Menu

Аппаратные ускорители

Обзор

Введение

Введение в аппаратный акселератор

Преимущества продукта

Сценарии применения

Технические ограничения

Возможности

vGPU (на основе Opensource GPU-Manager)

pGPU (NVIDIA Device Plugin)

MPS (NVIDIA Multi-Process Service Plugin)

Установка

Установка

Установка Kubernetes Hardware accelerator Toolkit

Разработка приложений

Введение

Введение в разработку приложений

Руководства

Устранение неполадок

Управление конфигурацией

Введение

Введение в управление конфигурациями

Руководства

Мониторинг ресурсов

Введение

Введение в мониторинг ресурсов

Преимущества

Сценарии применения

Ограничения использования

Руководства

Menu

Обзор

Введение

Введение в аппаратный акселератор

Преимущества продукта

Сценарии применения

Технические ограничения

Возможности

vGPU (на основе Opensource GPU-Manager)

pGPU (NVIDIA Device Plugin)

MPS (NVIDIA Multi-Process Service Plugin)

Введение

Содержание

Введение в аппаратный акселератор

Набор аппаратных акселераторов Kubernetes — это корпоративное решение для оптимизации распределения, изоляции и совместного использования ресурсов GPU в

облачных нативных средах. Основанный на device plugins Kubernetes и технологиях NVIDIA, он включает три основных модуля:

1. **vGPU Module**
   Основанный на Opensource GPU-Manager, этот модуль обеспечивает тонкую виртуализацию GPU, разделяя физические GPU на виртуальные единицы с квотами по памяти и вычислительным ресурсам. Идеально подходит для мультиарендных сред с динамическим распределением ресурсов.

2. **pGPU Module**
   Используя официальный Device Plugin NVIDIA, обеспечивает полную изоляцию физического GPU с планированием, учитывающим NUMA. Предназначен для высокопроизводительных вычислительных задач (HPC), требующих выделенного доступа к GPU.

3. **MPS Module**
   Реализует Multi-Process Service NVIDIA для одновременного выполнения нескольких контекстов GPU с ограничениями ресурсов. Оптимизирует приложения с чувствительностью к задержкам за счёт слияния CUDA-ядр.

Преимущества продукта

vGPU Module

- **Динамическое разделение**: Разделение GPU для поддержки нескольких процессов на одном физическом GPU
- **Обеспечение QoS**: Гарантированные вычислительные единицы (vcuda-core) и квоты памяти (vcuda-memory)

pGPU Module

- **Изоляция на уровне аппаратуры**: Прямой PCIe passthrough с защитой IOMMU
- **Оптимизация NUMA**: Минимизация передачи данных между сокетами за счёт автоматического привязывания к NUMA-узлам

MPS Module

- **Низкая задержка выполнения**: Снижение задержек на 30-50% благодаря слиянию CUDA-контекстов
- **Ограничения ресурсов**: Лимитирование вычислительных ресурсов GPU (0-100%) и использования памяти на процесс
- **Отсутствие изменений в коде**: Работает с немодифицированными CUDA-приложениями

Сценарии применения

Случаи использования vGPU

- **Мультиарендные AI-платформы**: Совместное использование GPU A100/H100 между командами с гарантированными SLA
- **VDI-среды**: Предоставление виртуальных рабочих столов с ускорением GPU для CAD/3D рендеринга
- **Пакетный вывод**: Параллельное обслуживание моделей с дробным распределением GPU

Случаи использования pGPU

- **HPC-кластеры**: Запуск MPI-задач с эксклюзивным доступом к GPU для моделирования погоды
- **Обучение ML**: Полное использование GPU для обучения больших языковых моделей
- **Медицинская визуализация**: Обработка высокоразрешённых данных МРТ без конкуренции за ресурсы

Случаи использования MPS

- **Инференс в реальном времени**: Аналитика видео с низкой задержкой с использованием параллельных CUDA-потоков

- **Оркестрация микросервисов**: Совмещение нескольких GPU-микросервисов на общем оборудовании
- **Обслуживание с высокой конкуренцией**: Увеличение QPS в 3 раза для рекомендательных систем

Технические ограничения

Требуется привилегированный доступ

Требования к доступу к аппаратным устройствам

Права доступа к устройствам
Устройства NVIDIA GPU требуют прямого доступа к защищённым системным ресурсам:

```
# Права и владельцы файлов устройств
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
```

- **Требование**: Доступ root для чтения/записи файлов устройств
- **Последствие**: Контейнеры без root получают ошибки отказа в доступе

Операции на уровне ядра

Необходимые взаимодействия с драйвером NVIDIA

Операция	Требуемые привилегии	Назначение
Загрузка модуля	CAP_SYS_MODULE	Загрузка модулей ядра NVIDIA
Управление памятью	CAP_IPC_LOCK	Выделение памяти GPU

Операция	Требуемые привилегии	Назначение
Обработка прерываний	CAP_SYS_RAWIO	Обработка прерываний GPU

Требования архитектуры device plugin Kubernetes

1. **Создание сокета**: Запись в `/var/lib/kubelet/device-plugins`
2. **Мониторинг состояния**: Доступ к `nvidia-smi` и логам ядра
3. **Распределение ресурсов**: Изменение cgroups устройств

Ограничения vGPU

- Поддержка только CUDA ниже версии 12.4
- Отсутствие поддержки MIG при включённом vGPU

Ограничения pGPU

- Отсутствие возможности совместного использования GPU (1:1 сопоставление пода и GPU)
- Требуется Kubernetes 1.25+ с включённым SR-IOV
- Ограничено GPU, подключёнными через PCIe/NVSwitch

Ограничения MPS

- Возможное распространение ошибок между объединёнными контекстами
- Требуется CUDA 11.4+ для ограничения памяти
- Нет поддержки GPU с MIG-разделением

Возможности

Содержание

vGPU (на основе Opensource GPU-Manager)

- **Тонкое разделение ресурсов**
  Делит физические ядра GPU на квоты от 1 до 100. Поддерживает динамическое распределение для мультиарендных сред, таких как AI inference и виртуальные рабочие столы.
- **Планирование с учётом топологии**
  Автоматически приоритизирует GPU, соединённые через NVLink/C2C, чтобы минимизировать задержки передачи данных между сокетами. Обеспечивает оптимальное сочетание GPU для распределённых тренировок.

pGPU (NVIDIA Device Plugin)

- **Оптимизация под NUMA**
  Обеспечивает сопоставление 1:1 GPU с Pod с привязкой к NUMA-узлу, снижая конкуренцию на шине PCIe для задач высокопроизводительных вычислений (HPC), таких как обучение LLM.

- **Эксклюзивный доступ к оборудованию**
  Предоставляет полную изоляцию физического GPU через PCIe passthrough, что идеально подходит для критически важных приложений с требованием детерминированной производительности (например, обработка медицинских изображений).

MPS (NVIDIA Multi-Process Service Plugin)

- **Оптимизация задержек выполнения**
  Позволяет слияние CUDA-ядр между процессами, снижая задержки инференса на 30-50% для приложений реального времени, таких как видеоаналитика.
- **Совместное использование ресурсов с ограничениями**
  Позволяет одновременное выполнение контекстов GPU с контролем вычислительных (0-100%) и памятьных лимитов на процесс через переменные окружения.

Установка

Содержание

Установка Kubernetes Hardware accelerator Toolkit

Предварительные требования

Перед установкой убедитесь, что выполнены следующие условия:

1. **Kubernetes Cluster**: версия ≥1.25 с включённым флагом функции `DevicePlugins`.
2. **NVIDIA Drivers**: установлены на всех GPU-узлах. Проверьте с помощью `nvidia-smi`.
3. **Container Runtime**: настроен с NVIDIA Container Toolkit (≥1.7.0) для поддержки GPU.

Установка через веб-консоль

1. **Перейдите в Cluster Plugins**:
   - Зайдите в **Platform Management** → **Catalog** → **Cluster Plugin**
   - Найдите "gpu" и нажмите **Install**

2. **Переключатели функций**: Включайте/отключайте расширенные возможности во время установки:

Опция	Функциональность	Рекомендуемый сценарий
PGPU	Изоляция физического GPU с планированием с учётом NUMA	Обучение ИИ/вычисления высокой производительности
vGPU	Виртуальное разделение GPU через GPU-Manager	Мультиарендное использование/квоты ресурсов
MPS	Multi-Process Service для совместного использования вычислительных и памятьных ресурсов	Низколатентный вывод/ параллельные задачи

Menu

Application Development

Введение

Введение

Введение в разработку приложений

Руководства

Совместимость CUDA Driver и Runtime

Иерархическая архитектура и основные концепции

Матрица совместимости версий и ограничения

Лучшие практики развертывания

Руководство по устранению неполадок

Добавление пользовательских устройств с использованием ConfigMap

Введение

Особенности

Преимущества

Функциональный модуль 1: Спецификации создания ConfigMap

Функциональный модуль 2: Определение значений ресурсов

Устранение неполадок

Устранение ошибки "float16 поддерживается только на GPU с compute capability не ниже xx" в vLLM

Описание проблемы

Корневая причина

Устранение неполадок

Решение

Профилактические меры

Связанный контент

Сбой при выделении памяти Paddle Autogrow на GPU-Manager

Описание проблемы

Корневая причина

Решение

Методы проверки

Профилактические меры

Связанный контент

Введение

Содержание

Введение в разработку приложений

Модуль разработки приложений помогает пользователям настраивать аппаратные ускорители от различных производителей (например, AMD/Intel GPU, FPGA) через единый интерфейс, что позволяет организовывать и оптимизировать гетерогенные вычислительные ресурсы в контейнеризованных средах для повышения производительности задач с высокими вычислительными нагрузками, таких как обучение ИИ и обработка изображений.

Menu

Руководства

Совместимость CUDA Driver и Runtime

Иерархическая архитектура и основные концепции

Матрица совместимости версий и ограничения

Лучшие практики развертывания

Руководство по устранению неполадок

Добавление пользовательских устройств с использованием ConfigMap

Введение

Особенности

Преимущества

Функциональный модуль 1: Спецификации создания ConfigMap

Функциональный модуль 2: Определение значений ресурсов

Совместимость CUDA Driver и Runtime

Содержание

Иерархическая архитектура и основные концепции

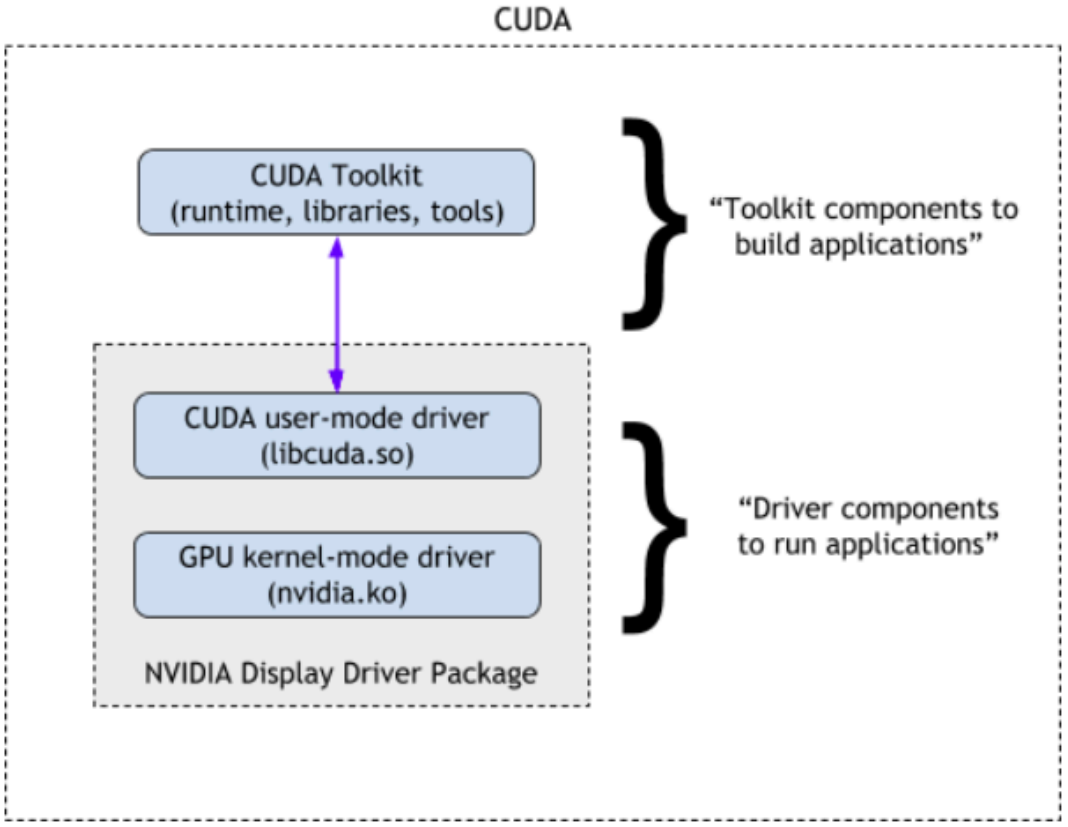

Figure 1: Components of CUDA #

1. Слой CUDA Runtime API

Техническое позиционирование

1. **Функциональная область**: Предоставляет высокоуровневые интерфейсы для разработчиков, инкапсулируя основные операции с GPU (выделение памяти, управление потоками, запуск кернелов и т.д.)

2. **Привязка версии**: Определяется версией CUDA Toolkit, используемой при сборке (например, CUDA 12.0.1)

Методы определения версии

```
# Определение в Python окружении (рекомендуемый приоритетный метод)
pip list | grep cuda
conda list |grep cuda
# Пример вывода: cu121  # cu121 означает окружение CUDA 12.1

# Определение системной runtime-библиотеки
find / -name "libcudart*"
# cudart означает cuda runtime
# Пример вывода:
/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
Указывает на CUDA 12.4
```

Если обнаружено несколько версий библиотек, необходимо проверить, какую версию использует ваша программа, например, через PATH, LD_LIBRARY_PATH или другие настройки программы

```
env |grep PATH
# Пример вывода:
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
# это означает, что ваша cuda-программа использует первую библиотеку по порядку в lib
```

2. Слой CUDA Driver API

Техническое позиционирование

1. **Функциональная область**: Низкоуровневый интерфейс, напрямую взаимодействующий с аппаратным обеспечением GPU, обрабатывающий трансляцию инструкций и планирование ресурсов
2. **Привязка версии**: Определяется версией драйвера NVIDIA, согласно спецификации SemVer

Методы определения версии

```
nvidia-smi
# Пример вывода:
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.144.03             Driver Version: 550.144.03     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 A30                     Off |   00000000:00:0B.0 Off |                    0 |
| N/A   31C    P0             28W /  165W |   10195MiB /  24576MiB |      0%      Default |
|                                         |                        |             Disabled |
+-----------------------------------------+------------------------+----------------------+
```

Матрица совместимости версий и ограничения

Физическое развертывание GPU – основные принципы совместимости

В первую очередь следует опираться на официальное заявление NVIDIA, **базовые ограничения** таковы:

1. **Версия драйвера всегда должна быть ≥ версии Runtime**
2. NVIDIA официально гарантирует **обратную совместимость на 1 мажорную версию** (например, CUDA Driver 12.x поддерживает Runtime 11.x)
3. Совместимость через две и более мажорных версий (например, Driver 12.x с Runtime 10.x) официально не поддерживается и не рекомендуется

При развертывании cuda-программы соблюдайте базовые ограничения

Формальные правила

```
+ Обязательно: версия драйвера ≥ версии Runtime
+ Рекомендуется: разница мажорных версий драйвера и Runtime ≤ 1
- Запрещено: версия драйвера < версии Runtime → может вызвать CUDA_ERROR_UNKNOWN(999)
- Нестабильно: разница мажорных версий драйвера и Runtime > 1 → возможны сбои приложения
```

Улучшения для сценариев виртуализации (HAMI/GPU-Manager)

При использовании решений виртуальных GPU, таких как GPU-Manager или HAMI, **помимо базовых ограничений выше**, необходимо соблюдать **дополнительные ограничения**:

Требования к версиям

1. Базовая версия виртуального GPU решения ≥ версии Runtime
2. Мажорные версии Runtime, драйвера и базовой версии совпадают

Особое примечание для GPU-Manager:
Мы реализовали частичную совместимость с перекрытием на 1 мажорную версию (например, базовая 12.4 поддерживает vLLM 11.8). Однако это требует индивидуальной настройки хуков для каждого приложения и должно анализироваться в каждом конкретном случае.

Лучшие практики развертывания

Рекомендуемая стратегия

• Использовать более новые версии CUDA (например, CUDA 12.x) для драйвера и Runtime при планировании новых GPU-кластеров

Альтернативные решения для устаревших систем

1. Планирование физических GPU или выделение всей карты через GPU-Manager

Выделение всей карты обеспечивает нативную совместимость, эквивалентную доступу к физическому GPU
GPU-Manager может использовать режим whole card, если задать tencent.com/vcuda-core в 100 или кратное положительное целое, например 100, 200, 300

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

2. Стратегия маркировки узлов

Маркируйте узлы на основе поддерживаемых версий драйвера CUDA:

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

Это означает, что ваш узел — cuda 12.4

Настройте affinity для планирования в деплойментах:
вы можете задать cuda-major-version и cuda-minor-version в соответствии с требованиями вашего cuda runtime

```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. Обновление версии Runtime

Устаревшие версии CUDA Runtime могут содержать уязвимости безопасности (CVE) и не поддерживать новые возможности GPU. Приоритетно обновляйтесь до CUDA 12.x.

NVIDIA рекомендует обновлять обе версии

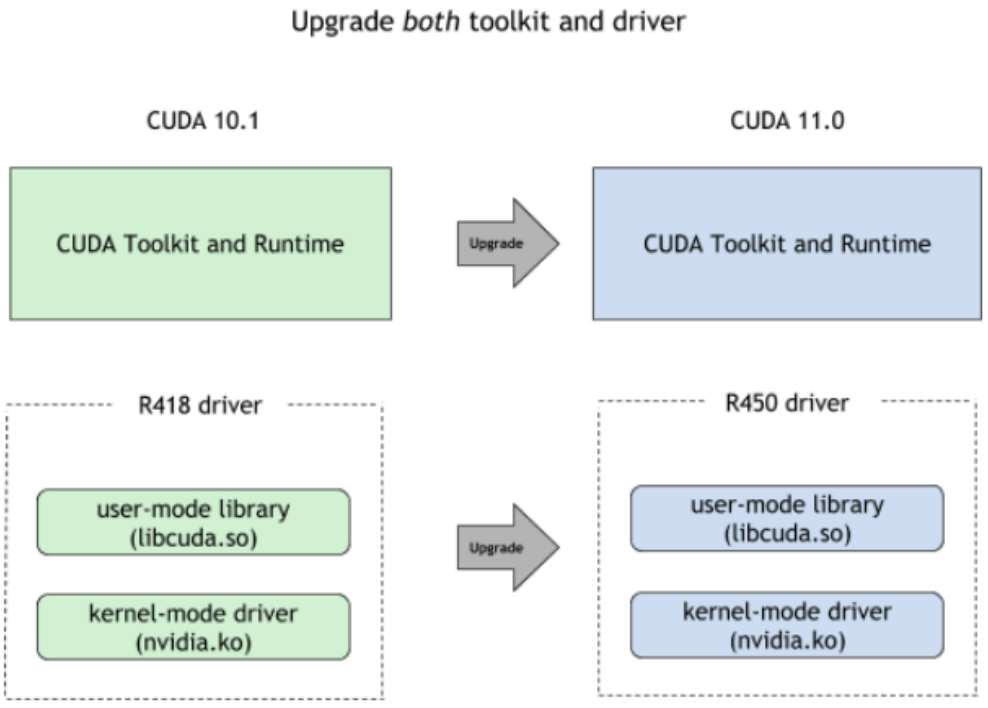

Figure 2: CUDA Upgrade Path

Руководство по устранению неполадок

Часто встречающиеся коды ошибок

Код ошибки	Описание	Рекоменд дейст
CUDA_ERROR_INVALID_IMAGE	Несовместимость драйвера и Runtime	Согласоват версию дра версией CU контейнере
CUDA_ERROR_ILLEGAL_ADDRESS	Нарушение виртуальной памяти (часто при несоответствии версий)	Проверить соответстви Runtime баз ограничени
CUDA_ERROR_UNSUPPORTED_PTX_VERSION	Несовпадение версии набора инструкций PTX	Перекомпил с явным ука -arch=sm_x

Добавление пользовательских устройств с использованием ConfigMap

Содержание

Введение

- Реализует стандартизированное определение и управление пользовательскими ресурсами Kubernetes через ConfigMap, решая задачи:
- Унифицированное управление спецификациями пользовательских ресурсов для предотвращения фрагментации конфигураций
- Стандартизированный формат определения ресурсов для лучшей поддерживаемости
- Поддержка многоязычных описаний и настройка значений по умолчанию
- Подходит для сценариев, требующих расширения модели ресурсов Kubernetes (например, управление GPU-ресурсами), предоставляя стандартизированную

структуру определения ресурсов

Особенности

- Спецификация определения ресурса с одним ключом
- Определение ресурсов с несколькими связанными ключами
- Стандартизированный интерфейс запроса ресурсов
- Поддержка двуязычных описаний на китайском и английском языках
- Механизм настройки значений ресурсов по умолчанию

Преимущества

- **Расширяемость**: управление группами ресурсов через метки
- **Безопасность**: изоляция по namespace (kube-public)
- **Стабильность**: принудительное соблюдение правил валидации формата
- **Поддерживаемость**: унифицированные спецификации меток метаданных

Функциональный модуль 1: Спецификации создания ConfigMap

Основные правила

1. **Принцип единственной ответственности**: один ConfigMap на определение ключа
2. **Namespace**: фиксирован на `namespace=kube-public`
3. **Правила именования**:

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

- `cf-crl`: фиксированный префикс
- `customName`: пользовательское валидное имя
- `keyName`: идентификатор ключа (специальные символы заменяются на '-')

4. **Требования к меткам**:

```yaml
labels:
  features.alauda.io/type: CustomResourceLimitation  # фиксированное значение
  features.alauda.io/group: {resource-group}         # например, gpu-manager
  features.alauda.io/enabled: "true"                 # флаг активации
```

Спецификация параметров

Параметр	Обязательный	Описание
формат имени	Да	Соответствует cf-crl-{customName}-{keyName}
namespace	Да	Фиксирован как kube-public
группа меток	Да	Должна содержать указанные 3 метки features

Функциональный модуль 2: Определение значений ресурсов

Пример с одним ключом

```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"       # Ключ ресурса
  dataType: "integer"                 # Тип значения
  defaultValue: "20"                  # Значение по умолчанию
  descriptionZh: ""                   # Описание на китайском
  descriptionEn: "GPU vcore count, 100 virtual cores equal 1 physical GPU core" #
Описание на английском
  group: "gpu-manager"                # Группа ресурса
  limits: "optional"                  # Политика поля limits
  requests: "disabled"                # Политика поля requests
```

Ассоциация с несколькими ключами

```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]  # Ассоциация через одинаковые метки
```

Спецификация политики

Поле	Допустимые значения	Описание
limits	disabled/required/optional	Конфигурация ограничений ресурса
requests	disabled/required/fromLimits	Конфигурация запросов ресурса

Menu

Устранение неполадок

Устранение ошибки "float16 поддерживается только на GPU с compute capability не ниже xx" в vLLM

Сбой при выделении памяти Paddle Autogrow на GPU-Manager

Устранение ошибки "float16 поддерживается только на GPU с compute capability не ниже xx" в vLLM

Содержание

Описание проблемы

Окружение

- **Аппаратное обеспечение**: GPU NVIDIA с compute capability <8.0 (например, Tesla V100, T4)
- **Типы моделей**: LLM, требующие точности bfloat16/FP8 (например, LLaMA-2-70B, GPT-NeoX-20B)

Симптомы

1. Явное сообщение об ошибке:

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

2. Ошибка компиляции ядра при загрузке модели

Связанные логи

```
# Стек вызовов ошибки vLLM
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
```

Корневая причина

Основная причина

Недостаточная compute capability GPU

Compute capability (CC) GPU не соответствует минимальным требованиям для определённых типов данных:

- **bfloat16/FP8**: требуется CC ≥8.0 (архитектура Ampere или новее)
- **Оптимизация FP16 Tensor Core**: требуется CC ≥7.0 (архитектура Volta или новее)

Технический анализ

1. **Ограничения архитектуры**:
   - GPU до Ampere (CC <8.0) не имеют специализированных матричных вычислительных блоков для операций с bfloat16
   - Tensor Cores в Volta/Turing (CC 7.0-7.5) поддерживают только смешанную точность FP16/FP32
2. **Проверка в рамках фреймворка**:

```
# Проверка compute capability в vLLM (упрощённо)
def _verify_cuda_compute_capability():
    if device.compute_capability < MIN_REQUIRED_CC:
        raise ValueError(f"Requires compute capability ≥{MIN_REQUIRED_CC}")
```

Устранение неполадок

Шаг 1: Проверка compute capability GPU

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

Шаг 2: Проверка требований к точности модели

```
cat model/config.json | grep "torch_dtype"
# Ожидаемый вывод: "bfloat16" или "float16"
```

Шаг 3: Проверка совместимости фреймворка

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

Решение

Решение при недостаточной compute capability

Важные моменты

- Ожидается снижение производительности при понижении точности
- Точность модели может измениться при использовании разных типов точности

Требования

- CUDA Toolkit версии ≥11.8

Шаги

1. **Изменить yaml InferenceService**:
   добавить аргумент --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 # Принудительное использование точности FP16
          - --tensor-parallel-size=1
        resources:
          limits:
            nvidia.com/gpu: '1'
```

2. **Дождаться перезапуска деплоя**

Профилактические меры

1. **Проверки перед запуском**:

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

2. **Конфигурация кластера**:

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

3. **Оптимизация модели**:

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

Связанный контент

Таблица compute capability GPU

Архитектура	Диапазон CC	Поддерживаемые типы точности
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 с динамическим масштабом

Официальные ссылки

1. NVIDIA Compute Capability Table ↗
2. vLLM Hardware Requirements ↗

Сбой при выделении памяти Paddle Autogrow на GPU-Manager

Содержание

Описание проблемы

Симптомы

При одновременном включении стратегии выделения памяти Autogrow в PaddlePaddle и виртуализированного управления памятью GPU-Manager могут возникать следующие аномалии:

1. Ошибки OOM из-за прерывистой аллокации памяти
2. Аномальные колебания загрузки GPU
3. Случайные сбои процесса обучения
4. Несоответствие использования памяти между отчетами nvidia-smi и статистикой фреймворка

Корневая причина

Анализ корневой причины

1. **Конфликт стратегий выделения памяти**
   Autogrow Paddle использует динамическое сегментированное выделение, тогда как виртуализация GPU-Manager требует непрерывного отображения физической памяти
2. **Несовместимость механизмов управления**
   Механизм отложенного освобождения Autogrow конфликтует со стратегией возврата памяти GPU-Manager
3. **Конфликт в поддержке метаданных**
   Раздельное ведение метаданных обеими системами приводит к несогласованному представлению памяти

Механизм срабатывания:

- Autogrow пытается оптимально подобрать размер блока при выделении
- Слой виртуализации GPU-Manager перехватывает запросы к физической памяти
- Непрерывные аллокации вызывают сбои в отображении виртуальных адресов
- Двойное управление приводит к исключениям согласованности метаданных

Решение

Обзор решения

Принудительно переключить Paddle на традиционную стратегию выделения через переменную окружения:

```
FLAGS_allocator_strategy=naive_best_fit
```

Особенности

1. Требуется перезапуск процесса обучения
2. Может снизить эффективность повторного использования памяти в Paddle

Шаги реализации

Развертывание в Kubernetes

1. Отредактировать конфигурацию Deployment

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

2. Применить конфигурацию

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

3. Проверить конфигурацию

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

Развертывание на Bare Metal

1. Установить переменную окружения перед запуском

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

2. Или задать в Python-коде

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

Методы проверки

1. Проверить подтверждение стратегии выделения в логах

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

2. Мониторинг непрерывности выделения памяти

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

3. Тест на стрессовую нагрузку

```
# Скрипт для теста непрерывного выделения памяти
import paddle
for i in range(10):
    data = paddle.randn([1024, 1024, 100], dtype='float32')
    print(f"Allocated {i+1}GB")
```

Профилактические меры

1. **Проверка совместимости версий**
   Изучать релиз-ноты Paddle на предмет изменений в стратегии выделения памяти при обновлениях

2. **Настройка мониторинга**
   Добавить правило оповещения Prometheus:

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

3. **Базовое тестирование**
   Выполнять базовые тесты выделения памяти для новых сред:

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

Связанный контент

Сравнение стратегий выделения памяти

Стратегия	Преимущества	Недостатки
autogrow	Высокая эффективность	Плохая производительность с большими блоками
naive_best_fit	Стабильное выделение	Возможна фрагментация

Ссылки

Paddle Memory Optimization Whitepaper ↗

Menu

Управление конфигурацией

Введение

Введение

Введение в управление конфигурациями

Руководства

Настройка аппаратного ускорителя на GPU-узлах

Предварительные требования

Конфигурация физического GPU

Конфигурация NVIDIA MPS (драйвер должен поддерживать CUDA версии >= 11.5)

Конфигурация GPU-Manager

Проверка результатов

Введение

Содержание

Введение в управление конфигурациями

Управление конфигурациями — это централизованный портал документации для настройки возможностей аппаратного ускорения GPU в средах Kubernetes. Этот постоянно обновляемый документ предоставляет администраторам единые рекомендации по настройке физических GPU (pGPU), виртуальных GPU (vGPU) и Multi-Process Service (MPS) в гибридной инфраструктуре.

Menu

Руководства

Настройка аппаратного ускорителя на GPU-узлах

Предварительные требования

Конфигурация физического GPU

Конфигурация NVIDIA MPS (драйвер должен поддерживать CUDA версии >= 11.5)

Конфигурация GPU-Manager

Проверка результатов

Настройка аппаратного ускорителя на GPU-узлах

По мере увеличения объёмов бизнес-данных, особенно в сценариях, таких как искусственный интеллект и анализ данных, вы можете захотеть использовать возможности GPU в вашем самостоятельно построенном бизнес-кластере для ускорения обработки данных. Помимо подготовки GPU-ресурсов для узлов кластера, необходимо также выполнить настройку GPU.

В данном решении узлы кластера, обладающие возможностями GPU-вычислений, называются **GPU Nodes**.

Примечание: Если не указано иное, шаги выполнения применимы к обоим типам узлов. По вопросам установки драйверов обращайтесь к официальной документации NVIDIA по установке ↗.

Содержание

Предварительные требования

GPU-ресурсы подготовлены на рабочем узле, который относится к GPU-узлу, описанному в этом разделе.

Установка драйвера GPU

Внимание: Если GPU-узел использует плагин NVIDIA MPS, убедитесь, что архитектура GPU узла — Volta или новее (Volta/Turing/Ampere/Hopper и т. д.), а драйвер поддерживает CUDA версии 11.5 или выше.

Получение адреса для загрузки драйвера

1. Войдите на GPU-узел и выполните команду `lspci |grep -i NVIDIA`, чтобы проверить модель GPU узла.

   В следующем примере модель GPU — Tesla T4.

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

2. Перейдите на официальный сайт NVIDIA ↗, чтобы получить ссылку для загрузки драйвера.

   1. Нажмите на **Drivers** в верхней навигационной панели на главной странице.

2. Заполните необходимые данные для загрузки драйвера в соответствии с моделью GPU узла.
3. Нажмите **Search**.
4. Нажмите **Download**.
5. Щёлкните правой кнопкой мыши по **Download** > **Copy Link Address**, чтобы скопировать ссылку для загрузки драйвера.

3. Выполните следующие команды на GPU-узле для создания каталога `/home/gpu` и загрузки файла драйвера в этот каталог.

```
# Создать каталог /home/gpu
mkdir -p /home/gpu
cd /home/gpu/
# Загрузить файл драйвера в каталог /home/gpu, пример: wget
https://cn.download.nvidia.com/tesla/515.65.01/NVIDIA-Linux-x86_64-515.65.01.run
wget <Driver download address>
# Проверить успешность загрузки файла драйвера, если возвращается имя файла
драйвера (например: NVIDIA-Linux-x86_64-515.65.01.run), значит загрузка прошла
успешно
ls <Driver file name>
```

Установка драйвера

1. Выполните следующую команду на GPU-узле для установки пакетов gcc и kernel-devel, соответствующих текущей операционной системе.

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

2. Выполните следующие команды для установки драйвера GPU.

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

3. После установки выполните команду `nvidia-smi`. Если возвращается информация о GPU, аналогичная приведённому примеру, значит установка драйвера прошла

успешно.

```
# 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                                                 |
+-----------------------------------------------------------------------------+
```

Установка NVIDIA Container runtime

1. На **GPU Node** добавьте репозиторий NVIDIA yum.

```
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
```

   При появлении сообщения "Metadata cache created." добавление прошло успешно.

2. Установите NVIDIA Container Runtime.

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

При появлении сообщения `Complete!` установка завершена успешно.

3. Настройте Runtime по умолчанию. Добавьте следующую конфигурацию в файл.

   - Containerd: Измените файл `/etc/containerd/config.toml`.

```
[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: Измените файл `/etc/docker/daemon.json`.

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

4. Перезапустите Containerd / Docker.

- Containerd

```
systemctl restart containerd   #Перезапуск

crictl info |grep Runtime  #Проверка
```

- Docker

```
systemctl restart docker   #Перезапуск

docker info |grep Runtime  #Проверка
```

Конфигурация физического GPU

Развёртывание плагина физического GPU в GPU бизнес-кластере

В интерфейсе управления GPU-кластером выполните следующие действия:

1. В левой боковой панели каталога выберите подраздел "Cluster Plugins", нажмите для развёртывания "ACP GPU Device Plugin" и включите опцию "pGPU";

2. Во вкладке "Nodes" выберите узлы, на которых необходимо развернуть физический GPU, затем нажмите "Label and Taint Manager", добавьте "device label" и выберите "pGPU", нажмите OK;

3. Во вкладке "Pods" проверьте статус работы контейнерной группы, соответствующей nvidia-device-plugin-ds, чтобы убедиться в отсутствии сбоев и что она запущена на указанных узлах.

Конфигурация NVIDIA MPS (драйвер должен поддерживать CUDA версии >= 11.5)

Развёртывание плагина NVIDIA MPS в GPU бизнес-кластере

В интерфейсе управления GPU-кластером выполните следующие действия:

1. В левой боковой панели каталога выберите подраздел "Cluster Plugins", нажмите для развёртывания "ACP GPU Device Plugin" и включите опцию "MPS";

2. Во вкладке "Nodes" выберите узлы, на которых необходимо развернуть физический GPU, затем нажмите "Label and Taint Manager", добавьте "device label" и выберите "MPS", нажмите OK;

3. Во вкладке "Pods" проверьте статус работы контейнерной группы, соответствующей nvidia-mps-device-plugin-daemonset, чтобы убедиться в отсутствии сбоев и что она запущена на указанных узлах.

Настройка kube-scheduler (kubernetes >= 1.23)

1. На **узле управления бизнес-кластером** проверьте, правильно ли scheduler ссылается на политику планирования.

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

Проверьте, есть ли опция –config со значением /etc/kubernetes/scheduler-config.yaml, например

```
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
```

Примечание: Указанные параметры и значения — это конфигурации по умолчанию платформы. Если вы их изменяли, пожалуйста, верните к значениям по умолчанию. Ваши оригинальные пользовательские настройки можно скопировать в файл политики планирования.

2. Проверьте конфигурацию файла политики планирования.

   1. Выполните команду: `kubectl describe service kubernetes -n default |grep Endpoints`.

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

   2. Замените содержимое файла `/etc/kubernetes/scheduler-config.yaml` на всех Master-узлах следующим содержимым, где `${kube-apiserver}` нужно заменить на вывод из предыдущего шага.

```
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
```

Если в schedule-config.yaml уже есть extenders, добавьте этот yaml в конец.

```
- 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. Выполните следующую команду для получения ID контейнера:

- Containerd: Выполните `crictl ps |grep kube-scheduler`, вывод будет примерно таким, первый столбец — это ID контейнера.

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

- Docker: Выполните `docker ps |grep kube-scheduler`, вывод будет примерно таким, первый столбец — это ID контейнера.

```
30528a45a118   d8a9fef7349c    "kube-scheduler --au..."   37 minutes ago   Up 37
minutes     k8s_kube-scheduler_kube-scheduler-192.168.130.240_kube-
system_3e9f7007b38f4deb6ffd1c7587621009_28
```

4. Перезапустите контейнер Containerd/Docker, используя полученный ID контейнера.

   - Containerd

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

5. Перезапустите Kubelet.

```
systemctl restart kubelet
```

Конфигурация GPU-Manager

Настройка kube-scheduler (kubernetes >= 1.23)

1. На **узле управления бизнес-кластером** проверьте, правильно ли scheduler ссылается на политику планирования.

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

   Проверьте, есть ли опция –config со значением /etc/kubernetes/scheduler-config.yaml, например

```
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
```

Примечание: Указанные параметры и значения — это конфигурации по умолчанию платформы. Если вы их изменяли, пожалуйста, верните к значениям по умолчанию. Ваши оригинальные пользовательские настройки можно скопировать в файл политики планирования.

2. Проверьте конфигурацию файла политики планирования.

   1. Выполните команду: `kubectl describe service kubernetes -n default |grep Endpoints`.

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

   2. Замените содержимое файла `/etc/kubernetes/scheduler-config.yaml` на всех Master-узлах следующим содержимым, где `${kube-apiserver}` нужно заменить на вывод из предыдущего шага.

```
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. Выполните следующую команду для получения ID контейнера:

   - Containerd: Выполните `crictl ps |grep kube-scheduler`, вывод будет примерно таким, первый столбец — это ID контейнера.

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

   - Docker: Выполните `docker ps |grep kube-scheduler`, вывод будет примерно таким, первый столбец — это ID контейнера.

   ```
   30528a45a118   d8a9fef7349c    "kube-scheduler --au..."   37 minutes ago   Up 37 minutes     k8s_kube-scheduler_kube-scheduler-192.168.130.240_kube-system_3e9f7007b38f4deb6ffd1c7587621009_28
   ```

4. Перезапустите контейнер Containerd/Docker, используя полученный ID контейнера.

   - Containerd

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

5. Перезапустите Kubelet.

```
systemctl restart kubelet
```

Развёртывание плагина GPU Manager в GPU бизнес-кластере

В интерфейсе управления GPU-кластером выполните следующие действия:

1. В левой боковой панели каталога выберите подраздел "Cluster Plugins", нажмите для развёртывания "ACP GPU Device Plugin" и включите опцию "GPU-Manager";
2. Во вкладке "Nodes" выберите узлы, на которых необходимо развернуть физический GPU, затем нажмите "Label and Taint Manager", добавьте "device label" и выберите "vGPU", нажмите OK;
3. Во вкладке "Pods" проверьте статус работы контейнерной группы, соответствующей gpu-manager-daemonset, чтобы убедиться в отсутствии сбоев и что она запущена на указанных узлах.

Проверка результатов

Метод 1: Проверьте наличие доступных GPU-ресурсов на GPU-узлах, выполнив следующую команду на управляющем узле бизнес-кластера:

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

Метод 2: Разверните GPU-приложение на платформе, указав необходимое количество GPU-ресурсов. После развертывания выполните exec в Pod и выполните следующую команду:

```
# 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                                                 |
+-----------------------------------------------------------------------------+
```

Проверьте, корректно ли получена информация о GPU.

Menu

Мониторинг ресурсов

Введение

Введение

Введение в мониторинг ресурсов

Преимущества

Сценарии применения

Ограничения использования

Руководства

Мониторинг ресурсов GPU

Feature Overview

Core Features

Feature Advantages

Node Monitoring

Pod Monitoring

Time Range Selection

Введение

Содержание

Введение в мониторинг ресурсов

Мониторинг ресурсов — это ключевой компонент Kubernetes Hardware Accelerator Suite, предназначенный для обеспечения всесторонней видимости использования GPU-ресурсов в ваших контейнеризованных рабочих нагрузках. Этот модуль предоставляет как **использование вычислительных ресурсов**, так и **потребление памяти GPU** на двух основных уровнях:

Мониторинг ресурсов — это ключевой компонент Kubernetes Hardware Accelerator Suite, предназначенный для обеспечения всесторонней видимости использования GPU-ресурсов в ваших контейнеризованных рабочих нагрузках. Этот модуль предоставляет как **использование вычислительных ресурсов**, так и **потребление памяти GPU** на двух основных уровнях:

- **Мониторинг на уровне узла**: отслеживание совокупного использования GPU-ресурсов на всех узлах Kubernetes
- **Мониторинг на уровне пода**: анализ потребления GPU по отдельным рабочим нагрузкам с детализацией по подам

Интегрированный с основными модулями платформы для ускорителей (pGPU/vGPU(GPU-Manager)/MPS), этот инструмент мониторинга позволяет пользователям оптимизировать распределение GPU, обеспечивать соблюдение квот ресурсов и устранять узкие места в производительности AI/ML нагрузок, сервисов реального времени и т.д.

Преимущества

Основные преимущества мониторинга ресурсов заключаются в следующем:

- **Многомерная наблюдаемость**

  Одновременный мониторинг как вычислительных блоков (CUDA-ядер), так и использования памяти на физических и виртуальных GPU, обеспечивающий комплексное понимание паттернов использования ускорителей.

- **Иерархический сбор метрик**

  Сбор данных как на уровне узла, так и на уровне пода, что позволяет сопоставлять общекластерные тенденции использования ресурсов с требованиями отдельных рабочих нагрузок.

- **Нативная интеграция**

  Бесшовная работа со всеми модулями ускорителей (pGPU/vGPU/MPS) без необходимости установки дополнительных агентов, используя нативные Kubernetes-пайплайны метрик.

- **Исторический анализ**

  Хранение метрик GPU с настраиваемыми периодами хранения (по умолчанию 7 дней) для планирования емкости и анализа паттернов использования с помощью встроенных инструментов визуализации.

Сценарии применения

Основные сценарии применения мониторинга ресурсов включают:

- **Оптимизация производительности**

  Выявление недоиспользуемых GPU в кластерах для обучения и корректировка запросов ресурсов для deep learning нагрузок. Например, обнаружение подов, которые постоянно используют менее <30% выделенной памяти GPU, для оптимизации распределения памяти.

- **Управление мультиарендностью**

  Обеспечение соблюдения квот GPU в совместных средах путем мониторинга потребления vGPU по командам. Отслеживание суммарного использования в сравнении с выделенными квотами в развертываниях AI платформ.

- **Распределение затрат**

  Формирование отчетов по использованию GPU на уровне namespace для моделей chargeback/showback в корпоративных Kubernetes-средах с сопоставлением метрик подов с организационными единицами.

- **Диагностика сбоев**

  Исследование инцидентов OOM (Out-of-Memory) в нагрузках с ускорением на GPU путем анализа тенденций использования памяти перед аварийным завершением контейнеров. Кросс-ссылка с событиями Kubernetes для выявления первопричин.

- **Планирование емкости**

  Анализ исторических паттернов использования GPU (например, периоды пикового спроса на вычисления) для принятия решений по масштабированию инфраструктуры и распределению бюджета на AI-инфраструктуру.

Ограничения использования

При использовании мониторинга ресурсов обратите внимание на следующие ограничения:

- **Зависимости модулей**

- Требуется развертывание как минимум одного модуля ускорителя (pGPU/vGPU/MPS) в кластере

Menu

Руководства

Мониторинг ресурсов GPU

GPU Resource Monitoring

Содержание

Feature Overview

Функция Resource Monitoring позволяет отслеживать в реальном времени и анализировать исторические данные по использованию GPU и памяти на узлах и в подах внутри Container Platform. Эта функциональность помогает администраторам и разработчикам:

- **Мониторинг производительности GPU**: выявлять узкие места в распределении ресурсов GPU.

- **Устранение неполадок**: анализировать тенденции использования GPU для отладки проблем, связанных с ресурсами.
- **Оптимизация рабочих нагрузок**: принимать решения на основе данных для улучшения распределения нагрузок.

Применимые сценарии:

- Мониторинг в реальном времени приложений с интенсивным использованием GPU.
- Исторический анализ использования GPU для планирования емкости.
- Сравнение производительности GPU между несколькими узлами/подами.

Преимущества:

- Повышенная видимость потребления ресурсов GPU.
- Улучшенная эффективность кластера благодаря практическим рекомендациям.

Core Features

- **Мониторинг на уровне узла**: отслеживание использования GPU и памяти по каждому узлу.
- **Мониторинг на уровне пода**: мониторинг метрик GPU для отдельных подов.
- **Пользовательские временные диапазоны**: анализ данных от 30 минут до 7 дней.

Feature Advantages

- **Визуализация в реальном времени**: интерактивные панели с возможностью автозагрузки.
- **Многомерная фильтрация**: сужение метрик по типу GPU, namespace или поду.

Node Monitoring

Отслеживайте ресурсы GPU на уровне узла, выполнив следующие шаги:

1 Доступ к панелям GPU

1. Перейдите в представление **Platform Management**
2. Откройте **Operations Center → Monitoring → Dashboards**
3. Переключитесь в каталог **GPU**

2 Выбор метрик узла

1. Выберите панель **Node Monitoring**
2. Выберите целевой узел из выпадающего списка
3. Укажите временной диапазон:

- Последние 30 минут
- Последний 1/6/12/24 часа
- Последние 2/7 дней
- Пользовательский диапазон

3 Интерпретация метрик

Metric	Description
GPU Utilization	Процент использования вычислительной мощности GPU (0-100%)
GPU Memory Usage	Объем потребляемой памяти по сравнению с доступной (в GiB)

Pod Monitoring

Анализируйте использование GPU на уровне пода с детальной фильтрацией:

1 Доступ к метрикам пода

1. Перейдите в панели каталога **GPU**
2. Выберите **Pod Monitoring**

2 Настройка фильтров

1. Выберите тип GPU:

- pGPU
- GPU-Manager(vGPU)
- MPS

2. Выберите namespace, содержащий GPU-поды
3. Выберите конкретный под

3 Ключевые метрики

Metric	Description
Pod GPU Utilization	Использование вычислительной мощности GPU выбранным подом
Pod GPU Memory	Распределение памяти для выбранного пода

Time Range Selection

Обе панели поддерживают гибкий выбор временного окна:

```
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
```