

// Jivops

GPUs en Kubernetes

Guía completa — 30 lecciones

Fundamentos · Scheduling · Compartición de GPU · Dynamic Resource
Allocation · Observabilidad · Producción avanzada

Contenido

Módulo 1: Fundamentos

- Lección 01 — Por qué las GPUs requieren gestión especial en Kubernetes
- Lección 02 — Device Plugins — cómo Kubernetes expone hardware especializado
- Lección 03 — NVIDIA GPU Operator — instalación y componentes
- Lección 04 — Node Feature Discovery — detectar hardware disponible
- Lección 05 — Runtime de contenedores compatible con GPU (NVIDIA Container Toolkit)

Módulo 2: Scheduling y asignación

- Lección 06 — Solicitar GPUs en un Pod — resources.limits
- Lección 07 — GPU scheduling — cómo decide el scheduler
- Lección 08 — Node selectors y taints para nodos con GPU
- Lección 09 — Topología de GPU — NVLink y afinidad
- Lección 10 — Bin packing vs spread — estrategias de colocación

Módulo 3: Compartición de GPU

- Lección 11 — Por qué compartir GPUs entre workloads
- Lección 12 — Time-slicing — compartición por tiempo
- Lección 13 — MIG (Multi-Instance GPU) — particionamiento físico
- Lección 14 — MPS (Multi-Process Service)
- Lección 15 — Comparación de estrategias de compartición

Módulo 4: Dynamic Resource Allocation (DRA)

- Lección 16 — Qué es DRA y por qué reemplaza al modelo de device plugins clásico
- Lección 17 — ResourceClaim y ResourceClass
- Lección 18 — Migración de device plugins a DRA
- Lección 19 — DRA para hardware heterogéneo
- Lección 20 — Estado actual y adopción de DRA

Módulo 5: Observabilidad y optimización

- Lección 21 — Monitoreo de GPU con DCGM Exporter
- Lección 22 — Métricas clave de utilización de GPU
- Lección 23 — Detección de GPUs infrautilizadas
- Lección 24 — Autoscaling de nodos con GPU (Cluster Autoscaler)
- Lección 25 — Optimización de costos de GPU en cloud

Módulo 6: Producción avanzada

Lección 26 — GPUs en cargas de inferencia vs entrenamiento

Lección 27 — Checkpointing y tolerancia a fallos en jobs con GPU

Lección 28 — Seguridad de GPU — aislamiento entre workloads

Lección 29 — Troubleshooting común de GPUs en Kubernetes

Lección 30 — Proyecto final: plataforma de GPUs en Kubernetes lista para producción

Módulo 1: Fundamentos

Lección 01 — Por qué las GPUs requieren gestión especial en Kubernetes

Kubernetes no tiene conocimiento nativo de qué es una GPU ni cómo exponerla de forma segura a un contenedor, requiere un mecanismo de extensión específico para reconocerla como recurso.

CPU y memoria existen en absolutamente todas las máquinas y Kubernetes se diseñó asumiendo su presencia universal desde el inicio; una GPU es hardware especializado que no todos los nodos tienen, por lo que necesitó un mecanismo de extensión añadido específicamente para reconocerla como recurso opcional.

Lección 02 — Device Plugins — cómo Kubernetes expone hardware especializado

Detecta el hardware especializado disponible en el nodo y lo anuncia al kubelet como un recurso extendido que puede ser solicitado y asignado a los Pods.

Al usar un mecanismo genérico de 'recursos extendidos' en vez de codificar el soporte de GPU directamente en el núcleo de Kubernetes, el mismo framework de extensión puede usarse también para otros tipos de hardware especializado futuro, sin requerir modificar el código central de Kubernetes cada vez.

Lección 03 — NVIDIA GPU Operator — instalación y componentes

Despliega y gestiona automáticamente todos los componentes necesarios (drivers, device plugin, container toolkit, monitoreo) requeridos para que las GPUs funcionen correctamente, sin instalación manual de cada pieza por separado.

Instalar manualmente cada componente en cada nodo por separado aumenta el riesgo de que distintos nodos terminen con versiones ligeramente distintas de drivers o configuración; el GPU Operator garantiza que todos los nodos gestionados reciban exactamente la misma configuración coordinada y consistente.

Lección 04 — Node Feature Discovery — detectar hardware disponible

Características detalladas del hardware y software del nodo (modelo de CPU, presencia y modelo de GPU, capacidades del kernel), etiquetando el nodo automáticamente con esa información.

Un cluster con GPUs de distintos modelos (algunas más potentes que otras) necesita poder distinguir entre ellas específicamente al programar una carga que requiere las capacidades de un modelo en particular; una etiqueta genérica de solo 'tiene GPU' no permitiría hacer esa distinción precisa.

Lección 05 — Runtime de contenedores compatible con GPU (NVIDIA Container Toolkit)

Configura el runtime de contenedores (como containerd) para exponer correctamente los dispositivos GPU y las bibliotecas necesarias de NVIDIA dentro del namespace del contenedor en ejecución.

El aislamiento normal de un contenedor por defecto no expone dispositivos de hardware específicos del host como la GPU; sin la configuración específica que aporta el toolkit, ese aislamiento estándar mantendría la GPU completamente inaccesible desde dentro del contenedor, aunque exista físicamente en el nodo.

Módulo 2: Scheduling y asignación

Lección 06 — Solicitar GPUs en un Pod — `resources.limits`

Declarando el recurso extendido correspondiente (como `nvidia.com/gpu`) dentro de la sección `resources.limits` del contenedor, con la cantidad deseada.

En el modelo clásico de recursos extendidos, una GPU se trata como una unidad indivisible asignada completa a un único contenedor a la vez, a diferencia de CPU que sí permite fracciones; compartir una misma GPU entre múltiples cargas requiere mecanismos adicionales específicos como `time-slicing` o `MIG`.

Lección 07 — GPU scheduling — cómo decide el scheduler

La disponibilidad de GPUs libres anunciadas por el device plugin en cada nodo candidato, filtrando aquellos nodos que no tienen suficiente capacidad GPU disponible en ese momento.

El scheduler solo puede colocar el Pod en un nodo que realmente tenga la capacidad GPU disponible solicitada; si ningún nodo cumple ese requisito en ese momento, el Pod permanece en estado `Pending` esperando a que se libere capacidad suficiente en algún nodo candidato.

Lección 08 — Node selectors y taints para nodos con GPU

Evita que Pods que no necesitan GPU se programen accidentalmente en esos nodos especializados, reservando esa capacidad costosa exclusivamente para cargas que realmente la requieren.

Sin ese taint protegiendo el nodo, el scheduler podría colocar cualquier Pod normal (que no requiere GPU) en ese nodo especializado simplemente porque tiene recursos de CPU/memoria disponibles, desperdiciando esa capacidad costosa de GPU que debería reservarse exclusivamente para cargas que realmente la necesitan.

Lección 09 — Topología de GPU — NVLink y afinidad

GPUs conectadas mediante NVLink (una interconexión de alta velocidad) pueden comunicarse entre sí mucho más rápido que a través del bus PCIe estándar, mejorando significativamente el rendimiento de entrenamientos que requieren sincronización frecuente entre GPUs.

Un entrenamiento distribuido con múltiples GPUs requiere sincronizar constantemente los gradientes entre ellas; si esas GPUs están conectadas por una interconexión lenta (en vez de NVLink de alta velocidad), esa comunicación frecuente se convertiría en un cuello de botella que ralentiza todo el proceso de entrenamiento, algo que un scheduler consciente de topología ayuda a evitar.

Lección 10 — Bin packing vs spread — estrategias de colocación

`Bin packing` intenta concentrar las cargas en el menor número posible de nodos (maximizando el uso de cada uno antes de usar otro), mientras `spread` distribuye las cargas de forma más uniforme entre todos los nodos disponibles.

Al concentrar las cargas en el menor número posible de nodos activos (`bin packing`), los nodos restantes sin ninguna carga podrían apagarse o escalarse hacia abajo, reduciendo el costo total; con `spread`, más nodos permanecerían parcialmente ocupados sin poder apagarse completamente, aumentando el costo total de nodos activos pagados.

Módulo 3: Compartición de GPU

Lección 11 — Por qué compartir GPUs entre workloads

Muchas cargas de trabajo (especialmente de inferencia con tráfico bajo o moderado) no utilizan la capacidad completa de una GPU dedicada, dejando recursos desperdiciados que podrían aprovecharse por otras cargas simultáneas.

Si ese modelo solo consume el 20% de la capacidad total de la GPU, el 80% restante permanecería completamente ocioso mientras esa GPU está dedicada exclusivamente a esa única carga; compartirla con otra carga complementaria permitiría aprovechar esa capacidad desperdiciada.

Lección 12 — Time-slicing — compartición por tiempo

Múltiples contenedores comparten la misma GPU física alternándose en el tiempo de acceso al procesador de la GPU, sin partición física real de la memoria ni aislamiento estricto entre ellos.

Sin una partición física real de memoria entre los contenedores que comparten la GPU mediante time-slicing, si una de esas cargas consume memoria de forma descontrolada o inesperada, podría afectar negativamente el funcionamiento de las demás cargas que comparten esa misma GPU física, ya que no hay una barrera de aislamiento real entre ellas.

Lección 13 — MIG (Multi-Instance GPU) — particionamiento físico

MIG divide físicamente la GPU en instancias independientes con su propia memoria y recursos de cómputo dedicados, dando aislamiento real de hardware entre las cargas que la usan.

En un escenario multi-tenant con clientes externos que no confían entre sí, la partición física real de MIG (con memoria y recursos verdaderamente separados) da una garantía de aislamiento mucho más sólida que time-slicing, que solo alterna el acceso en el tiempo sin ninguna separación física real entre las cargas.

Lección 14 — MPS (Multi-Process Service)

Permite que múltiples procesos CUDA ejecuten sus kernels de forma concurrente real en la misma GPU (en vez de estrictamente turnado en el tiempo), mejorando la utilización cuando las cargas individuales son pequeñas.

Si cada carga individual es pequeña y no satura por sí sola toda la capacidad de la GPU, ejecutarlas de forma estrictamente turnada en el tiempo (una a la vez) desperdiciaría capacidad ociosa durante cada turno; MPS permite que esas cargas pequeñas se ejecuten realmente en paralelo, aprovechando mejor la capacidad total agregada de la GPU.

Lección 15 — Comparación de estrategias de compartición

La necesidad de garantías reales de aislamiento de memoria y rendimiento predecible entre las cargas que comparten la GPU, algo que MIG ofrece mediante partición física real y time-slicing no.

MIG requiere GPUs de arquitecturas específicas compatibles con esa tecnología de partición física; si el hardware disponible del equipo es más antiguo y no soporta esa capacidad, time-slicing se convertiría en la única opción práctica realmente disponible, independientemente de cuál estrategia fuera preferible en un escenario ideal sin esa limitación de hardware.

Módulo 4: Dynamic Resource Allocation (DRA)

Lección 16 — Qué es DRA y por qué reemplaza al modelo de device plugins clásico

El modelo clásico trata los recursos extendidos como contadores simples (como 'nvidia.com/gpu: 1'), sin capacidad de expresar requisitos más complejos como una configuración específica de partición MIG o afinidad de topología en la propia solicitud.

El modelo clásico de recursos extendidos solo permite expresar una cantidad simple del recurso solicitado (como un contador), sin ningún mecanismo nativo para especificar características más detalladas del hardware requerido, como capacidad de memoria específica o topología de interconexión deseada entre dispositivos.

Lección 17 — ResourceClaim y ResourceClass

Una ResourceClass define una plantilla o categoría de recursos disponibles (con sus parámetros configurables), mientras un ResourceClaim es la solicitud concreta de un Pod pidiendo un recurso de esa clase con parámetros específicos.

Al separar la definición general (ResourceClass) de la solicitud específica (ResourceClaim), distintos Pods pueden reutilizar la misma clase de recurso base mientras cada uno especifica parámetros particulares propios de su solicitud, dando una flexibilidad de composición que el modelo clásico de un simple contador no permitía expresar.

Lección 18 — Migración de device plugins a DRA

Verificar la compatibilidad y madurez del soporte de DRA en la versión de Kubernetes del cluster, dado que es una funcionalidad relativamente más reciente que el modelo clásico ya establecido.

Al ser una funcionalidad relativamente más reciente que el modelo clásico ya consolidado, el soporte y estabilidad de DRA puede variar significativamente según la versión específica de Kubernetes; verificar esa madurez antes de migrar cargas de producción críticas evita sorpresas con una funcionalidad que podría no estar completamente estable todavía en esa versión particular.

Lección 19 — DRA para hardware heterogéneo

Permite expresar solicitudes de recursos que distinguen entre distintos tipos y capacidades específicas de hardware disponible, más allá de un simple contador genérico que trataría a todo el hardware especializado por igual.

Cada tipo de hardware especializado tiene capacidades, drivers y características distintas entre sí; una carga de trabajo que específicamente necesita las capacidades de un tipo de acelerador en particular necesita poder expresar ese requisito preciso, en vez de que el sistema trate indistintamente a cualquier acelerador disponible como si fueran intercambiables entre sí.

Lección 20 — Estado actual y adopción de DRA

Evaluar si la madurez actual de DRA en la versión de Kubernetes disponible ya cubre las necesidades reales del caso de uso, o si el modelo clásico de device plugins sigue siendo la opción más estable y probada para ese escenario específico por ahora.

Si el caso de uso específico del equipo no requiere las capacidades más avanzadas y expresivas que ofrece DRA (como requisitos complejos de topología o hardware heterogéneo), el modelo clásico ya estable y probado podría seguir siendo la opción más práctica, evitando la complejidad y el riesgo adicional de adoptar una funcionalidad más nueva sin una necesidad real que lo justifique.

Módulo 5: Observabilidad y optimización

Lección 21 — Monitoreo de GPU con DCGM Exporter

Expone las métricas nativas de NVIDIA DCGM (Data Center GPU Manager) en un formato que Prometheus puede consultar y recolectar, dando visibilidad de utilización, temperatura y otros indicadores clave de cada GPU.

Al exponer las métricas en el mismo formato que Prometheus ya usa para el resto de la infraestructura del cluster, las métricas de GPU se integran naturalmente en los mismos dashboards y sistema de alertas existentes, en vez de requerir una herramienta de monitoreo completamente aislada y separada específicamente para GPU.

Lección 22 — Métricas clave de utilización de GPU

La utilización de GPU mide qué tan ocupado está el procesador de cómputo en un período de tiempo, mientras la utilización de memoria mide cuánta capacidad de memoria de la GPU está siendo consumida, y ambas pueden comportarse de forma independiente entre sí.

Si la memoria está ocupada (los datos y el modelo ya están cargados) pero el procesador de cómputo muestra baja actividad, esto sugiere que la GPU está esperando ociosamente a que lleguen nuevos datos para procesar (un cuello de botella de I/O o preparación de datos), en vez de estar genuinamente limitada por su capacidad máxima de procesamiento.

Lección 23 — Detección de GPUs infrautilizadas

Configurar alertas basadas en un umbral de utilización de GPU sostenido bajo durante un período prolongado, usando las métricas ya recolectadas por herramientas como DCGM Exporter.

Si el patrón de infrautilización solo se descubre al revisar la factura del proveedor cloud al final del mes, ya se habría desperdiciado un mes completo de capacidad costosa sin aprovechar; detectarlo proactivamente mediante alertas en tiempo real permite corregir esa situación mucho antes, evitando que el desperdicio se acumule innecesariamente durante más tiempo.

Lección 24 — Autoscaling de nodos con GPU (Cluster Autoscaler)

Los nodos con GPU suelen ser significativamente más costosos y con tiempos de aprovisionamiento potencialmente más largos, por lo que las políticas de escalado deben balancear cuidadosamente la disponibilidad contra el costo de mantener capacidad GPU ociosa.

Si un nodo con GPU tarda más tiempo en aprovisionarse y estar completamente listo para usarse que un nodo estándar, una política de autoscaling que no considere ese retraso adicional podría dejar peticiones esperando más tiempo del esperado durante un pico repentino de demanda, mientras el nuevo nodo GPU todavía se está preparando.

Lección 25 — Optimización de costos de GPU en cloud

Usar instancias 'spot' o 'preemptible' significativamente más baratas que las instancias bajo demanda, aceptando el riesgo de que puedan ser interrumpidas, combinado con checkpointing para poder reanudar el trabajo tras una interrupción.

Sin un mecanismo de checkpointing que guarde periódicamente el progreso del entrenamiento, una interrupción inesperada de la instancia spot haría perder todo el trabajo acumulado hasta ese momento,

anulando por completo el ahorro de costo obtenido; el checkpointing es lo que convierte ese riesgo de interrupción en algo manejable y realmente viable.

Módulo 6: Producción avanzada

Lección 26 — GPUs en cargas de inferencia vs entrenamiento

El entrenamiento típicamente satura la GPU de forma sostenida durante períodos largos, mientras la inferencia suele tener picos de uso más cortos e intermitentes según llegan las peticiones, siendo mejor candidata para compartición de GPU.

Si el entrenamiento satura consistentemente la GPU durante todo su período de ejecución, dedicarla por completo aprovecha esa capacidad al máximo sin desperdicio; en cambio, una carga de inferencia con picos intermitentes dejaría la GPU ociosa durante los períodos de baja demanda si se le dedicara en exclusiva, siendo más eficiente compartirla con otras cargas.

Lección 27 — Checkpointing y tolerancia a fallos en jobs con GPU

Permite reanudar el entrenamiento desde el último punto guardado en caso de una interrupción (fallo de hardware, mantenimiento del nodo, instancia spot revocada), evitando perder todo el progreso computacional ya invertido desde el inicio.

El tiempo adicional que toma guardar un checkpoint periódicamente es mínimo comparado con el costo real de tener que reiniciar días o semanas completas de entrenamiento desde cero (con el gasto de GPU ya invertido completamente perdido) si ocurriera una interrupción inesperada sin ningún punto de recuperación guardado.

Lección 28 — Seguridad de GPU — aislamiento entre workloads

Sin un mecanismo de aislamiento real como MIG, datos residuales en la memoria de la GPU de una carga anterior podrían potencialmente ser accesibles o inferibles por la siguiente carga que use esa misma GPU, dependiendo de las protecciones implementadas.

En un entorno donde clientes externos no confían entre sí, la falta de partición física real de memoria en time-slicing representa un riesgo de seguridad más significativo que en un entorno interno controlado; MIG, al ofrecer aislamiento físico real, reduce ese riesgo de forma mucho más sólida y apropiada para ese contexto específico de mayor sensibilidad de seguridad.

Lección 29 — Troubleshooting común de GPUs en Kubernetes

Los eventos del Pod (kubect! describe pod) para ver el mensaje específico del scheduler explicando por qué no pudo encontrar un nodo candidato adecuado con la capacidad GPU solicitada.

El estado 'Pending' por sí solo solo indica que el Pod no ha sido programado todavía, sin explicar por qué; el mensaje específico del evento del scheduler (como 'capacidad de GPU insuficiente') revela la causa exacta y concreta del problema, dirigiendo el diagnóstico directamente hacia la solución correcta sin tener que adivinar entre múltiples causas posibles.

Lección 30 — Proyecto final: plataforma de GPUs en Kubernetes lista para producción

GPU Operator con NFD configurado + taints/tolerations correctos + estrategia de compartición adecuada (MIG/time-slicing según el caso) + monitoreo con DCGM Exporter + checkpointing para cargas largas + autoscaling con límites definidos.

Intenta mencionar al menos dos pilares: GPU Operator bien configurado, estrategia de compartición adecuada, monitoreo con DCGM Exporter o checkpointing para cargas largas.