubernetes/pki/apiserver-kubelet-client.crt
    keyFile: /etc/kubernetes/pki/apiserver-kubelet-client.key
    caFile: /etc/kubernetes/pki/ca.crt
```

3. 运行以下命令以获取容器 ID：

- Containerd：执行 `crictl ps |grep kube-scheduler`，输出如下，第一列为容器 ID。

```
1d113ccf1c1a9       03c72176d0f15       2 seconds ago       Running
```

- Docker：运行 `docker ps |grep kube-scheduler`，输出如下，第一列为容器 ID。

```
30528a45a118   d8a9fef7349c    "kube-scheduler --au..."   37 minutes ago
```

4. 使用上一步获得的容器 ID 重启 Containerd/Docker 容器。

- Containerd

```
crictl stop <容器 ID>
```

5. 重启 Kubelet。

```
systemctl restart kubelet
```

GPU 管理器配置

配置 **kube-scheduler**（**kubernetes >= 1.23**）

1. 在 业务集群控制节点 上，检查调度程序是否正确引用调度策略。

```
cat /etc/kubernetes/manifests/kube-scheduler.yaml
```

检查是否有 –config 选项，并且值为 /etc/kubernetes/scheduler-config.yaml，如下所示

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    component: kube-scheduler
    tier: control-plane
  name: kube-scheduler
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-scheduler
    - --config=/etc/kubernetes/scheduler-config.yaml
```

注意：上述参数和配置为平台的默认配置。如果您已修改它们，请改回默认值。可以将原始自定义配置复制到调度策略文件中。

2. 检查调度策略文件的配置。

    1. 执行命令： `kubectl describe service kubernetes -n default |grep Endpoints` 。

        ```
        期望效果 Endpoints:         192.168.130.240:6443
        ```

    2. 将所有 Master 节点上 `/etc/kubernetes/scheduler-config.yaml` 文件的内容替换为以下内容，其中 `${kube-apiserver}` 应该替换为第一步的输出。

```
apiVersion: kubescheduler.config.k8s.io/v1beta2
kind: KubeSchedulerConfiguration
clientConnection:
  kubeconfig: /etc/kubernetes/scheduler.conf
extenders:
- enableHTTPS: true
  filterVerb: predicates
  managedResources:
  - ignoredByScheduler: false
    name: tencent.com/vcuda-core
  nodeCacheCapable: false
  urlPrefix: https://${kube-apiserver}/api/v1/namespaces/kube-system/serv
  tlsConfig:
    insecure: false
    certFile: /etc/kubernetes/pki/apiserver-kubelet-client.crt
    keyFile: /etc/kubernetes/pki/apiserver-kubelet-client.key
    caFile: /etc/kubernetes/pki/ca.crt
```

3. 运行以下命令以获取容器 ID：

- Containerd：执行 `crictl ps |grep kube-scheduler`，输出如下，第一列为容器 ID。

```
1d113ccf1c1a9       03c72176d0f15       2 seconds ago       Running
```

- Docker：运行 `docker ps |grep kube-scheduler`，输出如下，第一列为容器 ID。

```
30528a45a118   d8a9fef7349c    "kube-scheduler --au..."   37 minutes ago
```

4. 使用上一步获得的容器 ID 重启 Containerd/Docker 容器。

- Containerd

```
crictl stop <容器 ID>
```

5. 重启 Kubelet。

```
systemctl restart kubelet
```

在 **GPU** 业务集群上部署 **GPU** 管理器插件

在 GPU 集群的管理界面上执行以下操作：

1. 在目录左侧栏中，选择 "集群插件" 子栏，单击部署 "ACP GPU 设备插件" 并打开 "GPU-管理器" 选项；
2. 在 "节点" 选项卡中，选择需要部署物理 GPU 的节点，然后单击 "标签和污点管理器"，添加 "设备标签" 并选择 "vGPU"，然后单击 OK；
3. 在 "Pods" 选项卡中，检查与 gpu-manager-daemonset 对应的容器组运行状态，查看是否有异常，并确保它在指定节点上运行。

结果验证

方法 1：在业务集群的控制节点上运行以下命令检查 GPU 节点上是否有可用的 GPU 资源：

```
kubectl get node  ${nodeName} -o=jsonpath='{.status.allocatable}'
```

方法 2：在平台上通过指定所需的 GPU 资源量来部署 GPU 应用程序。部署后，进入 Pod 并执行以下命令：

```
# nvidia-smi
Tue Sep 13 01:31:33 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 515.65.01    Driver Version: 515.65.01    CUDA Version: 11.7     |
+-------------------------------+----------------------+----------------------+
| 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  Tesla T4            Off  | 00000000:00:08.0 Off |                    0 |
| N/A   55C    P0    28W /  70W |      2MiB / 15360MiB |      5%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

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

检查是否正确检索到 GPU 信息。

Menu

资源监控

介绍

介绍

资源监控介绍

优势

应用场景

使用限制

功能指南

GPU资源监控

功能概述

核心功能

功能优势

节点监控

容器组监控

时间范围选择

介绍

目录

资源监控介绍

资源监控是 Kubernetes 硬件加速器套件的关键组成部分，旨在为您容器化工作负载中的 GPU 资源利用率提供全面的可见性。该模块提供实时指标和历史数据分析，涵盖两个基本层面上的计算利用率 和 **GPU** 内存消耗：

资源监控是 Kubernetes 硬件加速器套件的关键组成部分，旨在为您容器化工作负载中的 GPU 资源利用率提供全面的可见性。该模块在两个基本层面提供 计算利用率 和 **GPU** 内存消耗：

- 节点级监控：跟踪整个 Kubernetes 节点的 GPU 资源总使用情况
- 容器组级监控：分析每个工作负载的 GPU 消耗，具有容器组粒度

此监控解决方案与平台的核心加速器模块（pGPU/vGPU（GPU-Manager）/MPS）集成，使用户能够优化 GPU 分配，实施资源配额，并排除 AI/ML 工作负载、实时推理服务等中的性能瓶颈。

优势

资源监控的核心优势如下：

- 多维可观察性

  同时监控物理/虚拟 GPU 的计算单元（CUDA 内核）和内存利用率，提供有关加速器使用模式的全面见解。

- 层次化指标收集

  在节点和容器组粒度下捕获数据，使集群范围的资源趋势与个别工作负载需求之间能够进行关联。

- 原生集成

  与所有加速器模块（pGPU/vGPU/MPS）无缝协作，无需额外代理，利用 Kubernetes 原生指标管道。

- 历史分析

  存储 GPU 指标并设定可配置的保留期（默认 7 天），通过集成可视化工具进行容量规划和使用模式分析。

应用场景

资源监控的主要应用场景如下：

- 性能优化

  在训练集群中识别未充分利用的 GPU，并为深度学习工作负载调整资源请求。例如，检测持续使用 <30% 分配 GPU 内存的容器组，以优化内存分配。

- 多租户治理

  在共享环境中监控 vGPU 消耗，强制执行 GPU 配额合规性。跟踪 AI 平台部署中累积使用情况与分配配额之间的关系。

- 成本归属

  为企业 Kubernetes 环境中的费用分摊/展示模型生成每个命名空间的 GPU 利用报告，将容器组级指标与组织单位相关联。

- 故障诊断

  通过分析容器崩溃前的内存使用趋势来调查 GPU 加速工作负载中的 OOM（内存溢出）事件。与 Kubernetes 事件交叉参考以进行根本原因分析。

- 容量规划

  分析历史 GPU 利用模式（例如，峰值计算需求期），为基础设施扩展决策和 AI 基础设施的预算分配提供参考。

使用限制

使用资源监控时，请注意以下限制：

- 模块依赖性
  - 需要在集群中部署至少一个加速器模块（pGPU/vGPU/MPS）

Menu

功能指南

GPU资源监控

功能概述

核心功能

功能优势

节点监控

容器组监控

时间范围选择

GPU资源监控

目录

功能概述

资源监控功能允许在容器平台中实时和历史性地跟踪各节点和容器组（Pods）内的GPU利用率和内存使用情况。该功能帮助管理员和开发人员：

- 监控**GPU**性能：识别GPU资源分配中的瓶颈。
- 排除故障：分析GPU使用趋势以调试与资源相关的问题。
- 优化工作负载：根据数据做出决策，改善工作负载分布。

适用场景：

- 实时监控GPU密集型应用程序。
- 对GPU利用率进行历史分析，以进行容量规划。
- 多节点/多容器组GPU性能比较。

提供的价值：

- 增强对GPU资源消耗的可视化。
- 通过可操作的见解提高集群效率。

核心功能

- 节点级监控：跟踪每个节点的GPU利用率和内存使用情况。
- 容器组级监控：监控独立容器组的GPU指标。
- 自定义时间范围：分析最近30分钟到7天的数据。

功能优势

- 实时可视化：具有自动刷新功能的互动仪表板。
- 多维度过滤：按GPU类型、命名空间或容器组缩小指标范围。

节点监控

通过以下步骤监控节点级的GPU资源：

1 访问**GPU**仪表板

1. 导航到平台管理视图
2. 转到操作中心 → 监控 → 仪表板
3. 切换到**GPU**目录

2 选择节点指标

1. 选择节点监控仪表板
2. 从下拉菜单中选择目标节点
3. 选择时间范围：

- 最近30分钟
- 最近1/6/12/24小时
- 最近2/7天
- 自定义范围

3 理解指标

指标	描述
GPU利用率	使用的GPU计算能力百分比（0-100%）
GPU内存使用量	使用的总内存与可用内存（以GiB为单位）

容器组监控

通过精细过滤分析容器组级的GPU使用情况：

1 访问容器组指标

1. 导航到**GPU**目录下的仪表板
2. 选择容器组监控

2 配置过滤器

1. 选择GPU类型：

- pGPU
- GPU-Manager(vGPU)
- MPS

2. 选择包含GPU容器组的命名空间

3. 选择特定容器组

3 关键指标

指标	描述
容器组**GPU**利用率	被选定容器组的GPU计算使用情况
容器组**GPU**内存	被选定容器组的内存分配

时间范围选择

两个仪表板都支持灵活的时间窗口：

可用预设:
- 最近30分钟
- 最近1小时
- 最近6小时
- 最近12小时
- 最近24小时
- 最近2天
- 最近7天
- 自定义范围