Hardware accelerators

Overview

Install

Application Development

Configuration Management

Introduction

Guides

Resource Monitoring

Introduction

Guides

Overview

Introduction

Features

Introduction

TOC

Hardware accelerator Introduction

The Kubernetes Hardware accelerator Suite is an enterprise-grade solution for optimizing GPU resource allocation, isolation, and sharing in cloud-native environments. Built on Kubernetes device plugins and NVIDIA-native technologies, it provides three core modules:

1. **vGPU Module** Based on Opensource GPU-Manager, this enables fine-grained GPU virtualization by splitting physical GPUs into shareable virtual units with memory/compute quotas. Ideal for multi-tenant environments requiring dynamic resource allocation.

2. **pGPU Module** Leveraging NVIDIA's official Device Plugin, it delivers full physical GPU isolation with NUMA-aware scheduling. Designed for high-performance computing (HPC) workloads needing dedicated GPU access.

3. **MPS Module** Implements NVIDIA's Multi-Process Service to allow concurrent GPU context execution with resource constraints. Optimizes latency-sensitive applications through CUDA kernel fusion.

Product Advantages

vGPU Module

- **Dynamic Slicing**: Split GPUs to support multi progress used one physical gpu
- **QoS Enforcement**: Guaranteed compute units (vcuda-core) and memory quotas (vcuda-memory)

pGPU Module

- **Hardware-Level Isolation**: Direct PCIe passthrough with IOMMU protection
- **NUMA Optimization**: Minimize cross-socket data transfer via automatic NUMA node binding

MPS Module

- **Low-Latency Execution**: 30-50% latency reduction through CUDA context fusion
- **Resource Caps**: Limit per-process GPU compute (0-100%) and memory usage

- **Zero Code Changes**: Works with unmodified CUDA applications

Application Scenarios

vGPU Use Cases

- **Multi-Tenant AI Platforms**: Share A100/H100 GPUs across teams with guaranteed SLAs
- **VDI Environments**: Deliver GPU-accelerated virtual desktops for CAD/3D rendering
- **Batch Inference**: Parallelize model serving with fractional GPU allocations

pGPU Use Cases

- **HPC Clusters**: Run MPI jobs with exclusive GPU access for weather simulation
- **ML Training**: Full GPU utilization for large language model training
- **Medical Imaging**: Process high-resolution MRI data without resource contention

MPS Use Cases

- **Real-Time Inference**: Low-latency video analytics using concurrent CUDA streams
- **Microservice Orchestration**: Co-locate multiple GPU microservices on shared Hardware
- **High-Concurrency Serving**: Improve QPS by 3x for recommendation systems

Technical Limitations

Privileged Required

Hardware Device Access Requirements

Device File Permissions NVIDIA GPU devices require direct access to protected system resources:

```
# Device file ownership and permissions
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
```

- **Requirement**: Root access to read/write device files
- **Consequence**: Non-root containers get permission denied errors

Kernel-Level Operations

Essential NVIDIA Driver Interactions

Operation	Privilege Requirement	Purpose
Module Loading	CAP_SYS_MODULE	Load NVIDIA kernel modules
Memory Management	CAP_IPC_LOCK	GPU memory allocation
Interrupt Handling	CAP_SYS_RAWIO	Process GPU interrupts

K8s Device Plugin Architecture Requirements

1. **Socket Creation**: Write to `/var/lib/kubelet/device-plugins`
2. **Health Monitoring**: Access to `nvidia-smi` and kernel logs
3. **Resource Allocation**: Modify device cgroups

vGPU Constraints

- support only cuda less then 12.4
- No MIG support when vGPU enabled

pGPU Constraints

- No GPU sharing capability (1:1 pod-to-GPU mapping)

- Requires Kubernetes 1.25+ with SR-IOV enabled
- Limited to PCIe/NVSwitch-connected GPUs

MPS Constraints

- Potential fault propagation across fused contexts
- Requires CUDA 11.4+ for memory limits
- No support for MIG-sliced GPUs

Features

TOC

vGPU (Based on Opensource GPU-Manager)

- **Fine-Grained Resource Slicing**
  Splits physical GPUs core from 1-100 quotas. Supports dynamic allocation for multi-tenant environments like AI inference and virtual desktops.

- **Topology-Aware Scheduling**
  Automatically prioritizes NVLink/C2C-connected GPUs to minimize cross-socket data transfer latency. Ensures optimal GPU pairing for distributed training workloads.

pGPU (NVIDIA Device Plugin)

- **NUMA-Optimized Allocation**
  Enforces 1:1 GPU-to-Pod mapping with NUMA node binding, reducing PCIe bus contention for high-performance computing (HPC) tasks like LLM training.

- **Exclusive Hardware Access**
  Provides full physical GPU isolation through PCIe passthrough, ideal for mission-critical applications requiring deterministic performance (e.g., medical imaging processing).

MPS (NVIDIA Multi-Process Service Plugin)

- **Latency-Optimized Execution**
  Enables CUDA kernel fusion across processes, reducing inference latency by 30-50% for real-time applications like video analytics.

- **Resource Sharing with Caps**
  Allows concurrent GPU context execution while enforcing per-process compute (0-100%) and memory limits via environment variables.

Install

TOC

Installing Kubernetes Hardware accelerator Toolkit

Prerequisites

Before installation, ensure the following requirements are met:

1. **Kubernetes Cluster**: Version ≥1.25 with `DevicePlugins` feature gate enabled.
2. **NVIDIA Drivers**: Installed on all GPU nodes . Verify with `nvidia-smi`.
3. **Container Runtime**: Configured with NVIDIA Container Toolkit (≥1.7.0) for GPU support.

Installing via Web Console

1. **Navigate to Cluster Plugins**:

- Go to **Platform Management** → **Catalog** → **Cluster Plugin**
- Search for "gpu" and click **Install**

2. **Feature Toggles**: Enable/disable advanced capabilities during installation:

Option	Functionality	Recommended Scenario
PGPU	Physical GPU isolation with NUMA-aware scheduling	AI training/high-performance computing
vGPU	Virtual GPU slicing via GPU-Manager	Multi-tenant sharing/resource quotas
MPS	Multi-Process Service for compute/memory sharing	Low-latency inference/parallel tasks

Application Development

Introduction

Introduction

Application Development Introduction

Guides

CUDA Driver and Runtime Com

Hierarchical Architecture & Core Concepts

Version Compatibility Matrix & Constraints

Deployment Best Practices

Troubleshooting Handbook

Add Custom Devices Using ConfigMap

Introduction

Features

Advantages

Function Module 1: ConfigMap Authoring Specifications

Function Module 2: Resource Value Definition

Troubleshooting

Introduction

TOC

Application Development Introduction

Application Development Module guides users in configuring Hardware accelerators from multiple vendors (e.g., AMD/Intel GPUs, FPGAs) through a unified interface, enabling the orchestration and optimization of heterogeneous computing resources in containerized environments to enhance high-performance workloads like AI training and image processing.

Guides

CUDA Driver and Runtime Compatibility

TOC

Hierarchical Architecture & Core Concepts

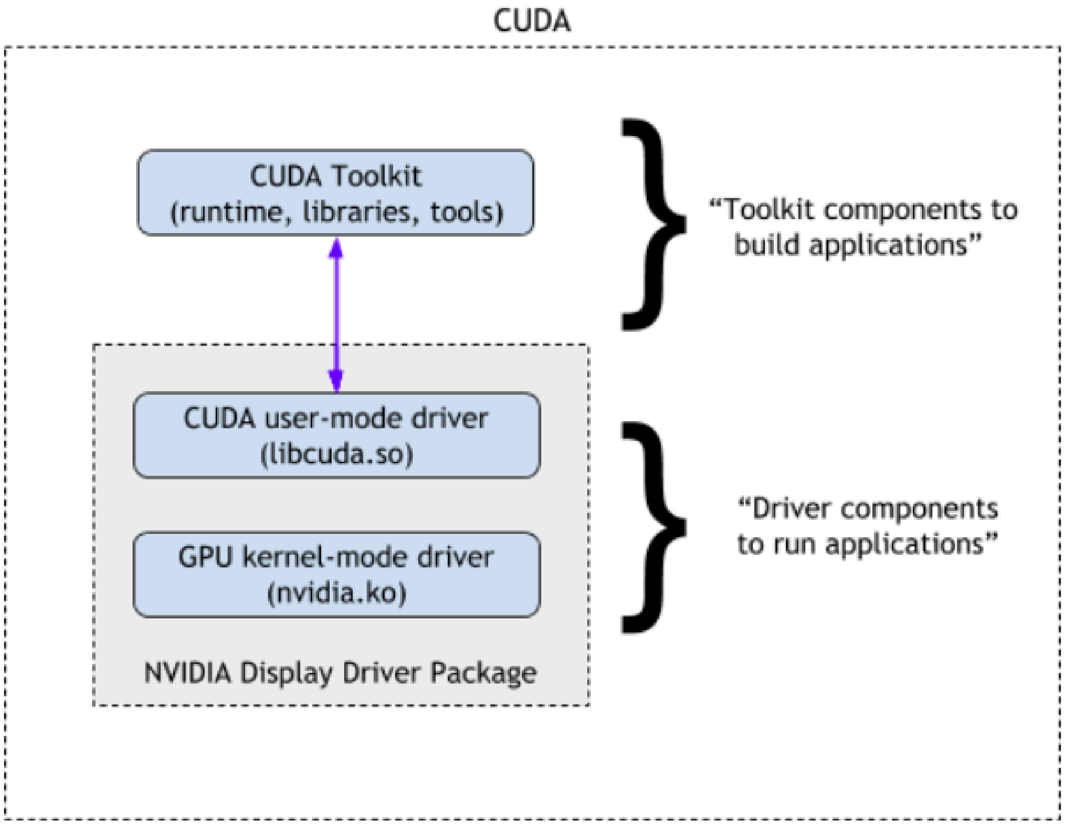
CUDA
CUDA Toolkit (runtime, libraries, tools)
"Toolkit components to build applications"
CUDA user-mode driver (libcuda.so)
GPU kernel-mode driver (nvidia.ko)
NVIDIA Display Driver Package
"Driver components to run applications"

Figure 1: Components of CUDA #

1. CUDA Runtime API Layer

Technical Positioning

1. **Functional Scope**: Provides high-level abstraction interfaces for developers, encapsulating core GPU operations (memory allocation, stream management, kernel launches, etc.)
2. **Version Binding**: Determined by the CUDA Toolkit version used during build (e.g., CUDA 12.0.1)

Version Detection Methods

```
# Python environment detection (recommended priority method)
pip list | grep cuda
conda list |grep cuda
# Example output: cu121  # cu121 indicates CUDA 12.1 environment

# System-level runtime library detection
find / -name "libcudart*"
# cudart means cuda runtime
# Example output:
/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
Indicates CUDA 12.4
```

if you find multi lib version, you should check your program which version used,like PATH, LD_LIBRARY_PATH or other program set

```
env |grep PATH
# Example output:
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/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
#that means your cuda program used the first lib by lib path order
```

2. CUDA Driver API Layer

Technical Positioning

1. **Functional Scope**: Low-level interface directly interacting with GPU hardware, handling instruction translation and hardware resource scheduling
2. **Version Binding**: Determined by NVIDIA driver version, following SemVer specification

Version Detection Methods

```
nvidia-smi
# Example output:
+------------------------------------------------------------------------
-----------------+
| NVIDIA-SMI 550.144.03             Driver Version: 550.144.03     CUDA V
ersion: 12.4     |
|-----------------------------------------+------------------------+-----
-----------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Vola
tile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-
Util  Compute M. |
|                                         |                        |
MIG M. |
|=========================================+========================+=====
=================|
|   0  NVIDIA A30                     Off |   00000000:00:0B.0 Off |
0 |
| N/A   31C    P0             28W /  165W |   10195MiB /  24576MiB |
0%      Default |
|                                         |                        |
Disabled |
+-----------------------------------------+------------------------+-----
-----------------+
```

Version Compatibility Matrix & Constraints

Physical GPU Deployment - Core Compatibility Principles

First reference NVIDIA's official statement,the **base constraints** is

1. **Driver version must always be ≥ Runtime version**
2. NVIDIA officially guarantees **1 major version backward compatibility** (e.g., CUDA Driver 12.x supports Runtime 11.x)
3. Cross-two-major-version compatibility (e.g., Driver 12.x with Runtime 10.x) is neither officially supported nor recommended

when you deploy cuda program，please comply with the base constraints

Formal Rules

```
+ Mandatory: Driver version ≥ Runtime version
+ Recommended: Driver major version - Runtime major version ≤ 1
- Blocked: Driver version < Runtime version → May trigger CUDA_ERROR_UNKNOWN(999)
- Unstable: Driver major version - Runtime major version > 1 → Application may malfunction
```

Virtualization Scenario Enhancements (HAMI/GPU-Manager)

When using Virtual GPU solutions like GPU-Manager or HAMI, **besides the base constraints up**, you must comply with the **additional constraints** apply:

Version Requirements

```
1. Virtual GPU solutions baseline version ≥ Runtime version
2. Runtime major version = Driver major version = Baseline major version
```

Special Note for GPU-Manager: We implemented partial cross-1-major-version compatibility (e.g., baseline 12.4 supporting vLLM 11.8). However, this requires per-application hook adjustments and must be analyzed case-by-case.

Deployment Best Practices

Recommended Strategy

- Adopt newer CUDA versions (e.g., CUDA 12.x) for both Driver and Runtime in new GPU cluster planning

Alternative Solutions for Legacy Systems

1. Physical GPU Scheduling or GPU-Manager Whole-Card Allocation

Whole-card scheduling provides native compatibility equivalent to physical GPU access GPU-Manager can use whole card mode when you set tencent.com/vcuda-core to 100 multily positive integer,like 100,200,300

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

2. Node Labeling Strategy

Label nodes based on supported Driver CUDA versions:

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

this means your node is cuda 12.4

Configure scheduling affinity in deployments: you can set cuda-major-version and cuda-minor-version by your program cuda runtime need

```yaml
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. Runtime Version Upgrade

Legacy CUDA Runtimes may have security vulnerabilities (CVEs) and lack support for new GPU features. Prioritize upgrades to CUDA 12.x.

nvidia recomend to upgrade both

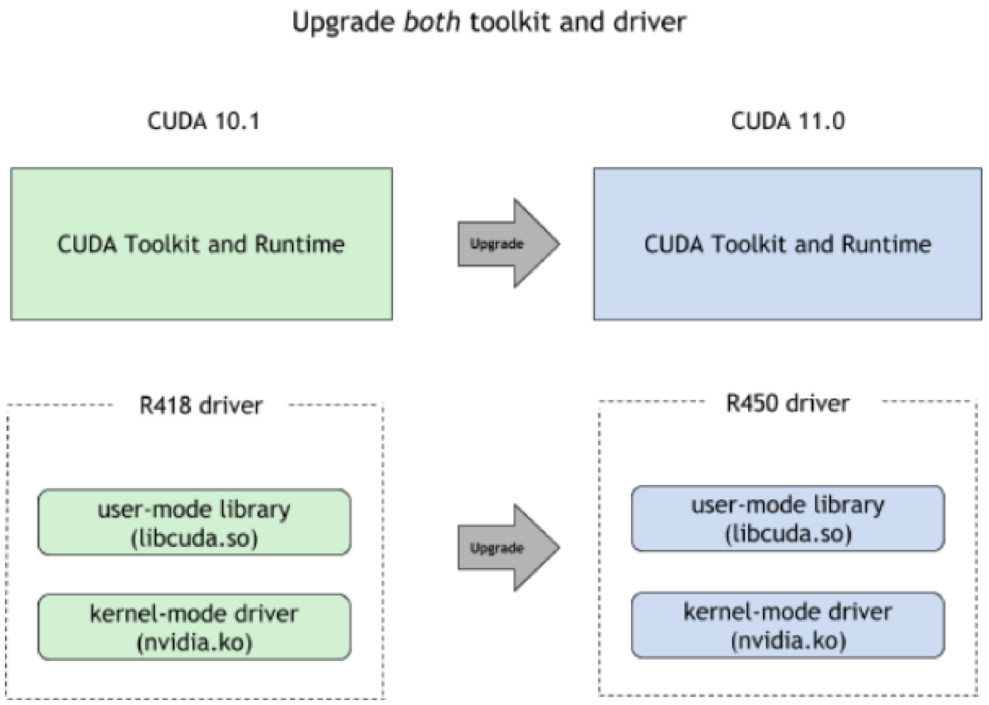
Upgrade *both* toolkit and driver
CUDA 10.1
CUDA 11.0
CUDA Toolkit and Runtime
Upgrade
CUDA Toolkit and Runtime
R418 driver
R450 driver
user-mode library (libcuda.so)
kernel-mode driver (nvidia.ko)
Upgrade
user-mode library (libcuda.so)
kernel-mode driver (nvidia.ko)

Figure 2: CUDA Upgrade Path

Troubleshooting Handbook

Common Error Codes

Error Code	**Description**	**Recommended Action**
CUDA_ERROR_INVALID_IMAGE	Driver-Runtime incompatibility	Align driver version with container CUDA version
CUDA_ERROR_ILLEGAL_ADDRESS	Virtual memory violation (common in version mismatch)	Verify Runtime vs baseline constraints
CUDA_ERROR_UNSUPPORTED_PTX_VERSION	PTX instruction set mismatch	Recompile with explicit -arch=sm_xx

Add Custom Devices Using ConfigMap

TOC

Introduction

- Implements standardized definition and management of Kubernetes custom resources through ConfigMap, addressing:
- Unified management of custom resource specifications to prevent configuration fragmentation
- Standardized resource definition format for better maintainability
- Multi-language description support and default value configuration

- Suitable for scenarios requiring Kubernetes resource model extension (e.g., GPU resource management), providing a standardized resource definition framework

Features

- Single-key resource definition specification
- Multi-key associated resource definition
- Standardized resource request interface
- Chinese/English bilingual description support
- Resource default value configuration mechanism

Advantages

- **Extensibility**: Resource group management through labels
- **Security**: Namespace isolation (kube-public)
- **Stability**: Enforced format validation rules
- **Maintainability**: Unified metadata label specifications

Function Module 1: ConfigMap Authoring Specifications

Core Rules

1. **Single Responsibility Principle**: One ConfigMap per key definition

2. **Namespace**: Fixed to `namespace=kube-public`

3. **Naming Convention**:

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

- `cf-crl`: Fixed prefix
- `customName`: Custom valid name
- `keyName`: Key identifier (special characters replaced with '-')

4. **Label Requirements**:

```
labels:
  features.alauda.io/type: CustomResourceLimitation  # Fixed value
  features.alauda.io/group: {resource-group}         # e.g., gpu-manager
  features.alauda.io/enabled: "true"               # Activation flag
```

Parameter Specification

Parameter	Required	Description
name format	Yes	Follows cf-crl-{customName}-{keyName}
namespace	Yes	Fixed as kube-public
label group	Yes	Must contain specified 3 feature labels

Function Module 2: Resource Value Definition

Single Key Example

```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"       # Resource key
  dataType: "integer"                # Value type
  defaultValue: "20"                 # Default value
  descriptionZh: "" # Chinese description
  descriptionEn: "GPU vcore count, 100 virtual cores equal 1 physical GPU core" # English description
  group: "gpu-manager"               # Resource group
  limits: "optional"                 # Limits field policy
  requests: "disabled"               # Requests field policy
```

Multi-key Association

```yaml
metadata:
  name: cf-crl-gpu-manager-vcuda-core
  labels: [same group labels]

metadata:
  name: cf-crl-gpu-manager-vcuda-memory
  labels: [same group labels]  # Association through identical labels
```

Policy Specification

Field	Allowed Values	Description
limits	disabled/required/optional	Resource limits configuration
requests	disabled/required/fromLimits	Resource requests configuration

Troubleshooting

Troubleshooting float16 is only supported on GPUs with compute capability at least xx Error in vLLM

TOC

Problem Description

Environment

- **Hardware**: NVIDIA GPUs with compute capability <8.0 (e.g., Tesla V100, T4)
- **Model Types**: LLMs requiring bfloat16/FP8 precision (e.g., LLaMA-2-70B, GPT-NeoX-20B)

Symptoms

1. Explicit error message:

   ```
   ValueError: float16/bfloat16 is only supported on GPUs with compute capability at least 8.0
   ```

2. Failed kernel compilation during model loading

Related Logs

```
# vLLM error stack trace
File "/usr/local/lib/python3.10/site-packages/vllm/model_executor/layers/quantization/__init__.py", line 37, in _verify_cuda_compute_capability
    raise ValueError(
ValueError: bfloat16 is only supported on GPUs with compute capability at least 8.0. Current GPU: Tesla V100-PCIE-16GB, compute capability 7.0
```

Root Cause

Primary Cause

Insufficient GPU Compute Capability The GPU's compute capability (CC) doesn't meet the minimum requirement for specific data types:

- **bfloat16/FP8**: Requires CC ≥8.0 (Ampere architecture or newer)
- **FP16 Tensor Core Optimization**: Requires CC \≥7.0 (Volta architecture or newer)

Technical Analysis

1. **Architecture Limitations**:
   - Pre-Ampere GPUs (CC <8.0) lack dedicated matrix math units for bfloat16 operations
   - Tensor Cores in Volta/Turing (CC 7.0-7.5) only support FP16/FP32 mixed precision
2. **Framework Enforcement**:

   ```
   # vLLM's capability check (simplified)
   def _verify_cuda_compute_capability():
       if device.compute_capability < MIN_REQUIRED_CC:
           raise ValueError(f"Requires compute capability ≥{MIN_REQUIRED_CC}")
   ```

Troubleshooting

Step 1: Verify GPU Compute Capability

```
import torch
print(f"Compute Capability: {torch.cuda.get_device_capability()}")
```

Step 2: Check Model Precision Requirements

```
cat model/config.json | grep "torch_dtype"
# Expected output: "bfloat16" or "float16"
```

Step 3: Validate Framework Compatibility

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

Solution

Solution for Insufficient Compute Capability

Considerations

- Performance degradation expected when downgrading precision
- Model accuracy may vary with different precision types

Prerequisites

- CUDA Toolkit ≥11.8

Steps

1. **Modify InferenceService yaml**: add args like --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 # Force FP16 precision
          - --tensor-parallel-size=1
        resources:
          limits:
            nvidia.com/gpu: '1'
```

2. **Wait deploy restart**

Preventive Measures

1. **Pre-Flight Checks**:

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

2. **Cluster Configuration**:

```bash
# NVIDIA device plugin config
helm upgrade -i nvidia-device-plugin \
  --set compatabilityPolicy=strict \
  --set computeCapabilities=8.0+
```

3. **Model Optimization**:

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

Related Content

GPU Compute Capability Reference

Architecture	CC Range	Supported Precisions
Volta	7.0-7.2	FP16 Tensor Core
Turing	7.5	FP16/INT8
Ampere	8.0-8.9	bfloat16/TF32/FP8
Hopper	9.0+	FP4/FP8 with dynamic scale

Official References

1. NVIDIA Compute Capability Table
2. vLLM Hardware Requirements

Paddle Autogrow Memory Allocation Crash on GPU-Manager

TOC

Problem Description

Symptoms

When both PaddlePaddle's Autogrow memory allocation strategy and GPU-Manager's virtualized memory management are enabled simultaneously, the following anomalies may occur:

1. OOM errors due to discontinuous memory allocation
2. Abnormal GPU utilization fluctuations
3. Random training process crashes
4. Inconsistent memory usage between nvidia-smi reports and framework statistics

Root Cause

Root Cause Analysis

1. **Memory Allocation Strategy Conflict** Paddle's Autogrow uses dynamic segmented allocation while GPU-Manager's virtualization requires contiguous physical memory mapping

2. **Management Mechanism Incompatibility** Autogrow's delayed release mechanism conflicts with GPU-Manager's memory reclamation strategy

3. **Metadata Maintenance Conflict** Separate metadata maintenance by both systems causes inconsistent memory views

Trigger Mechanism:

- Autogrow attempts optimal block sizing during allocation
- GPU-Manager virtualization layer intercepts physical memory requests
- Non-contiguous allocations cause virtual address mapping failures
- Dual management leads to metadata consistency exceptions

Solution

Solution Overview

Force Paddle to use traditional allocation strategy via environment variable:

```
FLAGS_allocator_strategy=naive_best_fit
```

Considerations

1. Requires training process restart
2. May reduce Paddle's memory reuse efficiency

Implementation Steps

Kubernetes Deployment

1. Edit Deployment configuration

```yaml
apiVersion: apps/v1
kind: Deployment
spec:
  template:
    spec:
      containers:
        - name: paddle-container
          env:
            - name: FLAGS_allocator_strategy
              value: 'naive_best_fit'
```

2. Apply configuration

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

3. Verify configuration

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

Bare Metal Deployment

1. Set environment variable before execution

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

2. Or set in Python code

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

Verification Methods

1. Check allocation strategy confirmation in logs

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

2. Monitor memory allocation continuity

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

3. Stress test validation

```
# Continuous allocation test script
import paddle
for i in range(10):
    data = paddle.randn([1024, 1024, 100], dtype='float32')
    print(f"Allocated {i+1}GB")
```

Preventive Measures

1. **Version Compatibility Check** Review Paddle release notes for memory allocation changes during upgrades

2. **Monitoring Configuration** Add Prometheus alert rule:

```
• alert: GPUAllocConflict
  expr: rate(paddle_gpu_malloc_failed_total[5m]) > 0
  labels:
    severity: critical
  annotations:
    summary: "GPU Memory Allocation Conflict Alert"
```

3. **Baseline Testing** Perform memory allocation baseline tests for new environments:

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

Related Content

Memory Allocation Strategy Comparison

Strategy	Advantages	Disadvantages
autogrow	High efficiency	Poor large-block perf
naive_best_fit	Stable allocation	Potential fragmentation

References

Paddle Memory Optimization Whitepaper ↗

Configuration Management

Introduction

Guides

Introduction

TOC

Configuration Management Introduction

Configuration Management is the centralized documentation portal for configuring GPU acceleration capabilities in kubernetes environments. This living document provides administrators with unified guidance for setting up physical GPU (pGPU), virtual GPU (vGPU), and Multi-Process Service (MPS) configurations across hybrid infrastructure.

Guides

Configure Hardware accelerator on GPU nodes

As the amount of business data increases, especially for scenarios such as artificial intelligence and data analysis, you may want to use GPU capabilities in your self-built business cluster to accelerate data processing. In addition to preparing GPU resources for cluster nodes, GPU configuration should also be performed.

This solution refers to nodes in the cluster that have GPU computing capabilities as **GPU Nodes**.

Note: Unless otherwise specified, the operation steps will apply to both types of nodes. For driver installation related issues, refer to the NVIDIA official installation documentation ↗.

TOC

Prerequisites

GPU resources have been prepared on the operating node, which belongs to the GPU node mentioned in this section.

Install GPU driver

Notice: If the GPU node uses the NVIDIA MPS plugin, ensure that the GPU architecture of the node is Volta or newer (Volta/Turing/Ampere/Hopper, etc.), and the driver supports CUDA version 11.5 or higher.

Gets the driver download address

1. Log in to the GPU node and run the command `lspci |grep -i NVIDIA` to check the GPU model of the node.

   In the following example, the GPU model is Tesla T4.

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

2. Go to the NVIDIA official website ↗ to obtain the driver download link.

   1. Click on **Drivers** in the top navigation bar on the homepage.

   2. Fill in the required information for downloading the driver according to the GPU node model .

   3. Click on **Search**.

   4. Click on **Download**.

   5. Right-click on **Download** > **Copy Link Address** to copy the download link of the driver.

3. Execute the following command lines on the GPU node in order to create the `/home/gpu` directory and download and save the driver file to this directory.

   ```
   # Create /home/gpu Directory
   mkdir -p /home/gpu
   cd /home/gpu/
   # Download the driver file to /home/gpu Directory, Example：wget https://cn.download.nvidia.com/tesla/515.65.01/NVIDIA-Linux-x86_64-515.65.01.run
   wget <Driver download address>
   # Verify that the driver file has been downloaded successfully, If the driver file name is returned（For example：NVIDIA-Linux-x86_64-515.65.01.run）Indicates that the download was successful
   ls <Driver file name>
   ```

Installation driver

1. Execute the following command on the GPU node to install the gcc and kernel-devel packages corresponding to the current operating system.

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

2. Execute the following commands in order to install the GPU driver.

   ```
   chmod a+x /home/gpu/<Driver file name>
   /home/gpu/<Driver file name> --dkms
   ```

3. After installation, execute the `nvidia-smi` command. If GPU information similar to the following example is returned, it indicates that the driver installation was successful.

```
# nvidia-smi
Tue Sep 13 01:31:33 2022
+----------------------------------------------------------------------
-------+
| NVIDIA-SMI 515.65.01    Driver Version: 515.65.01    CUDA Version: 1
1.7     |
+-------------------------------+----------------------+---------------
-------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncor
r. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Comp
ute M. |
|                               |                      |
MIG M. |
|===============================+======================+===============
=======|
|   0  Tesla T4            Off  | 00000000:00:08.0 Off |
0 |
| N/A   55C    P0    28W /  70W |      2MiB / 15360MiB |      5%      D
efault |
|                               |                      |
N/A |
+-------------------------------+----------------------+---------------
-------+

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

Installation the NVIDIA Container runtime

1. On the **GPU Node**, add the NVIDIA yum repository.

   ```
   distribution=$(. /etc/os-release;echo $ID$VERSION_ID) && curl -s -L https://nvidia.github.io/libnvidia-container/$distribution/libnvidia-container.repo | sudo tee /etc/yum.repos.d/nvidia-container-toolkit.repo
   yum makecache -y
   ```

   When the prompt "Metadata cache created." appears, it indicates that the addition is successful.

2. Install NVIDIA Container Runtime.

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

   When the prompt `Complete!` appears, it means the installation is successful.

3. Config the default Runtime. Add the following configuration to the file.

   - Containerd: Modify the `/etc/containerd/config.toml` file.

```
[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.options]
              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.nvidia.options]
              BinaryName = "/usr/bin/nvidia-container-runtime"
              SystemdCgroup = true
...
```

- Docker: Modify the `/etc/docker/daemon.json` file.

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

4. Restart Containerd / Docker.

   - Containerd

   ```
   systemctl restart containerd   #Restart

   crictl info |grep Runtime  #Check
   ```

   - Docker

   ```
   systemctl restart docker   #Restart

   docker info |grep Runtime  #Check
   ```

Physical GPU configuration

Deploy physical GPU plugin on a GPU Business Cluster

On the management interface of the GPU cluster, perform the following actions:

1. In the Catalog leftsidebar, choose "Cluster Plugins" subsidebar, click to deploy the "ACP GPU Device Plugin" and open the "pGPU" option;

2. In the "Nodes" tab, select the nodes that need to deploy the physical GPU, then click on "Label and Taint Manager", add a "device label" and choose "pGPU", and click OK;

3. In the "Pods" tab, check the running status of the container group corresponding to nvidia-device-plugin-ds to see if there are any abnormalities and ensure it is running on the specified nodes.

NVIDIA MPS configuration (driver support cuda version must >= 11.5)

Deploy NVIDIA MPS plugin on a GPU Business Cluster

On the management interface of the GPU cluster, perform the following actions:

1. In the Catalog leftsidebar, choose "Cluster Plugins" subsidebar, click to deploy the "ACP GPU Device Plugin" and open the "MPS" option;

2. In the "Nodes" tab, select the nodes that need to deploy the physical GPU, then click on "Label and Taint Manager", add a "device label" and choose "MPS", and click OK;

3. In the "Pods" tab, check the running status of the container group corresponding to nvidia-mps-device-plugin-daemonset to see if there are any abnormalities and ensure it is running on the specified nodes.

Configure kube-scheduler (kubernetes> = 1.23)

1. On the **Business Cluster Control Node**, check if the scheduler correctly references the scheduling policy.

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

   check if has –config option and value is /etc/kubernetes/scheduler-config.yaml, like

```
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
```

Note: The above parameters and values are the default configurations of the platform. If you have modified them, please change them back to the default values. The original custom configurations can be copied to the scheduling policy file.

2. Check the configuration of the scheduling policy file.

   1. Execute the command: `kubectl describe service kubernetes -n default |grep Endpoints`.

      ```
      Expected effectEndpoints:         192.168.130.240:6443
      ```

   2. Replace the contents of the `/etc/kubernetes/scheduler-config.yaml` file on all Master nodes with the following content, where `${kube-apiserver}` should be replaced with the output of the first step.

```
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/services/nvidia-mps-scheduler-plugin/proxy/scheduler
  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
```

if schedule-config.yaml already exist extenders,then append yaml to the end

```
- enableHTTPS: true
  filterVerb: predicates
  managedResources:
  - ignoredByScheduler: false
    name: nvidia.com/mps-core
  nodeCacheCapable: false
  urlPrefix: https://${kube-apiserver}/api/v1/namespaces/kube-system/services/nvidia-mps-scheduler-plugin/proxy/scheduler
  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. Run the following command to obtain the container ID:

- Containerd: Execute `crictl ps |grep kube-scheduler`, the output is as follows, with the first column being the container ID.

```
1d113ccf1c1a9       03c72176d0f15       2 seconds ago       Running
kube-scheduler              3                   ecd054bbdd465       k
ube-scheduler-192.168.176.47
```

- Docker: Run `docker ps |grep kube-scheduler`, the output is as follows, with the first column being the container ID.

```
30528a45a118   d8a9fef7349c    "kube-scheduler --au..."   37 minutes
ago   Up 37 minutes     k8s_kube-scheduler_kube-scheduler-192.168.13
0.240_kube-system_3e9f7007b38f4deb6ffd1c7587621009_28
```

4. Restart the Containerd/Docker container using the container ID obtained in the previous step.

- Containerd

```
crictl stop <Container ID>
```

5. Restart Kubelet.

```
systemctl restart kubelet
```

GPU-Manager configuration

Configure kube-scheduler (kubernetes> = 1.23)

1. On the **Business Cluster Control Node**, check if the scheduler correctly references the scheduling policy.

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

check if has –config option and value is /etc/kubernetes/scheduler-config.yaml, like

```
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
```

Note: The above parameters and values are the default configurations of the platform. If you have modified them, please change them back to the default values. The original custom configurations can be copied to the scheduling policy file.

2. Check the configuration of the scheduling policy file.

   1. Execute the command: `kubectl describe service kubernetes -n default |grep Endpoints`.

      ```
      Expected effectEndpoints:         192.168.130.240:6443
      ```

   2. Replace the contents of the `/etc/kubernetes/scheduler-config.yaml` file on all Master nodes with the following content, where `${kube-apiserver}` should be replaced with the output of the first step.

```
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/
services/gpu-quota-admission/proxy/scheduler
  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. Run the following command to obtain the container ID:

- Containerd: Execute `crictl ps |grep kube-scheduler`, the output is as follows, with the first column being the container ID.

```
1d113ccf1c1a9       03c72176d0f15       2 seconds ago       Running
kube-scheduler              3                   ecd054bbdd465       k
ube-scheduler-192.168.176.47
```

- Docker: Run `docker ps |grep kube-scheduler`, the output is as follows, with the first column being the container ID.

```
30528a45a118   d8a9fef7349c    "kube-scheduler --au..."   37 minutes
ago   Up 37 minutes     k8s_kube-scheduler_kube-scheduler-192.168.13
0.240_kube-system_3e9f7007b38f4deb6ffd1c7587621009_28
```

4. Restart the Containerd/Docker container using the container ID obtained in the previous step.

- Containerd

```
crictl stop <Container ID>
```

5. Restart Kubelet.

```
systemctl restart kubelet
```

Deploy GPU Manager plugin on a GPU Business Cluster

On the management interface of the GPU cluster, perform the following actions:

1. In the Catalog leftsidebar, choose "Cluster Plugins" subsidebar, click to deploy the "ACP GPU Device Plugin" and open the "GPU-Manager" option;
2. In the "Nodes" tab, select the nodes that need to deploy the physical GPU, then click on "Label and Taint Manager", add a "device label" and choose "vGPU", and click OK;
3. In the "Pods" tab, check the running status of the container group corresponding to gpu-manager-daemonset to see if there are any abnormalities and ensure it is running on the specified nodes.

Validation of results

Method 1: Check if there are available GPU resources on the GPU nodes by running the following command on the control node of the business cluster:

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

Method 2: Deploy a GPU application on the platform by specifying the required amount of GPU resources. After deployment, exec the Pod and execute the following command:.

```
# 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  Comput
e M. |
|                               |                      |               MI
G M. |
|===============================+======================+================
=====|
|   0  Tesla T4            Off  | 00000000:00:08.0 Off |
0 |
| N/A   55C    P0    28W /  70W |      2MiB / 15360MiB |      5%      Def
ault |
|                               |                      |
N/A |
+-------------------------------+----------------------+----------------
-----+

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

Check whether the correct GPU information is retrieved.

Resource Monitoring

Introduction

Introduction

Guides

GPU Resource Monitoring

Introduction

TOC

Resource Monitoring Introduction

Resource Monitoring is a critical component of the Kubernetes Hardware Accelerator Suite, designed to provide comprehensive visibility into GPU resource utilization across your containerized workloads. This module delivers real-time metrics and historical data analysis for both **compute utilization** and **GPU memory consumption** at two fundamental levels:

Resource Monitoring is a critical component of the Kubernetes Hardware Accelerator Suite, designed to provide comprehensive visibility into GPU resource utilization across your containerized workloads. This module delivers both **compute utilization** and **GPU memory consumption** at two fundamental levels:

- **Node-Level Monitoring**: Track aggregate GPU resource usage across entire Kubernetes nodes
- **Pod-Level Monitoring**: Analyze per-workload GPU consumption with pod granularity

Integrated with the platform's core accelerator modules (pGPU/vGPU(GPU-Manager)/MPS), this monitoring solution enables users to optimize GPU allocation, enforce resource quotas, and troubleshoot performance bottlenecks in AI/ML workloads, real-time inference services, etc.

Advantages

The core advantages of Resource Monitoring are as follows:

- **Multi-Dimensional Observability**

  Simultaneously monitor both compute units (CUDA cores) and memory utilization across physical/virtual GPUs, providing holistic insights into accelerator usage patterns.

- **Hierarchical Metrics Collection**

  Capture data at both node and pod granularity, enabling correlation between cluster-wide resource trends and individual workload demands.

- **Native Integration**

  Seamlessly works with all accelerator modules (pGPU/vGPU/MPS) without requiring additional agents, leveraging Kubernetes-native metrics pipelines.

- **Historical Analysis**

  Store GPU metrics with configurable retention periods (default 7 days) for capacity planning and usage pattern analysis through integrated visualization tools.

Application Scenarios

The main application scenarios for Resource Monitoring are as follows:

- **Performance Optimization**

  Identify underutilized GPUs in training clusters and right-size resource requests for deep learning workloads. For example, detect pods consistently using <30% of allocated GPU memory to optimize memory allocations.

- **Multi-Tenant Governance**

  Enforce GPU quota compliance in shared environments by monitoring vGPU consumption across teams. Track cumulative usage against allocated quotas in AI platform deployments.

- **Cost Attribution**

  Generate per-namespace GPU utilization reports for chargeback/showback models in enterprise Kubernetes environments, correlating pod-level metrics with organizational units.

- **Fault Diagnosis**

  Investigate OOM (Out-of-Memory) incidents in GPU-accelerated workloads by analyzing memory usage trends preceding container crashes. Cross-reference with Kubernetes events for root cause analysis.

- **Capacity Planning**

  Analyze historical GPU utilization patterns (e.g., peak compute demand periods) to inform infrastructure scaling decisions and budget allocations for AI infrastructure.

Usage Limitations

When using Resource Monitoring, please note the following constraints:

- **Module Dependencies**

  - Requires at least one accelerator module (pGPU/vGPU/MPS) to be deployed in the cluster

Guides

GPU Resource Monitoring

GPU Resource Monitoring

TOC

Feature Overview

The Resource Monitoring feature enables real-time and historical tracking of GPU utilization and memory usage across nodes and pods within the Container Platform. This functionality helps administrators and developers:

- **Monitor GPU Performance**: Identify bottlenecks in GPU resource allocation.

- **Troubleshoot Issues**: Analyze GPU usage trends for debugging resource-related problems.
- **Optimize Workloads**: Make data-driven decisions to improve workload distribution.

Applicable Scenarios:

- Real-time monitoring of GPU-intensive applications.
- Historical analysis of GPU utilization for capacity planning.
- Multi-node/pod GPU performance comparison.

Value Delivered:

- Enhanced visibility into GPU resource consumption.
- Improved cluster efficiency through actionable insights.

Core Features

- **Node-Level Monitoring**: Track GPU utilization and memory usage per node.
- **Pod-Level Monitoring**: Monitor GPU metrics for individual pods.
- **Custom Time Ranges**: Analyze data from 30 minutes up to 7 days.

Feature Advantages

- **Real-Time Visualization**: Interactive dashboards with auto-refresh capabilities.
- **Multi-Dimensional Filtering**: Narrow down metrics by GPU type, namespace, or pod.

Node Monitoring

Monitor GPU resources at the node level through these steps:

1 Access GPU Dashboards

1. Navigate to **Platform Management** view

2. Go to **Operations Center** → **Monitoring** → **Dashboards**
3. Switch to the **GPU** directory

2 Select Node Metrics

1. Choose **Node Monitoring** dashboard
2. Select target node from dropdown
3. Pick time range:
   - Last 30 minutes
   - Last 1/6/12/24 hours
   - Last 2/7 days
   - Custom range

3 Interpret Metrics

Metric	Description
GPU Utilization	Percentage of GPU computing capacity used (0-100%)
GPU Memory Usage	Total memory consumed vs. available memory (in GiB)

Pod Monitoring

Analyze GPU usage at the pod level with granular filtering:

1 Access Pod Metrics

1. Navigate to **GPU** directory dashboards
2. Choose **Pod Monitoring**

2 Configure Filters

1. Select GPU type:

- pGPU
- GPU-Manager(vGPU)
- MPS

2. Choose namespace containing GPU pods
3. Select specific pod

3 Key Metrics

Metric	Description
Pod GPU Utilization	GPU compute usage by selected pod
Pod GPU Memory	Memory allocation for selected pod

Time Range Selection

Both dashboards support flexible time windows:

```
Available Presets:
- Last 30 minutes
- Last 1 hour
- Last 6 hours
- Last 12 hours
- Last 24 hours
- Last 2 days
- Last 7 days
- Custom range
```