國立清華大學
NATIONAL TSING HUA UNIVERSITY

KubeShare: A Framework to Manage GPUs as First-Class and Shared Resources in Container Cloud

Ting-An Yeh, Hung-Hsin Chen, Jerry Chou
National Tsing Hua University
Hsinchu, Taiwan R.O.C.

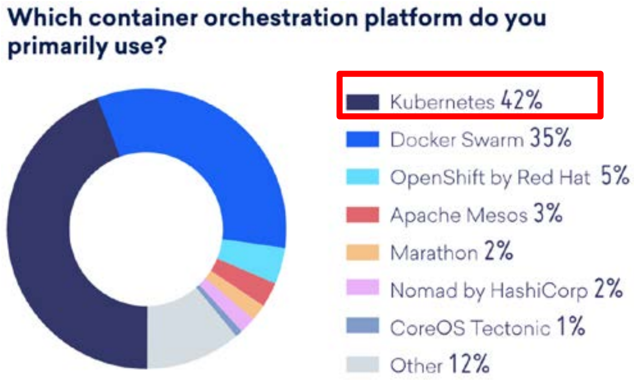
HPDC ’20, June 23–26, 2020, Stockholm, Sweden

Outline

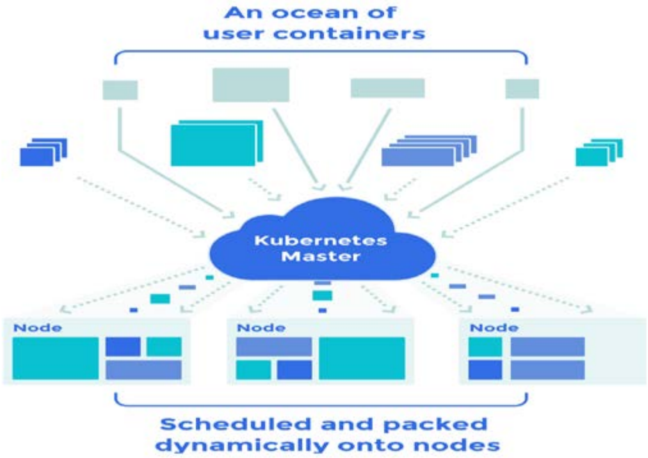
- Motivations & Objectives
  - Introduction of Kubernetes & GPU
  - Why GPU sharing & first clas scheduling is important
  - Our contributions
- KubeShare Design & Implementation
- Experimental Evaluations
- Conclusions

Container Cloud

- **Container** offers many advanvatages over **virtual machine**
  - Fast launch time
  - Higher deployment density
  - Less performance degradation
- **Kubernetes** is the primary platform to build container cloud
  - **Hide infrastructure details from developers**
  - Provide several automation features:
    - auto-scalability
    - self-healing
    - rolling update and rollback
    - service discovery and load balancing
  - ***Pod*** (a set of containers) is the basic execution unit



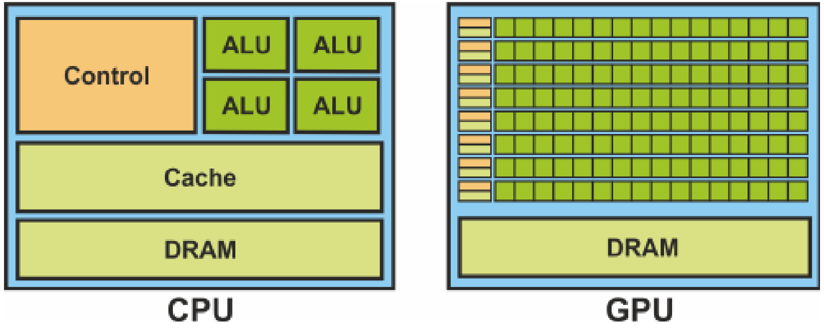
A quarterly report on developer trends in the cloud by Digital Ocean



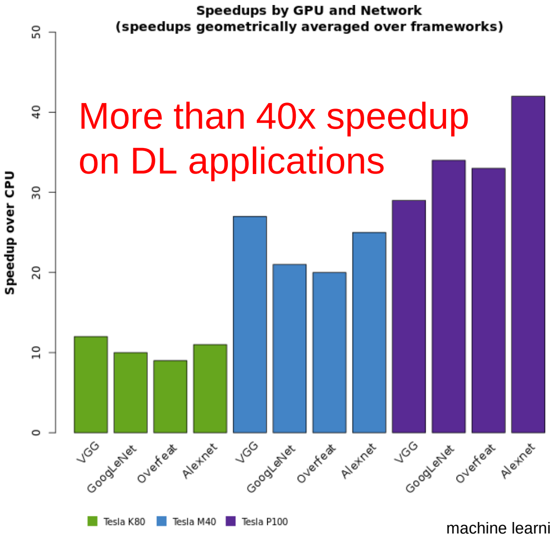
kubernetes container cluster
https://devopscube.com/docker-container-clustering-tools/

Graphics Processing Unit

- **GPUs** provide **tremendous throughput** powered by **massive parallelism**
- **Significant performance accelerations** are shown in many applications, especially for **deep learning** and **scientific computing workload**
- Widely installed in world's fast supercomputers and clouds



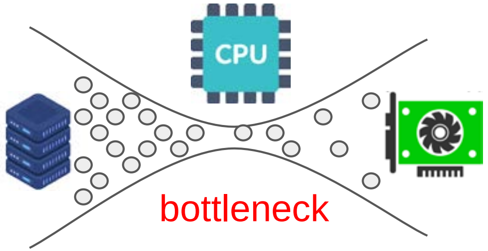
cpu vs gpu
https://www.researchgate.net/figure/Comparison-of-CPU-versus-GPU-architecture_fig2_231167191

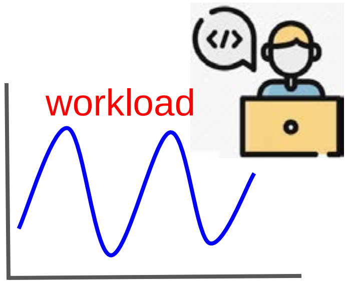


machine learning training GPU benchmark
https://www.microway.com/hpc-tech-tips/deep-learning-benchmarks-nvidia-tesla-p100-16gb-pcie-tesla-k80-tesla-m40-gpus/

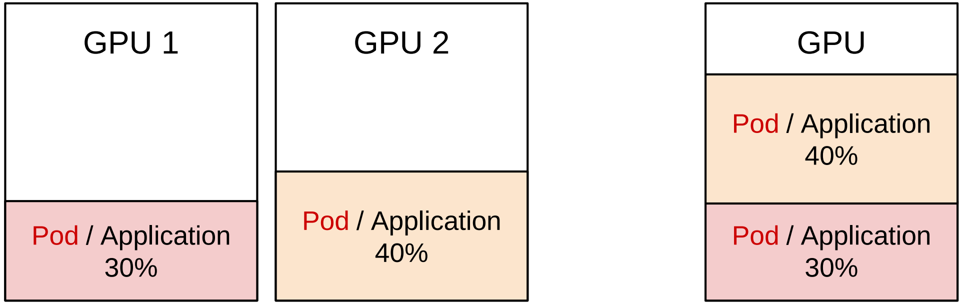
Motivations of GPU Sharing

- But GPUs are **expensive**, and often **under-utilized**
  - Code developing phase
  - Off-peak service hours
  - Limited data transfer bandwidth
  - Bounded by host/cpu performance





- **GPU sharing can effiectively maximize GPU utilization**



Dedicated GPU Allocation

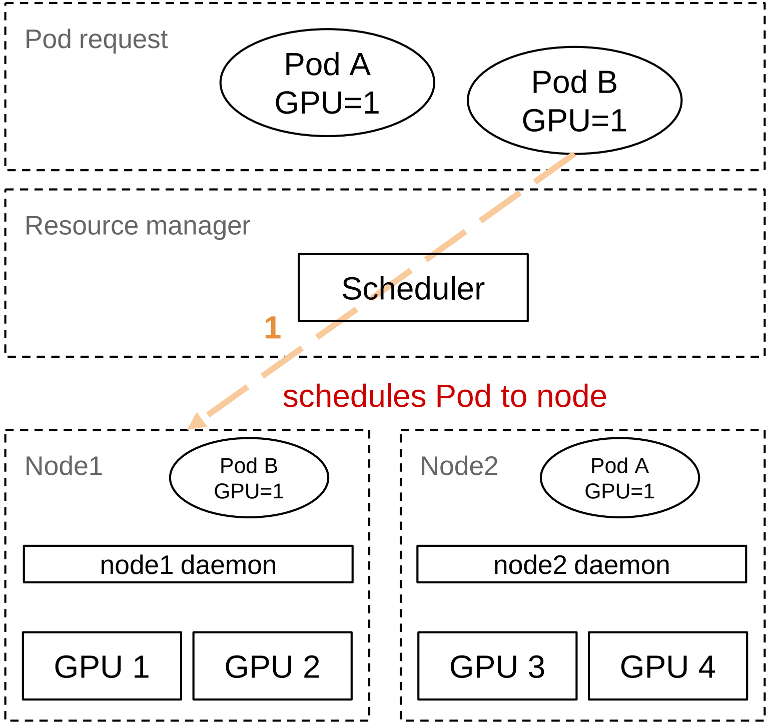
GPU Sharing

Challenges of GPU Sharing

- **CUDA compute capability 2.0+** support task parallelism, but **only from single process/application context**
  - No explicit resource management control from applications and host
  - Resoruce oversubscription cause performance degradation and program failure
- **Recent research work** on GPU sharing aims to improve GPU throughput and fairness, but **not from user resource allocation** aspect
  - FLEP, GPUShare, Disengaged Scheduling, ConVGPU, TimeGraph, ...
- **Kubernetes** has **no GPU sharing & isolation**
  - GPU device can only be dedicatedly assigned to a *Pod*
- GPUs are **not first class schedulable resources** in **Kubernetes**
  - User cannot request for a specific GPU device from Kubernetes

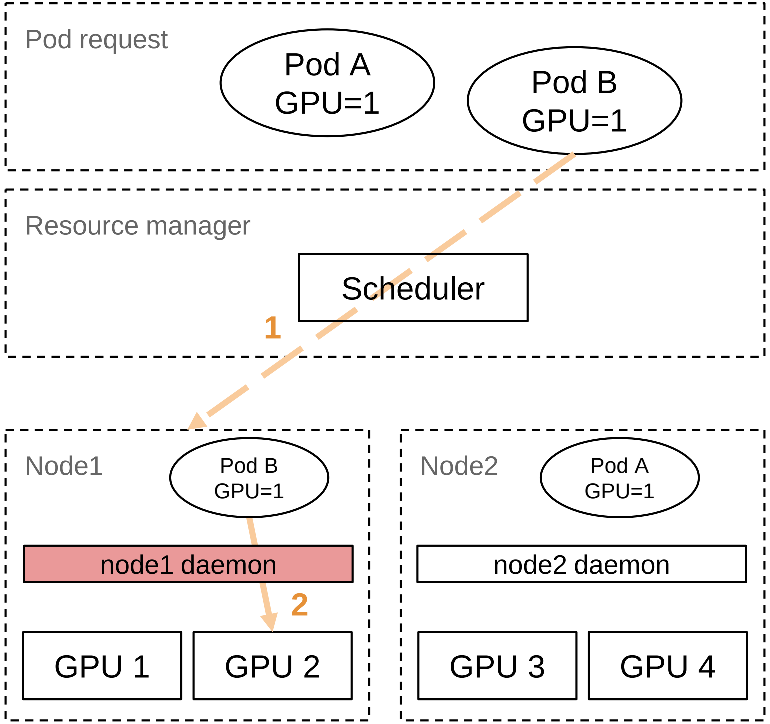
First Class Schedulable Entity

- What is a first class entity?
  - **Users or resource manager (scheduler) can request specific GPU devices for their pods**
  - The assignment is done implicitly by a node deamon (kubelet) in Kubernetes
- **Implicit** and **late** binding in Kubernetes
  - **Resource manager schedules requests at node level**.
  - Pod to GPU **binding is delayed after scheduling decision was made**



First Class Schedulable Entity

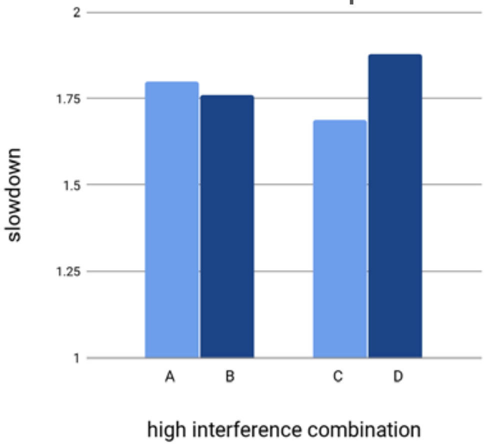
- What is a first class entity?
  - **Users or resource manager (scheduler) can request specific GPU devices for their pods**
  - The assignment is done implicitly by a node deamon (kubelet) in Kubernetes.
- **Implicit** and **late** binding in Kubernetes
  - **Resource manager schedules requests at node level.**
  - Pod to GPU **binding is delayed after scheduling decision was made**



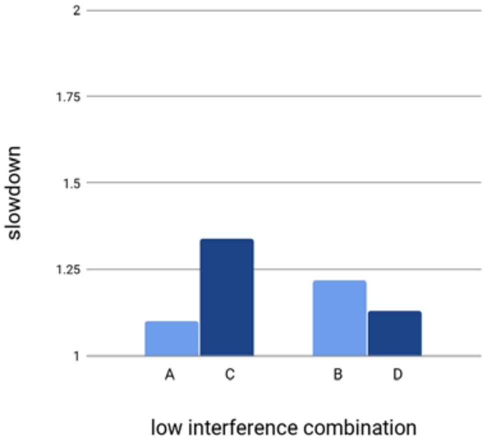
node daemon choose the GPU to bind

First Class Schedulable Entity

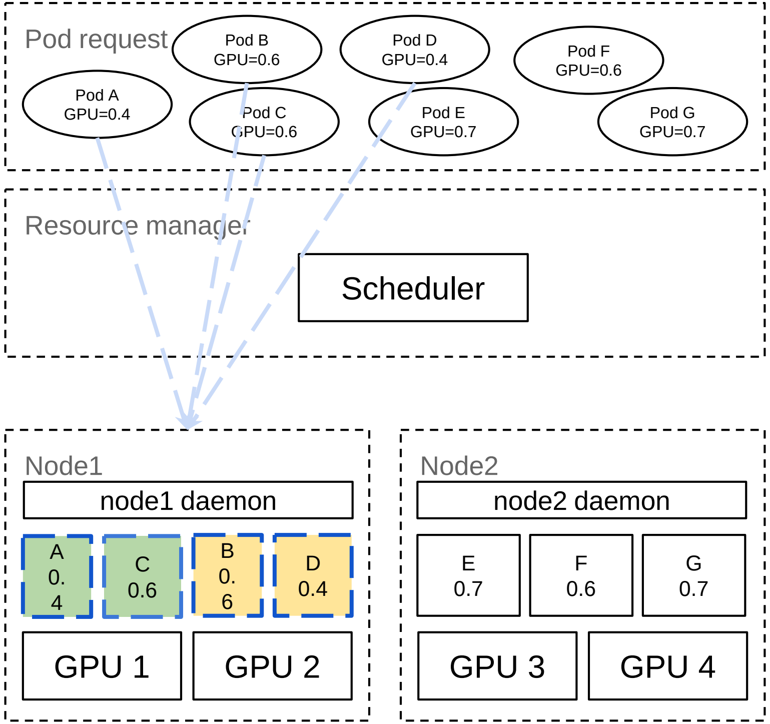
- Why first class is important?
  - **Performance interference problem**
    - Pods have different resource usage patterns



More resource contention between A&B or C&D

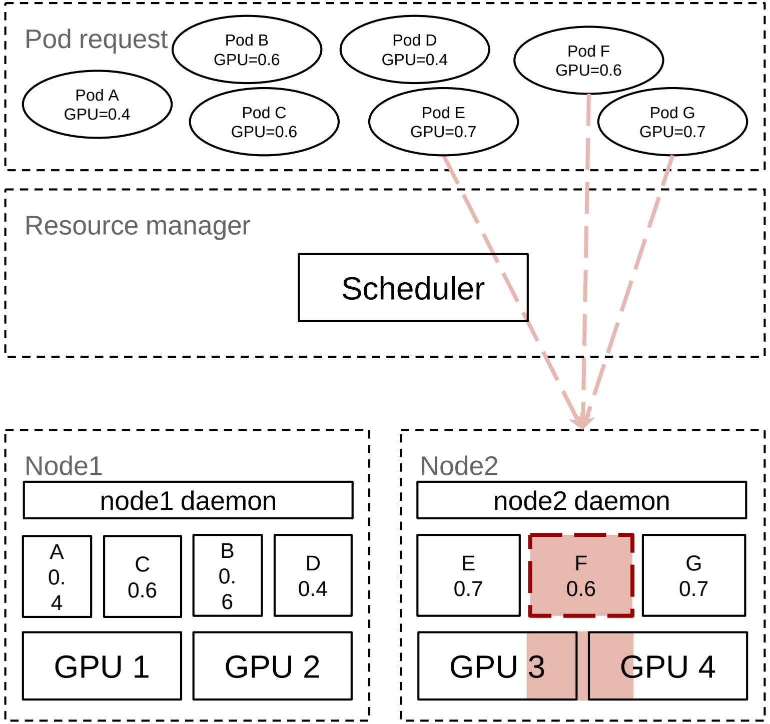


Less resource contention between A&C or D&B



First Class Schedulable Entity

- Why first class is important?
  - **Performance interference problem**
  - **Resource fragmentation problem**
    - **GPU allocation is indivisble between devices for a pod**
    - **Scheduler is only aware of the aggregated node capacity**



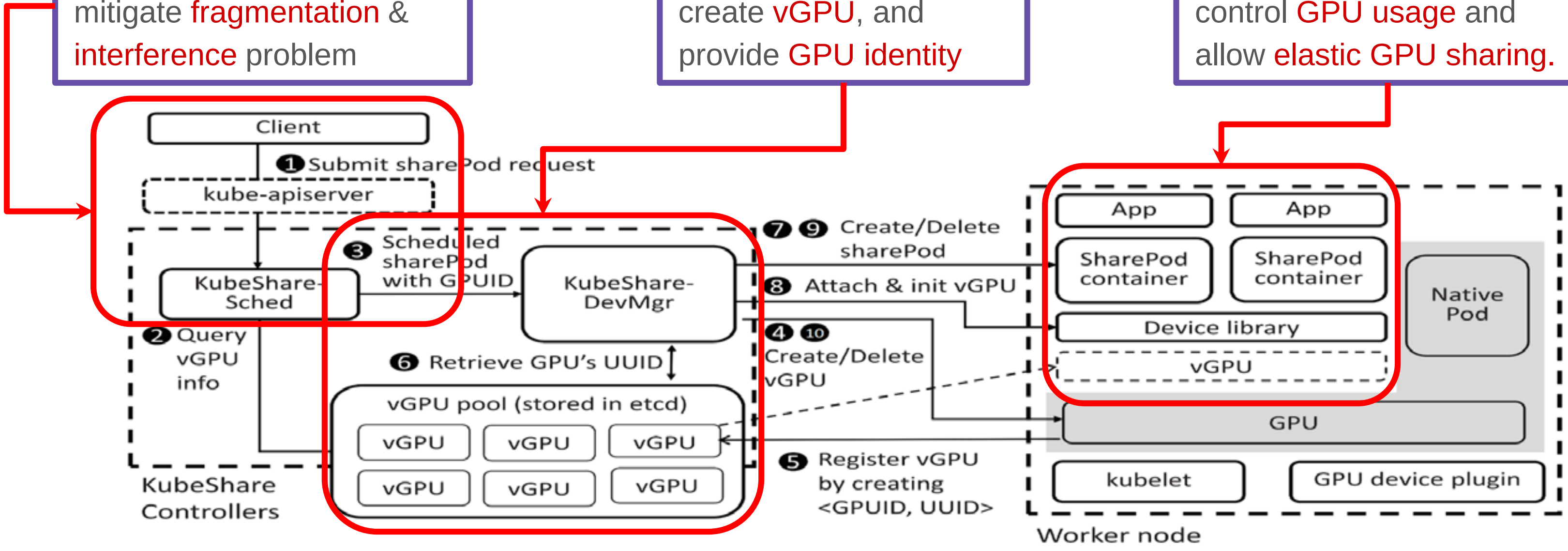
KubeShare Contributions

- Objectives: Enable **GPU sharing** in Kubernetes, and provide ***first class GPU scheduling*** to address ***utilization***, ***fragmentation*** and ***interference*** problems

KubeShare-Sched: mitigate fragmentation & interference problem

KubeShare-DevMgr: create vGPU, and provide GPU identity

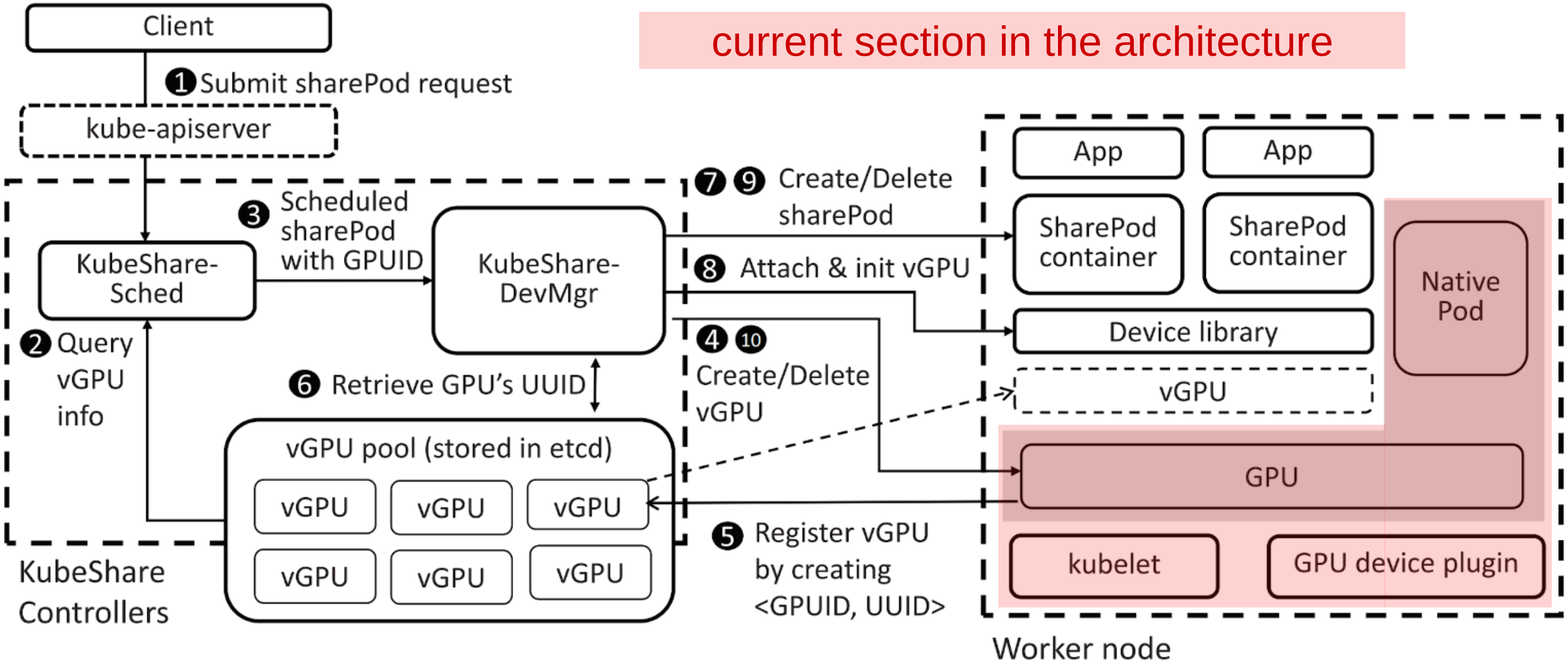
vGPU Device Library: control GPU usage and allow elastic GPU sharing.



Outline

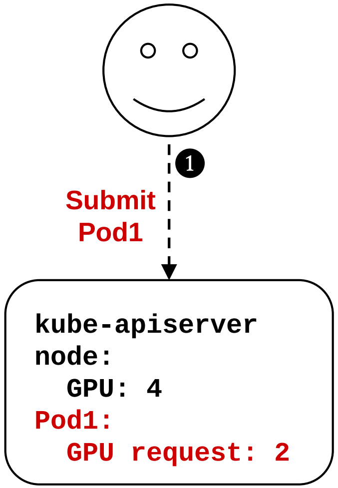
- Motivations & Objectives
- KubeShare Design & Implementation
  - Device Plugin Framework
  - vGPU creation & management
  - Shared GPU pod requirement & scheduling
  - GPU resource control & elastic allocation
- Experimental Evaluations
- Conclusions

Background: Kubernetes Device Plugin Framework



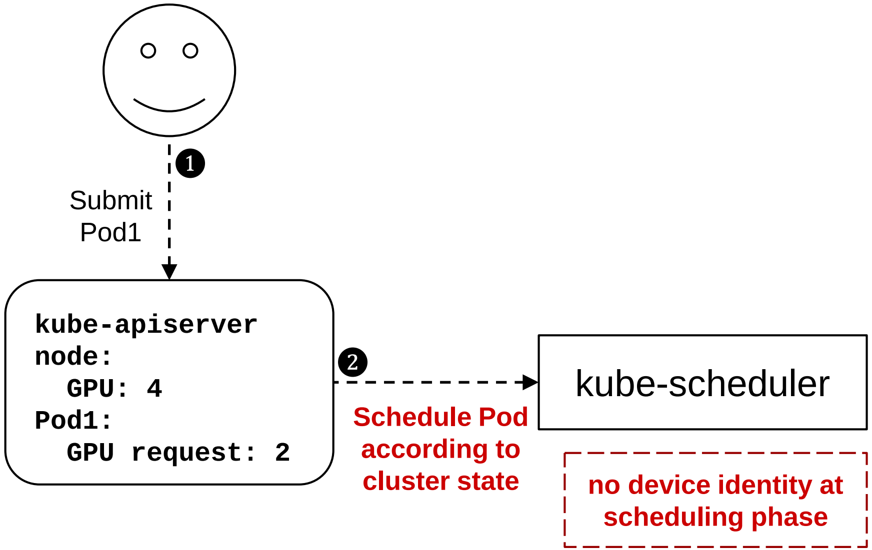
Background: Kubernetes Device Plugin Framework

- Device plugin framework: support custom computing resource (GPU, FPGA).



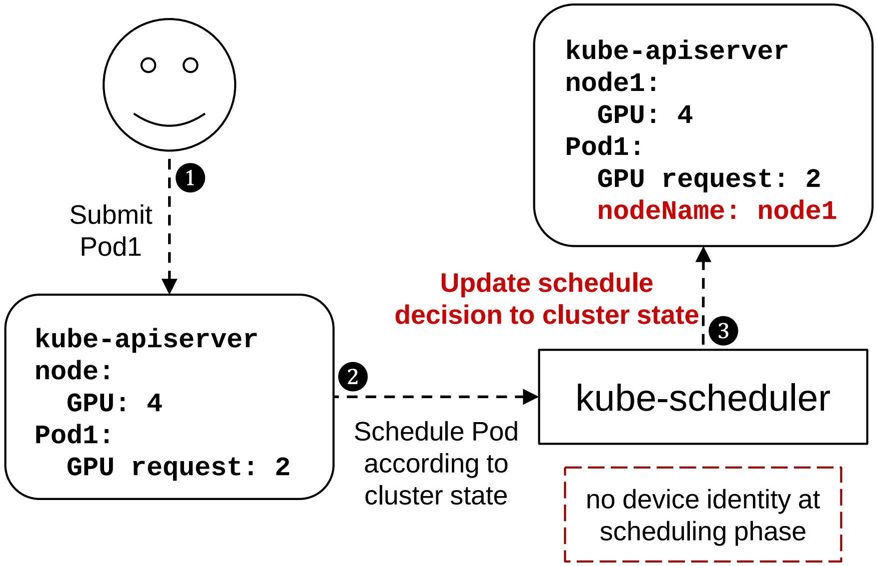
Background: Kubernetes Device Plugin Framework

- Device plugin framework: support custom computing resource (GPU, FPGA).



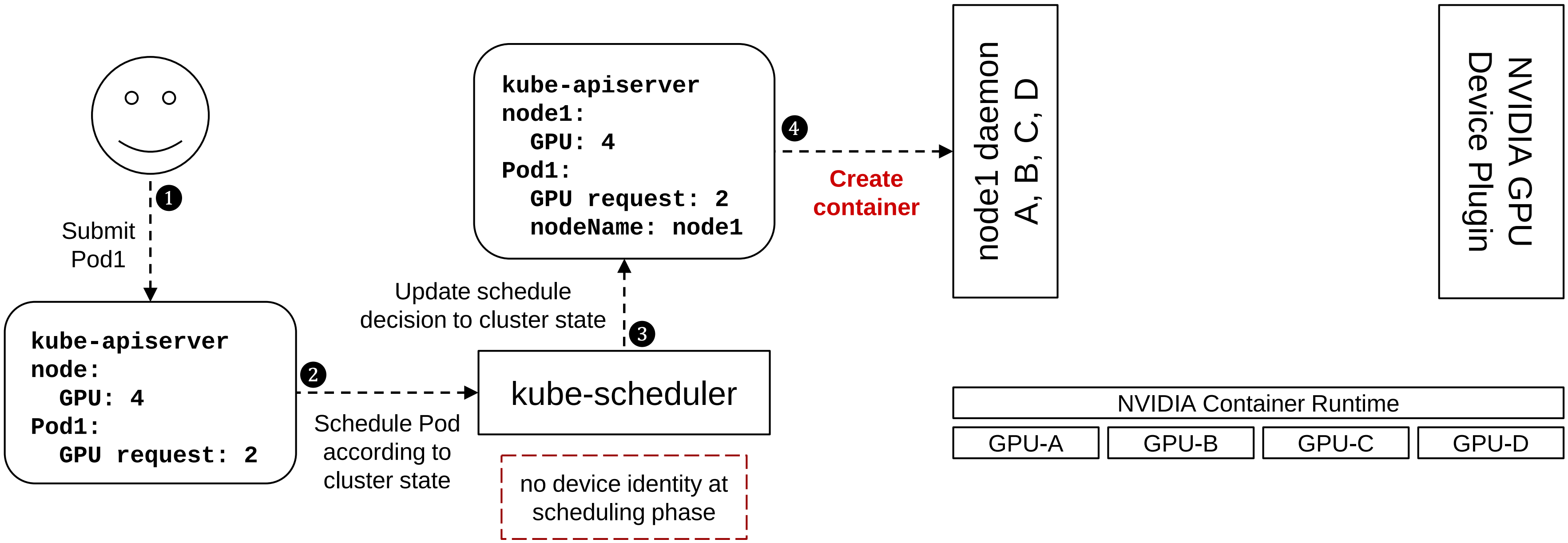
Background: Kubernetes Device Plugin Framework

- Device plugin framework: support custom computing resource (GPU, FPGA).



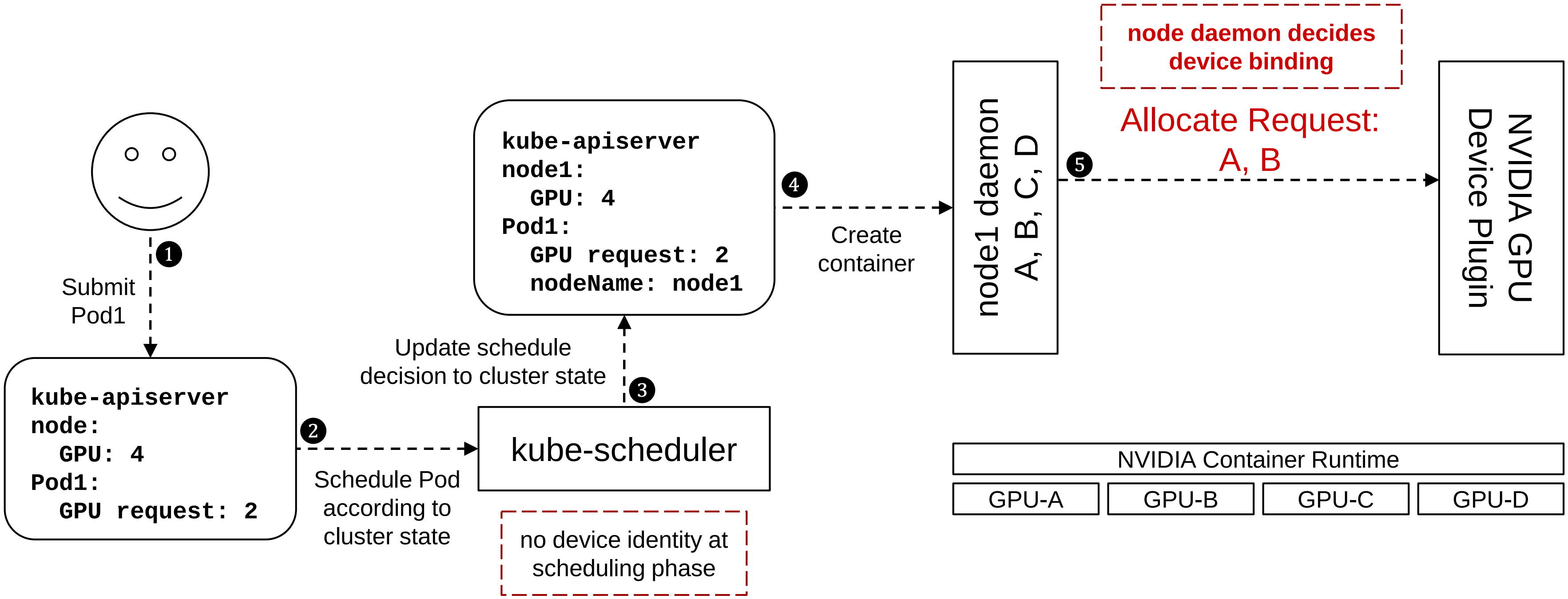
Background: Kubernetes Device Plugin Framework

- Device plugin framework: support custom computing resource (GPU, FPGA).



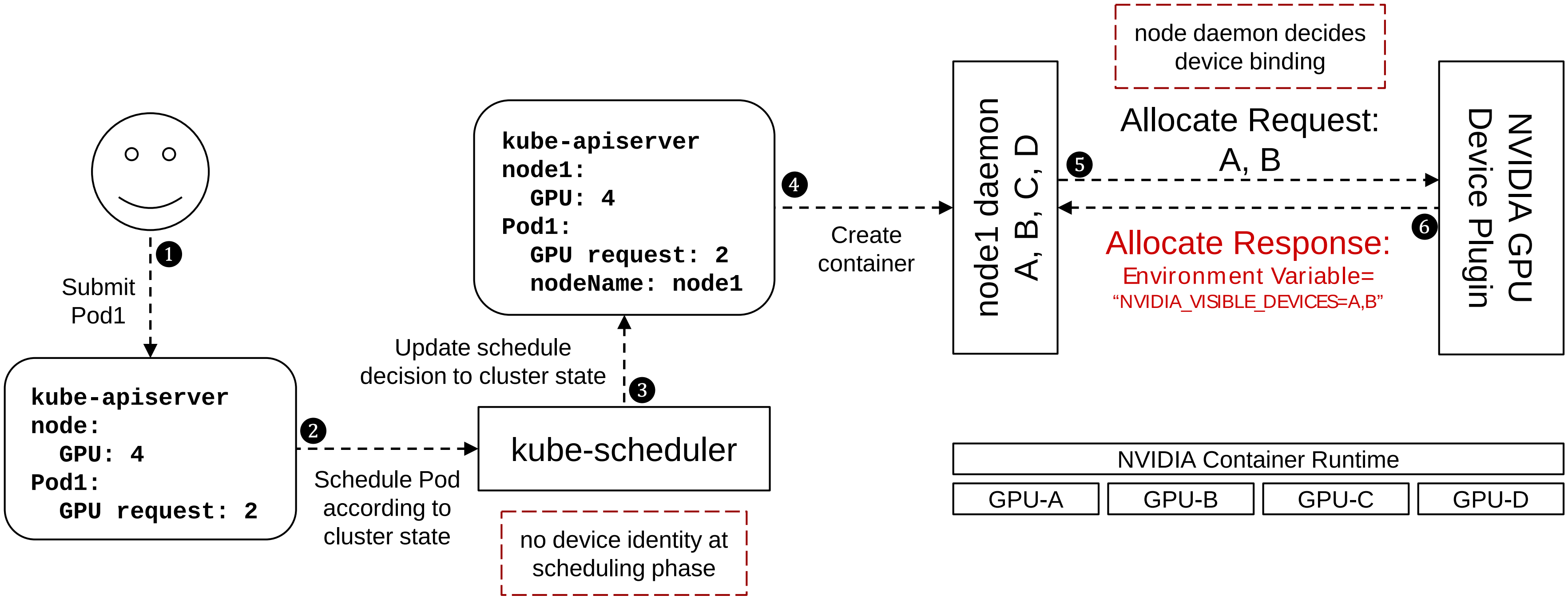
Background: Kubernetes Device Plugin Framework

- Device plugin framework: support custom computing resource (GPU, FPGA).



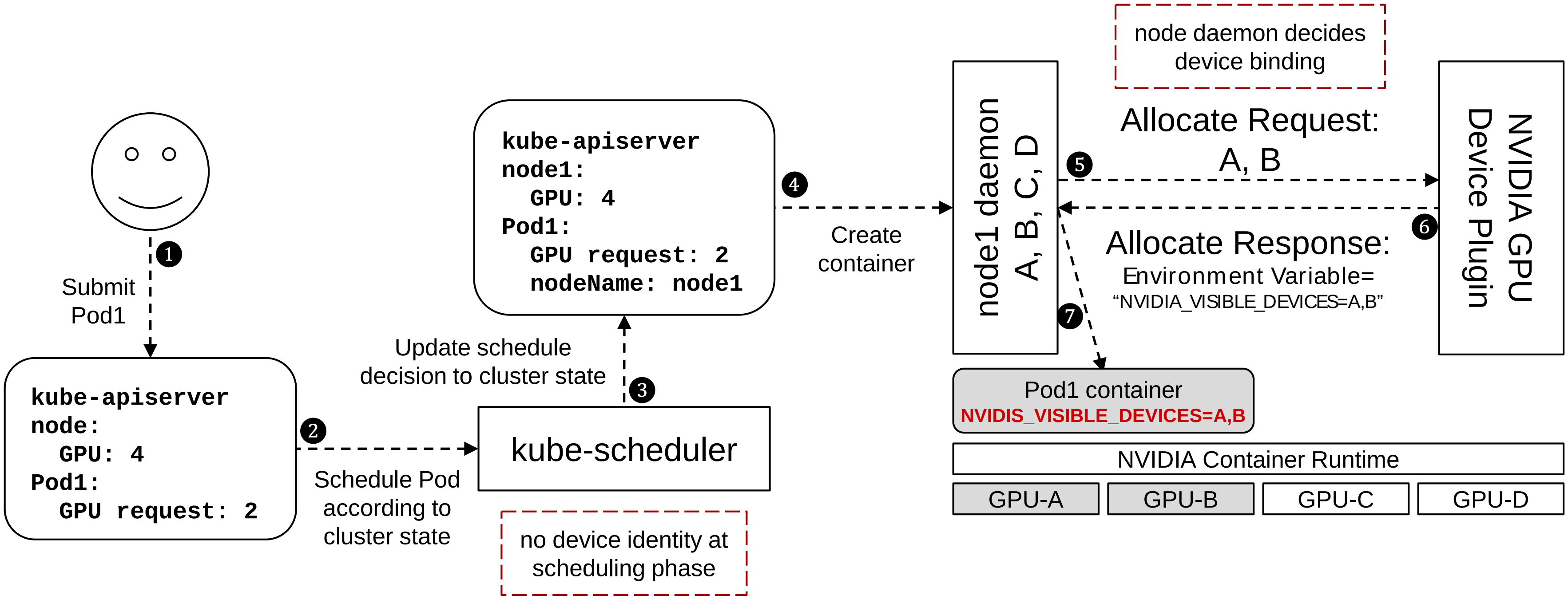
Background: Kubernetes Device Plugin Framework

- Device plugin framework: support custom computing resource (GPU, FPGA).



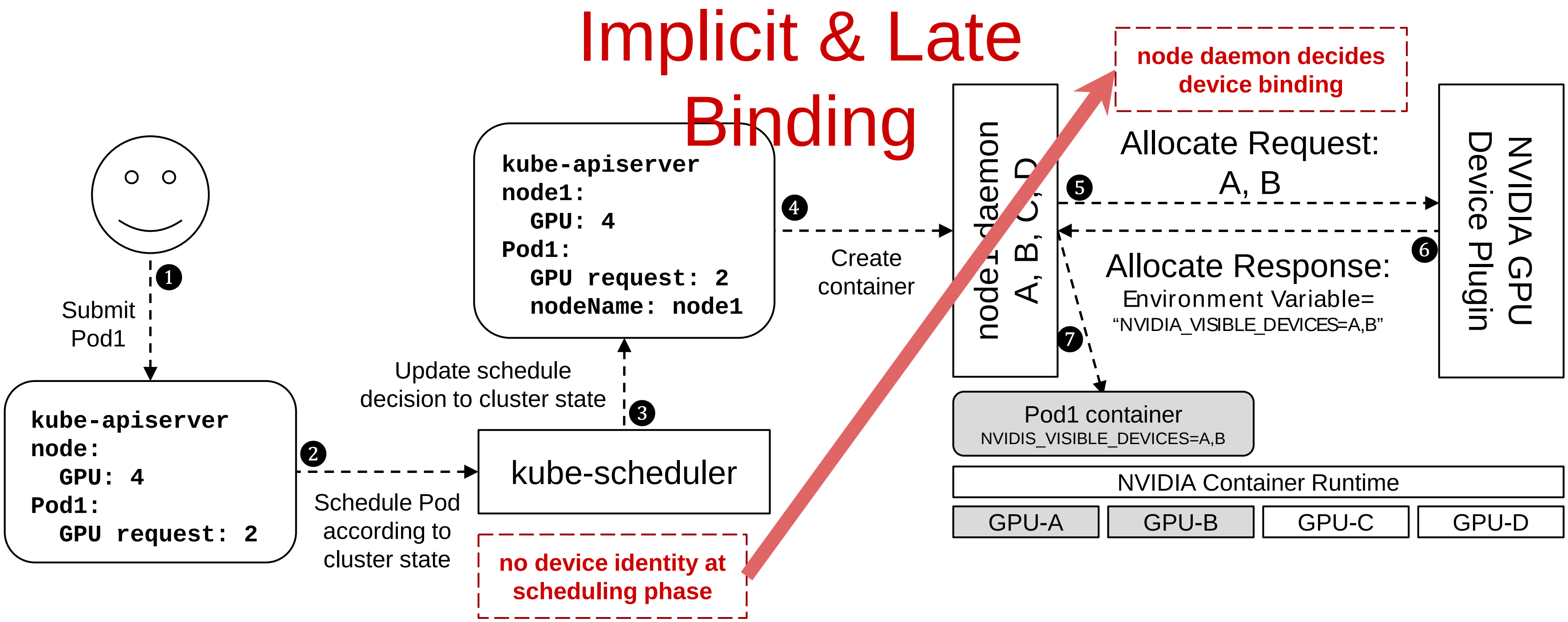
Background: Kubernetes Device Plugin Framework

- Device plugin framework: support custom computing resource (GPU, FPGA).



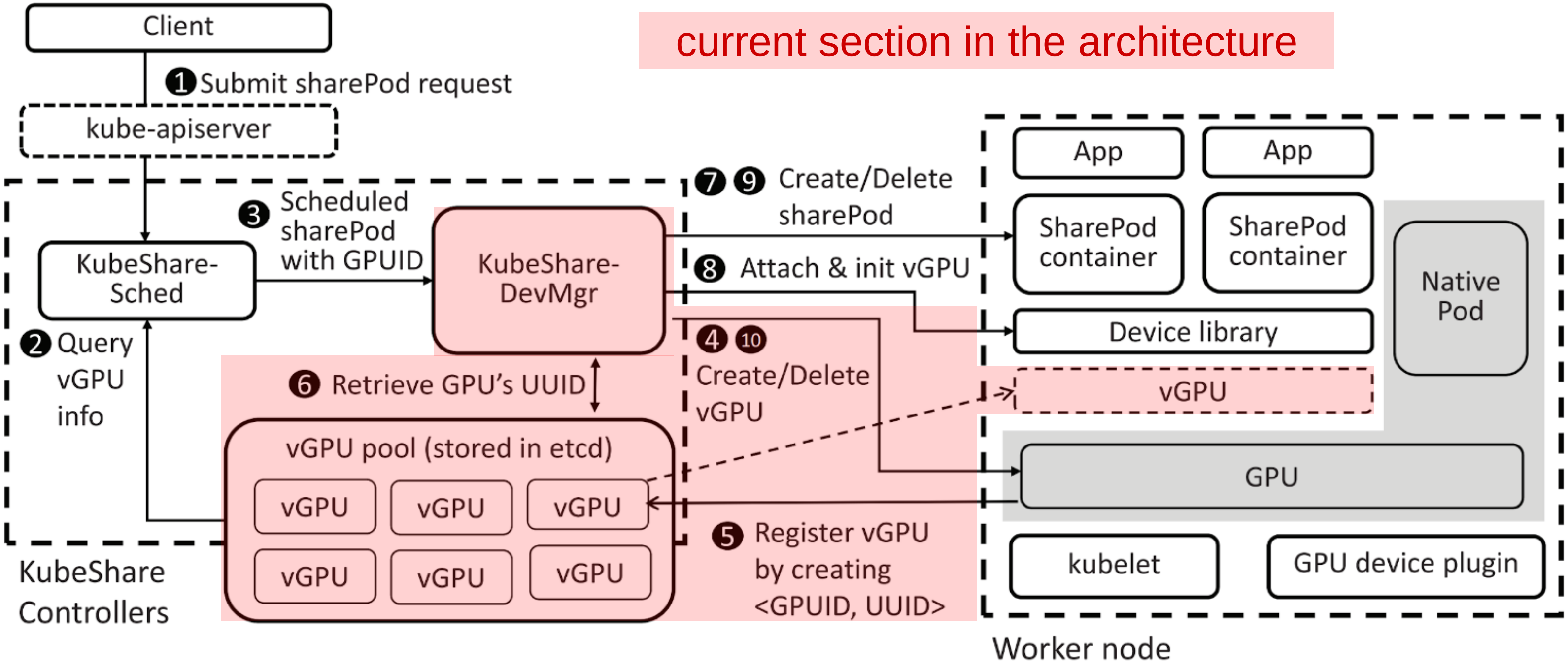
Background: Kubernetes Device Plugin Framework

- Device plugin framework: support custom computing resource (GPU, FPGA).



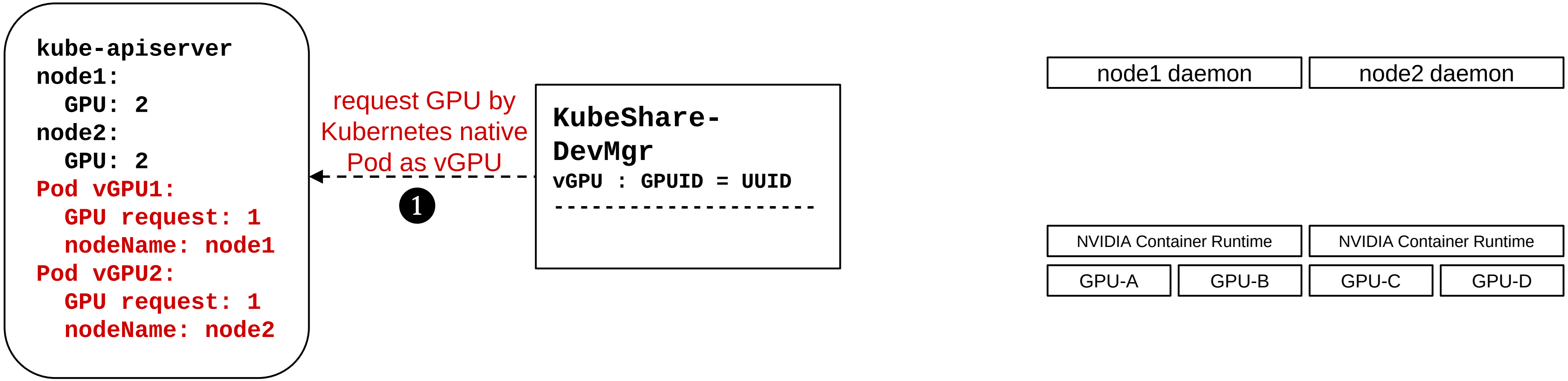
KubeShare-DevMgr: vGPU Creation

current section in the architecture



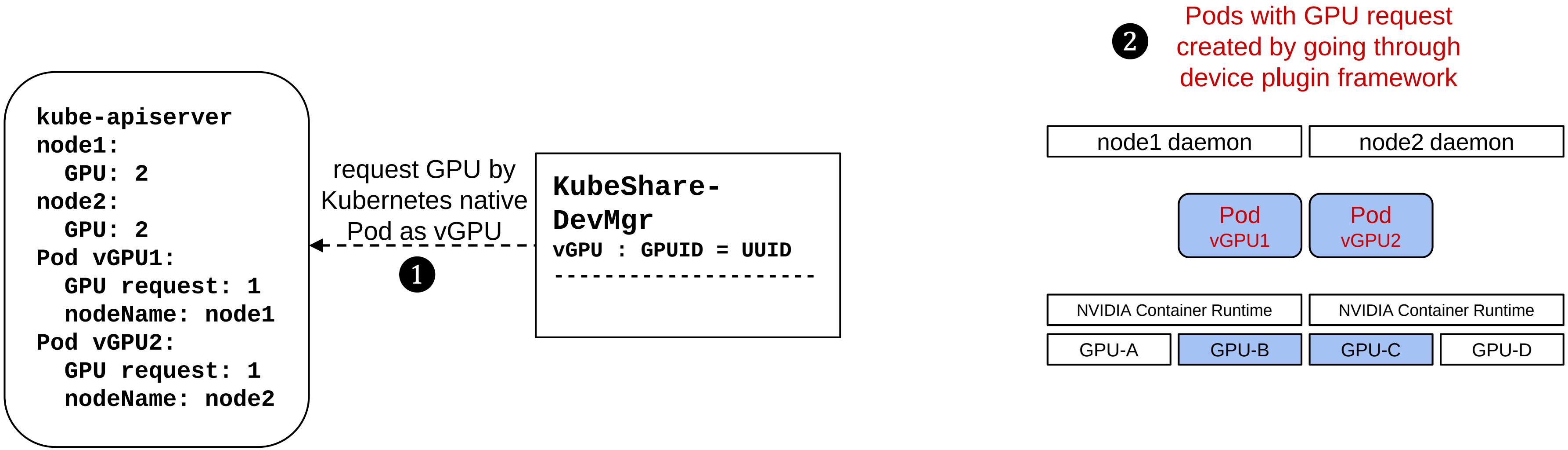
vGPU Creation

- KubeShare-DevMgr creates vGPU and provides GPU identity
  - **vGPU** is a logic GPU resoruce entity that can be shared and identified in KubeShare
  - Different from native GPU, vGPU can be fractional allocated by users
  - vGPU is also created by a pod



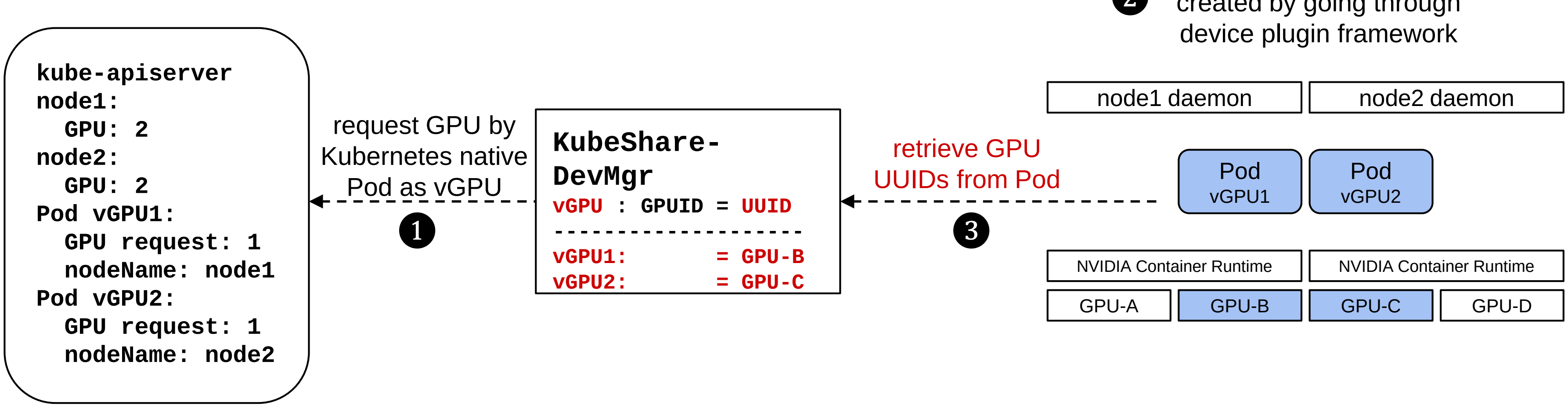
vGPU Creation

- KubeShare-DevMgr creates vGPU and provides GPU identity
    - GPU is acquired from Kubernetes through the **device plugin framework**



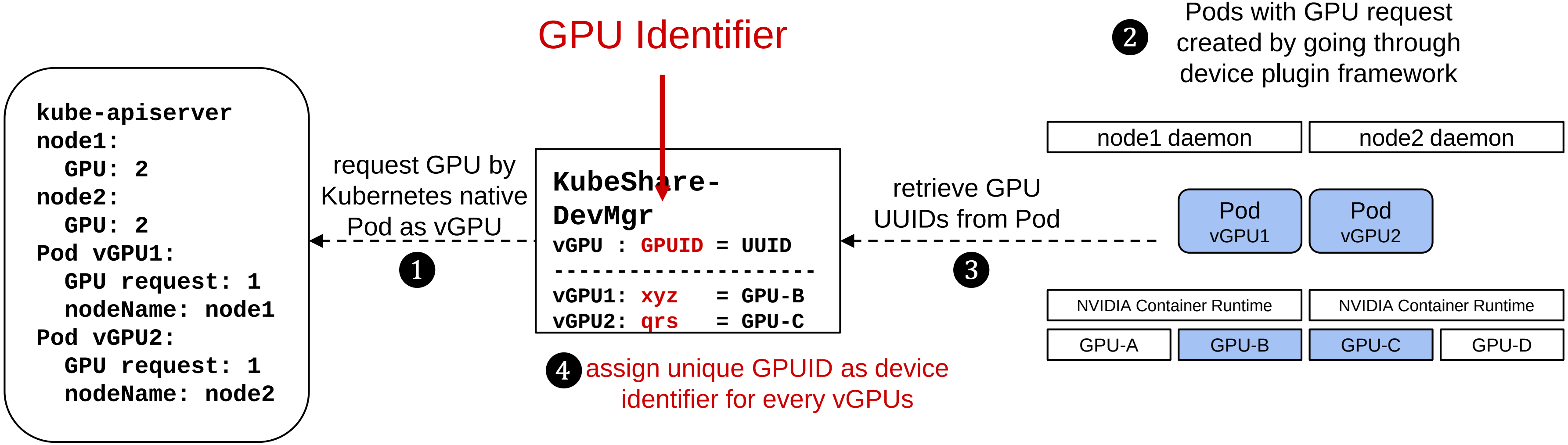
vGPU Creation

- KubeShare-DevMgr creates vGPU and provides GPU identity
    - **GPUID** is the identifier of a vGPU assigned by **KubeShare**
    - **UUID** is the identifier of a GPU returns by **device plugin**



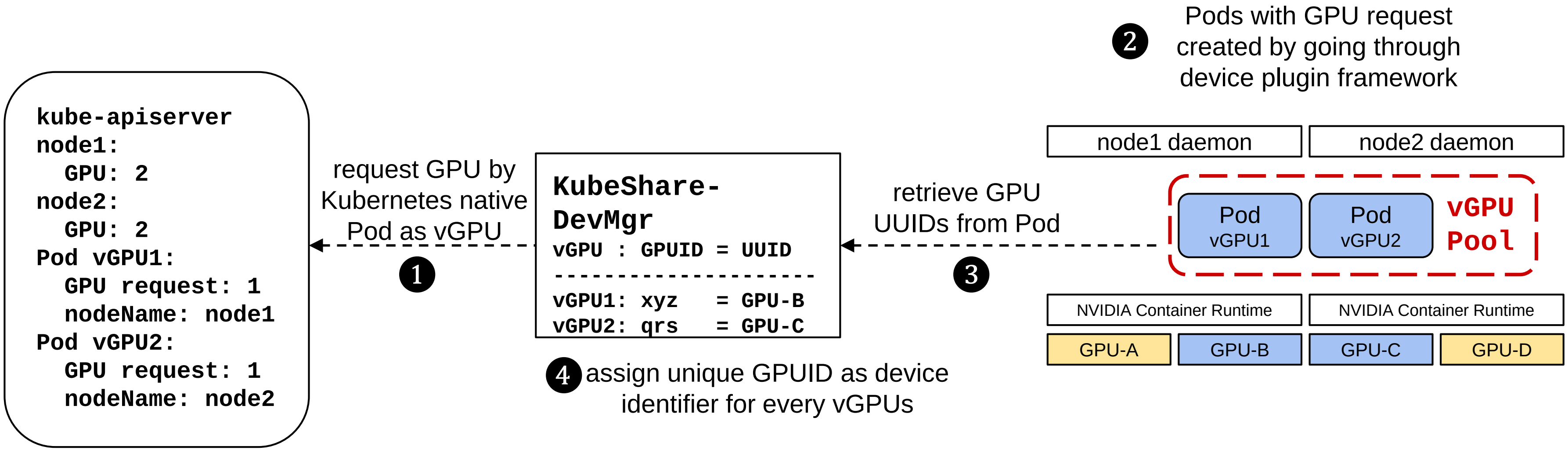
vGPU Creation

- KubeShare-DevMgr creates vGPU and provides GPU identity
  - The GPUID can be used at scheduling phase to co-locate pods on a new created vGPU

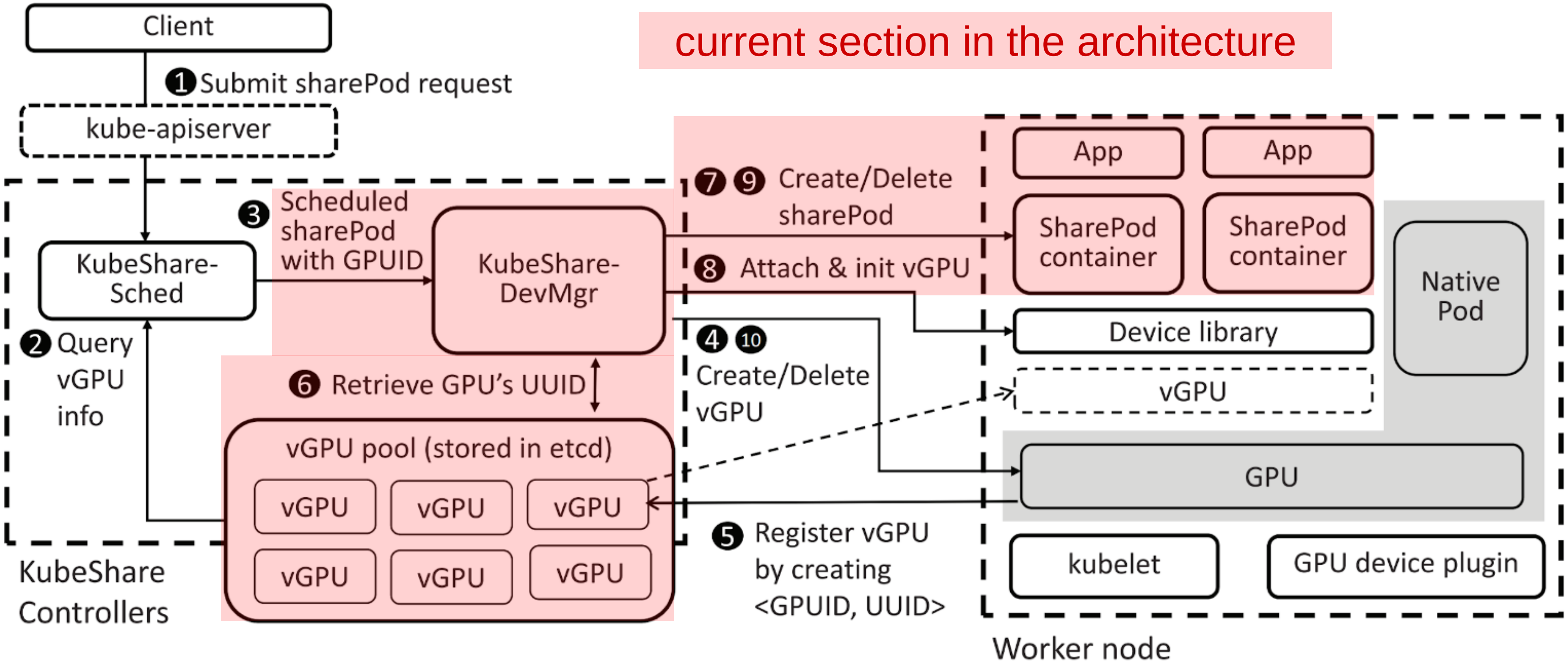


vGPU Creation

- KubeShare-DevMgr creates vGPU and provides GPU identity
  - The group of vGPUs managed by KubeShare is called **vGPU Pool**



KubeShare-DevMgr: SharePod Creation



SharePod Creation

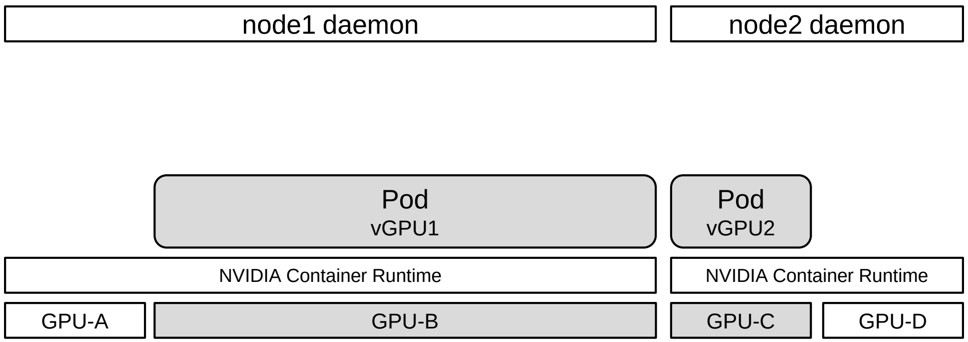
- Users request a shared GPU by creating a SharePod with GPUID

A SharePod == A pod that attaches a sharedGPU (vGPU)

```
SharePod1:
  gpu_req: 0.4
  node: node1
  GPUID: xyz
SharePod2:
  gpu_req: 0.6
  node: node1
  GPUID: xyz
```

1 submit SharePod with GPUID "**xyz**"

```
KubeShare-
DevMgr
vGPU : GPUID = UUID
---------------------
vGPU1: xyz   = GPU-B
vGPU2: qrs   = GPU-C
```



SharePod Creation

- Users request a shared GPU by creating a SharePod with GPUID

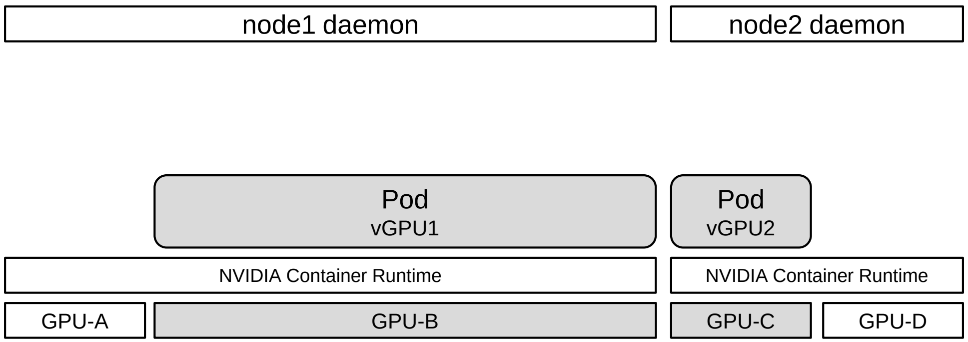
```
SharePod1:
  gpu_req: 0.4
  node: node1
  GPUID: xyz
SharePod2:
  gpu_req: 0.6
  node: node1
  GPUID: xyz
```

1 submit SharePod with GPUID “xyz”

```
KubeShare-
DevMgr
vGPU : GPUID = UUID
--------------------
vGPU1: xyz   = GPU-B
vGPU2: qrs   = GPU-C
```

2 create Pod with environment variable “NVIDIA_VISIBLE_DEVICES” to restrict GPU visibility in containers (without going through device plugin again)

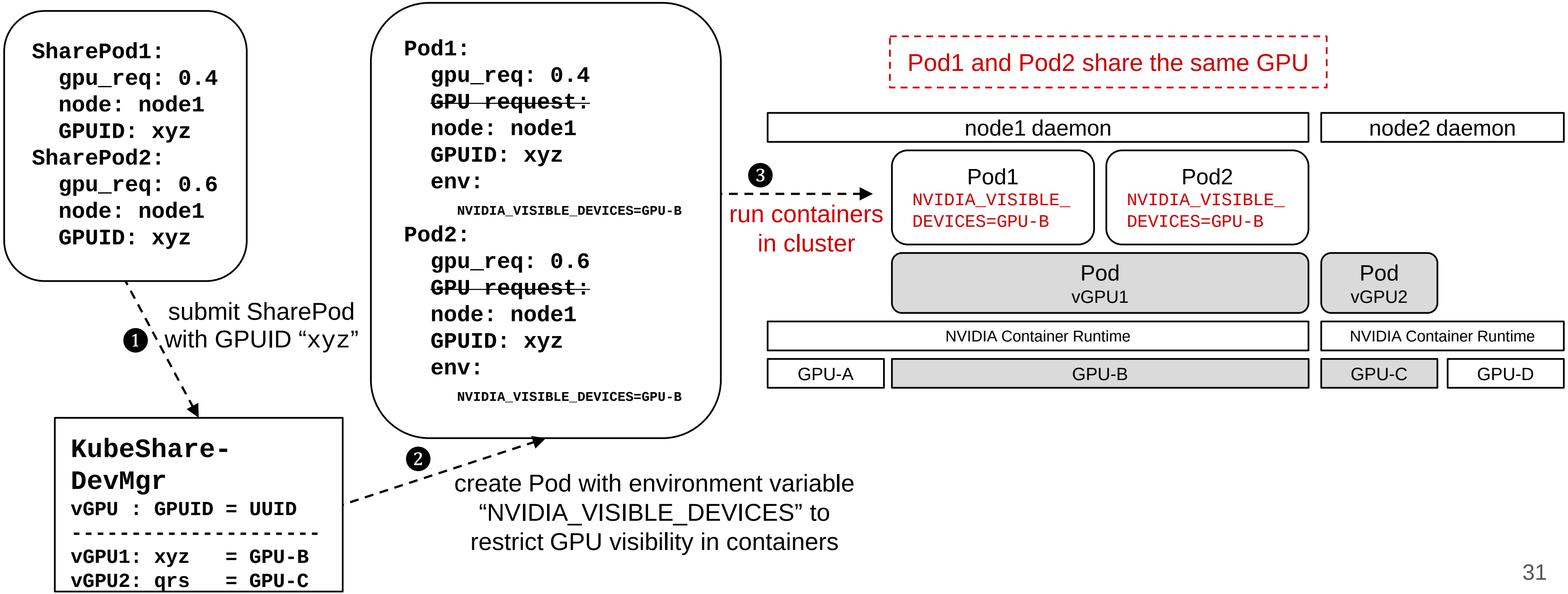
```
Pod1:
  gpu_req: 0.4
  GPU request:
  node: node1
  GPUID: xyz
  env:
    NVIDIA_VISIBLE_DEVICES=GPU-B
Pod2:
  gpu_req: 0.6
  GPU request:
  node: node1
  GPUID: xyz
  env:
    NVIDIA_VISIBLE_DEVICES=GPU-B
```



If the vGPU named “xyz” doesn’t exist, a new vGPU is created and assigned with the GPUID “xyz”.

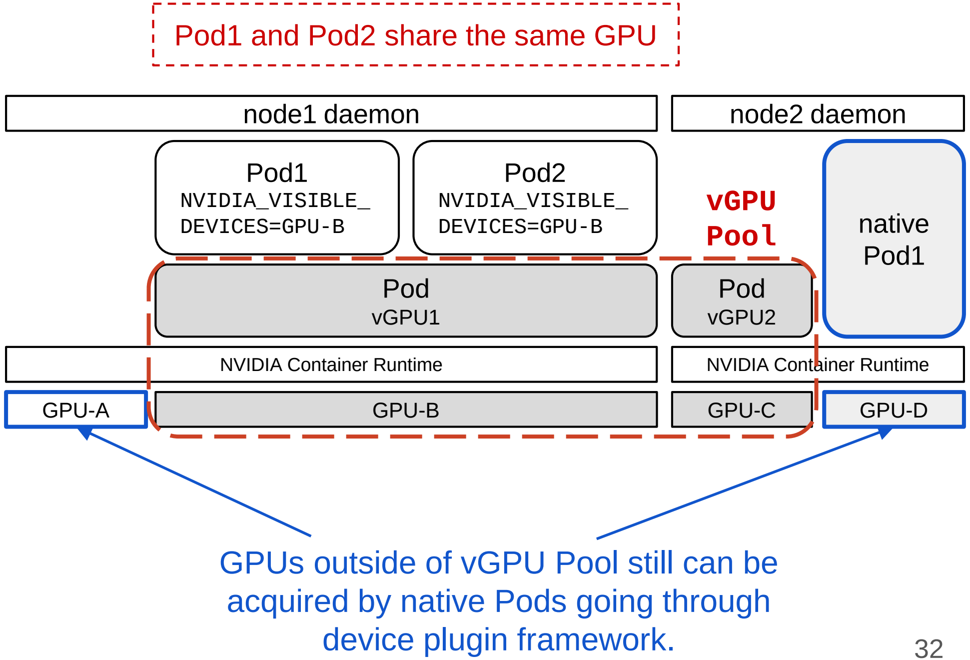
SharePod Creation

- Users request a shared GPU by creating a SharePod with GPUID



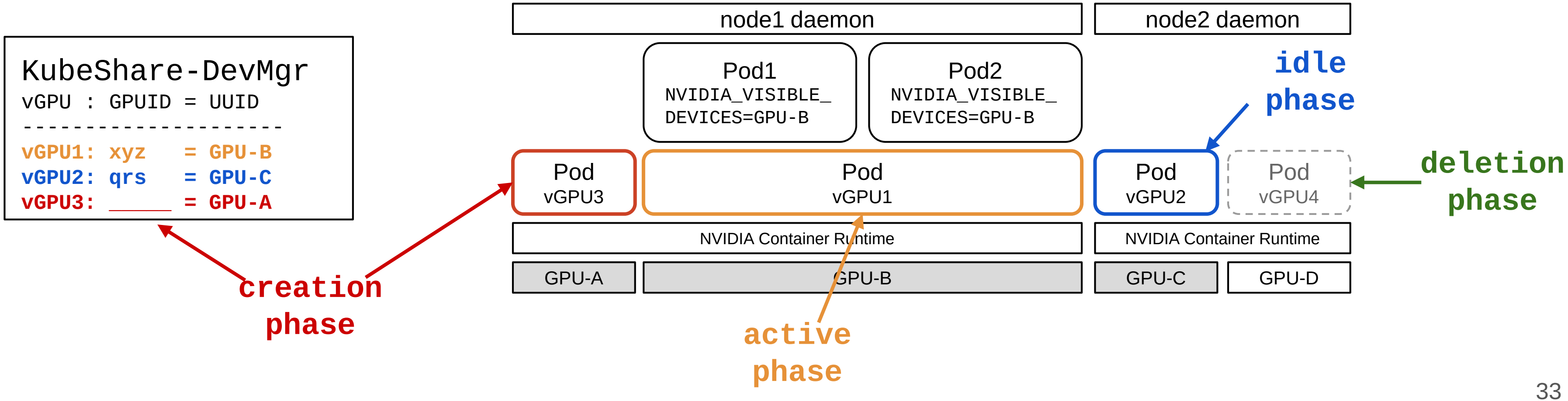
SharePod Creation

- **KubeShare is compatible with NVIDIA GPU device plugin management**
    - GPUs outsides vGPU pool still mamanged by NVIDIA GPU device plugin framework
    - Introduce minimum impact to the exisiting cluster management
    - Users can choose to attach GPUs on pods through KubeShare or the NVIDIA GPU device plugin



vGPU Lifecycle & vGPU Pool Management

- Phases of vGPU:
  - **creation**: a vGPU just allocated from Kubernetes and joins vGPU pool
  - **active**: a vGPU attached to one or multiple sharePods
  - **idle**: a vGPU without being attached to any sharePod
  - **deletion**: a vGPU released by KubeShare and leaves vGPU pool



vGPU Lifecycle & vGPU Pool Management

- Tradeoff between **Idle** and **Deletion**
  - **Our implementation choose on-demand because the creation overhead is limited**

Reservation based allocation
keep idle vGPU to handle future request

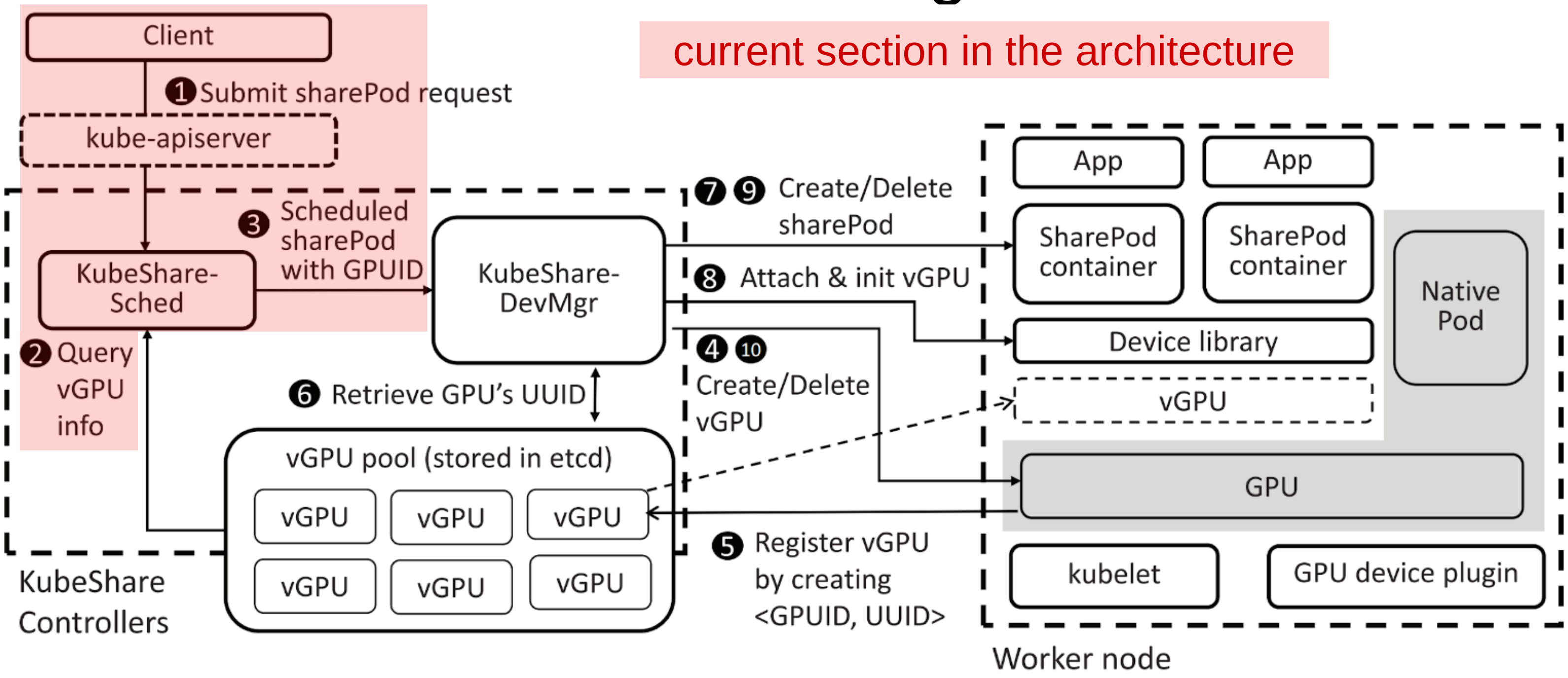
On-demand based allocation
delete idle vGPU immediately

Idle — **vGPU** — **Deletion**

Idle:
- Reduce pod creation time
- GPUs in vGPU Pool cannot be utilized by native Pod, which may result in lower resource utilization

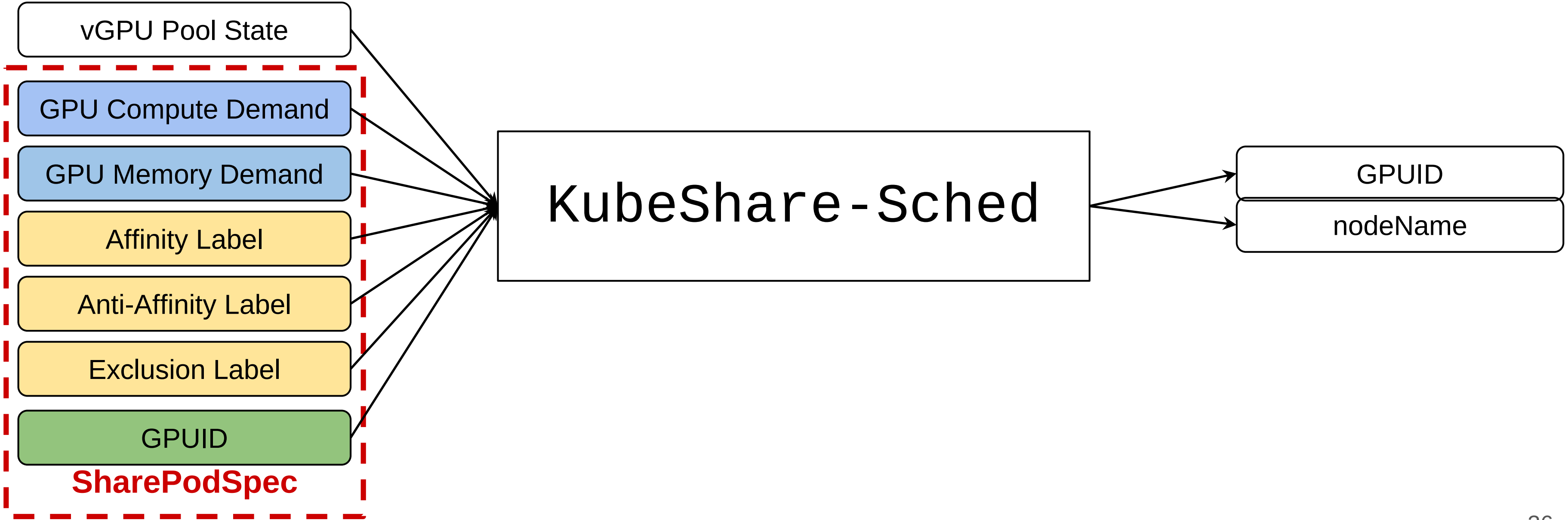
Deletion:
- Increase pod creation time
- GPU can be re-allocated by native Pod to provide better allocation flexibility

KubeShare-Sched: Resource Requirement & Scheduling



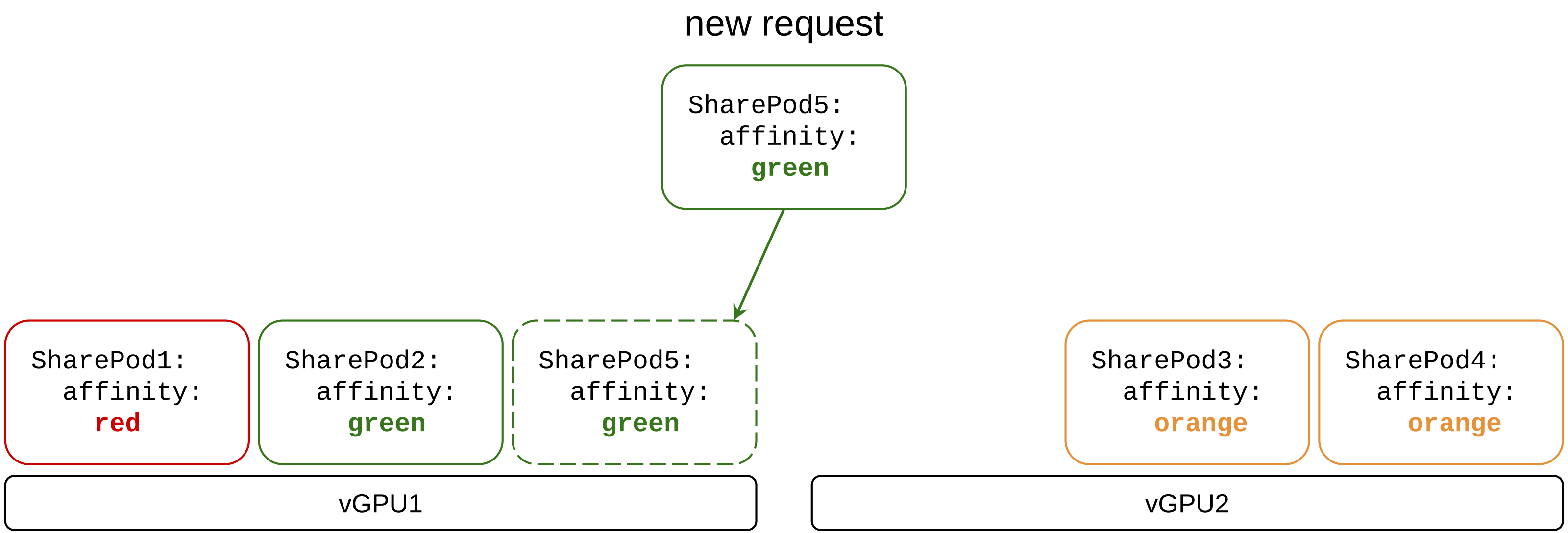
Resource Requirement Specifications

- **KubeShare-Sched** schedules SharePods by deciding their GPUID & nodeName
- Rich and Easy-to-use **user specifications** on GPU: ***usage***, *locality* & ***identity***



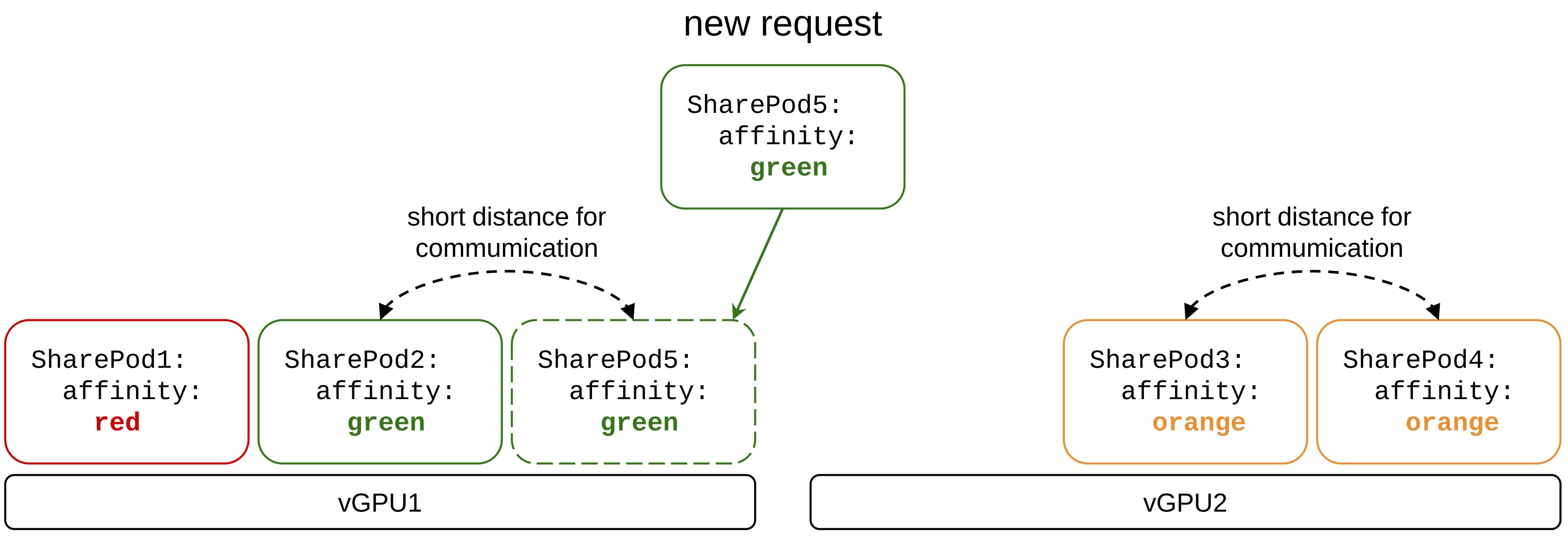
Scheduling Locality: Affinity

- Affinity forces containers with the **same label** scheduled on the **same** GPU



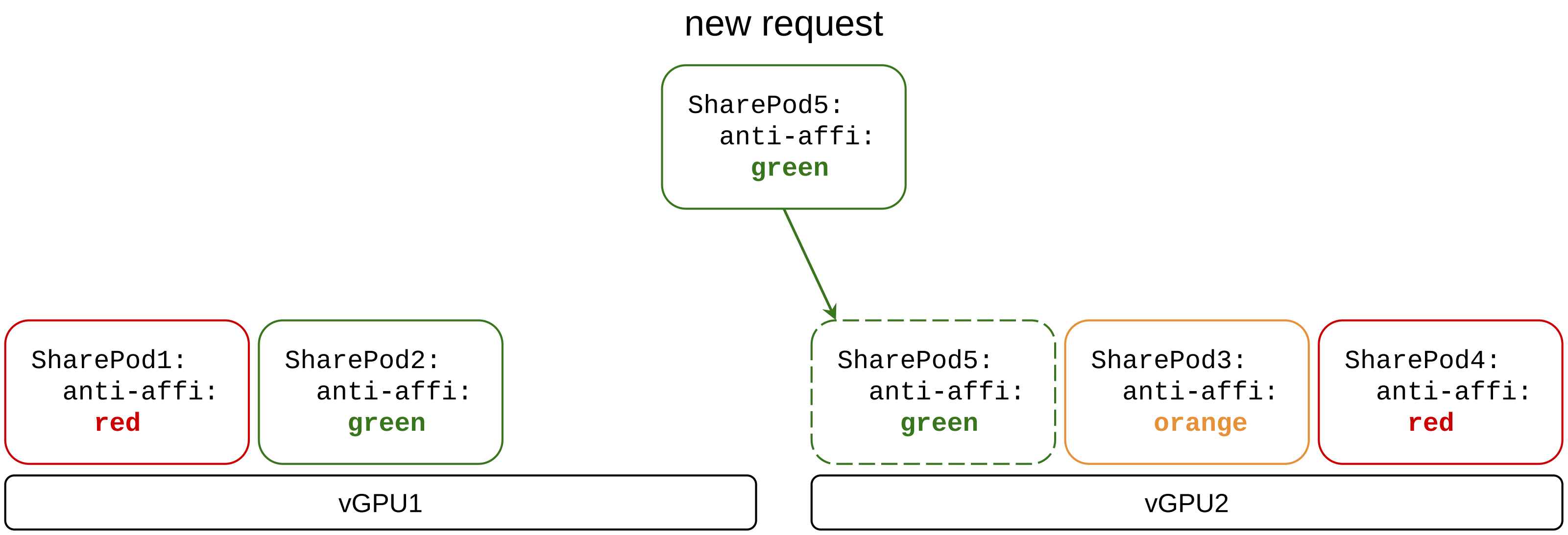
Scheduling Locality: Affinity

- Affinity forces containers with the **same label** scheduled on the **same GPU**
- Affinity can be used to **reduce communication or data transfer overhead**



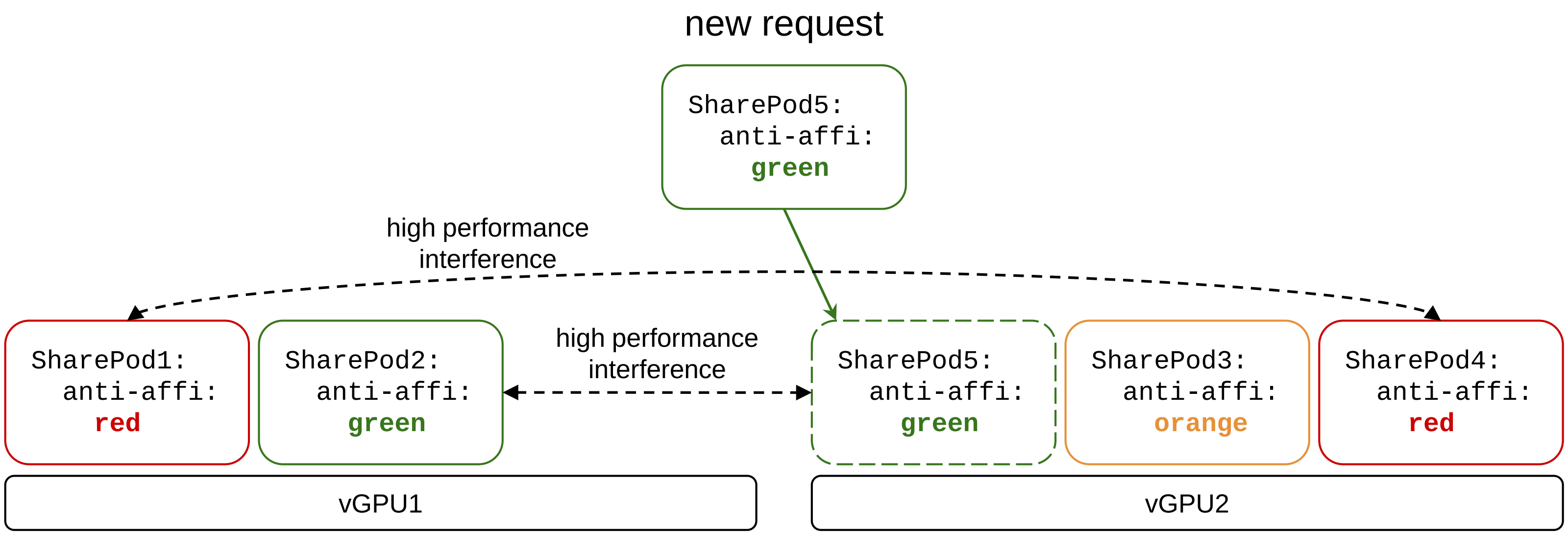
Scheduling Locality: Anti-Affinity

- Anti-affinity forces containers with the **same label** scheduled on **different GPUs**



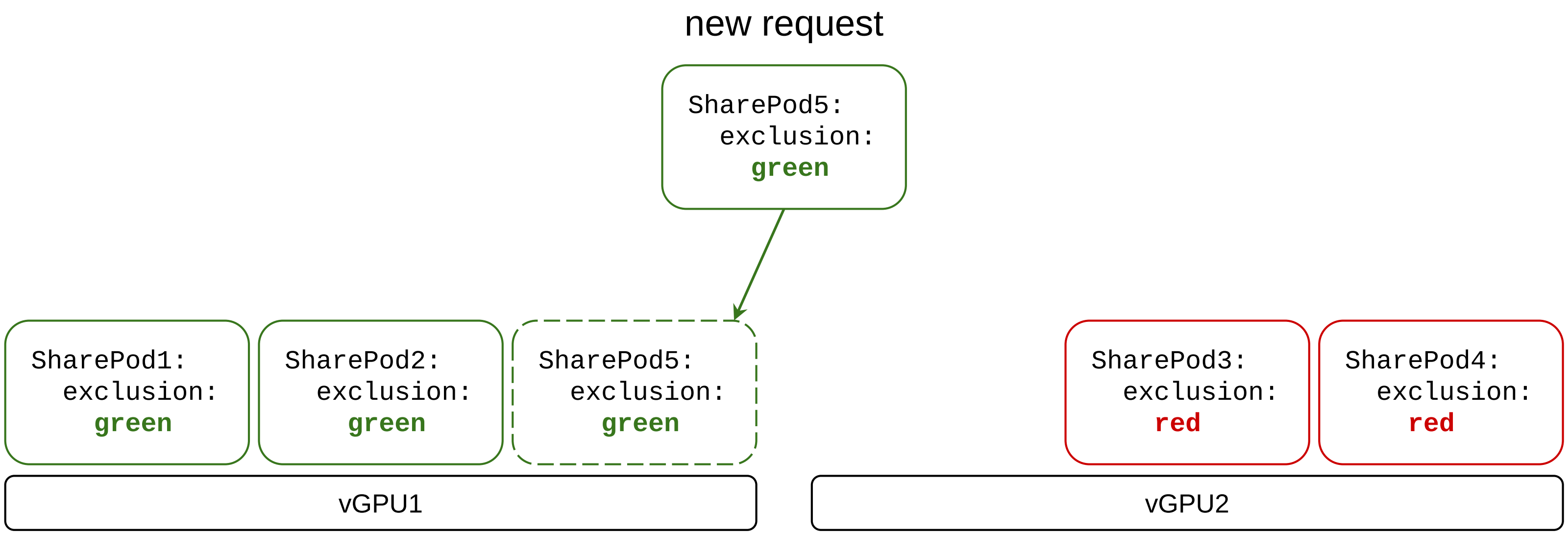
Scheduling Locality: Anti-Affinity

- Anti-affinity forces containers with the **same label** scheduled on **different GPUs**
- It can be used to **mitigate performance interference on shared GPU**



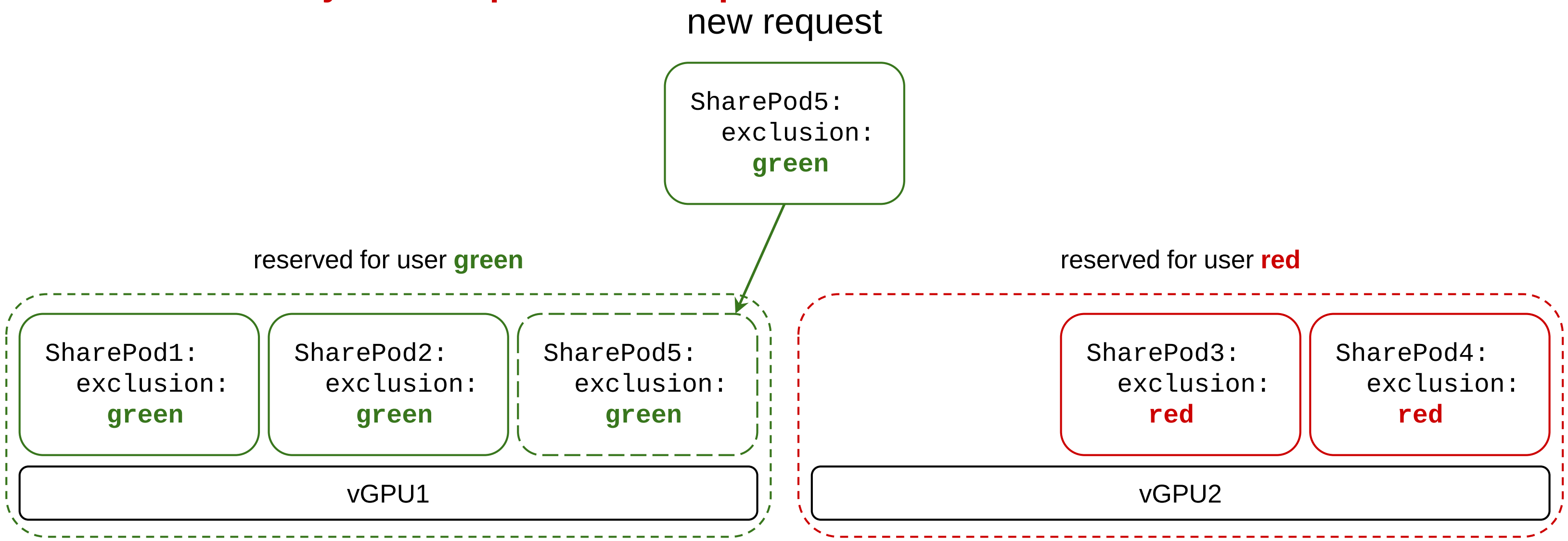
Scheduling Locality: Exclusion

- Exclusion avoids GPU sharing among containers with different labels

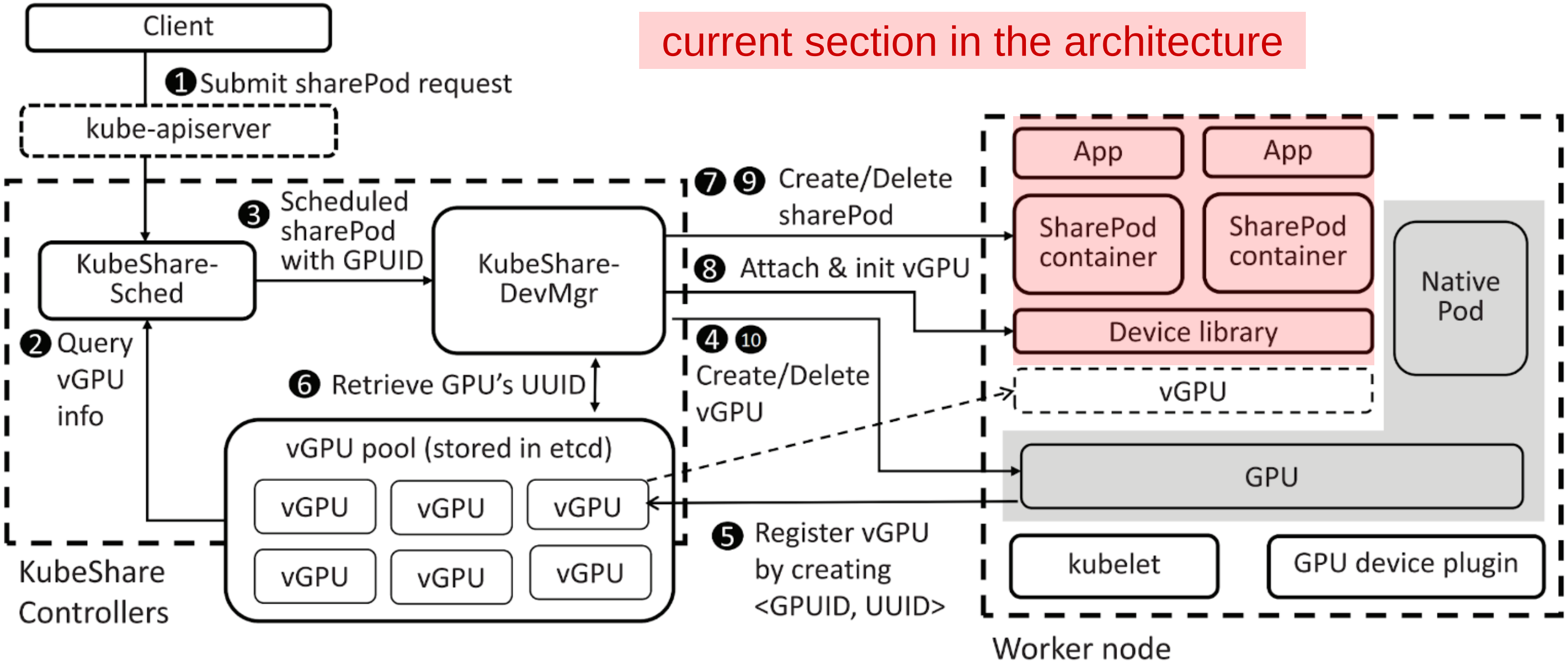


Scheduling Locality: Exclusion

- Exclusion avoids GPU sharing among containers with different labels
- Exclusion can be used to **dedicate GPU for specific users/applications**
  - **A commonly seen requirement for performance sensitive workload**



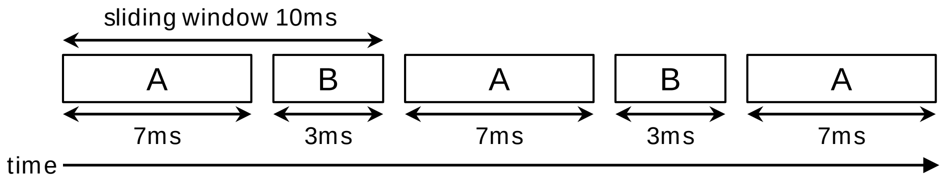
vGPU Device Library: Resource Control & Isolation



Resource Sharing Model

- Compute resource: time sharing
  - Usage = (accumulated execution time in a sliding window) / (length of the sliding window)

GPU compute request
SharePod A: 0.7
SharePod B: 0.3



- Memory resource: space sharing
  - Usage = total allocated memory size on GPU device memory
  - Memory can be oversubscriped using NIVIDIA unified memory

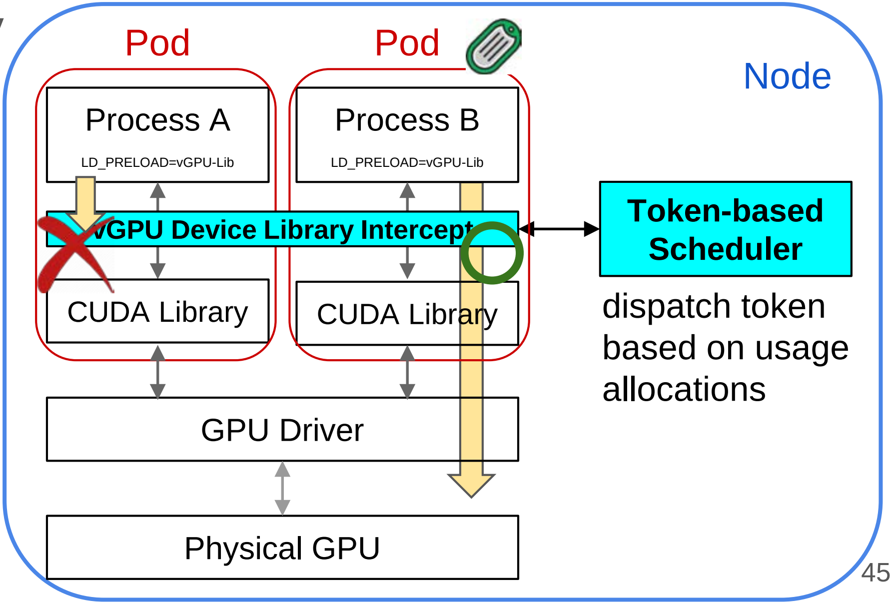
GPU memory space

Resource Control Mechanism

- Method: **intercept CUDA library calls** using **LD_PRELOAD**
  - A pod can only launch GPU kernels when it **receives a token from scheduler**
  - A pod can only allocate GPU memory when it **doesn't exceed size limit**

Category	*Function Name*
Compute	cuLaunchKernel
Compute	cuLaunchGrid
Memory	cuMemAlloc
Memory	cuArrayCreate

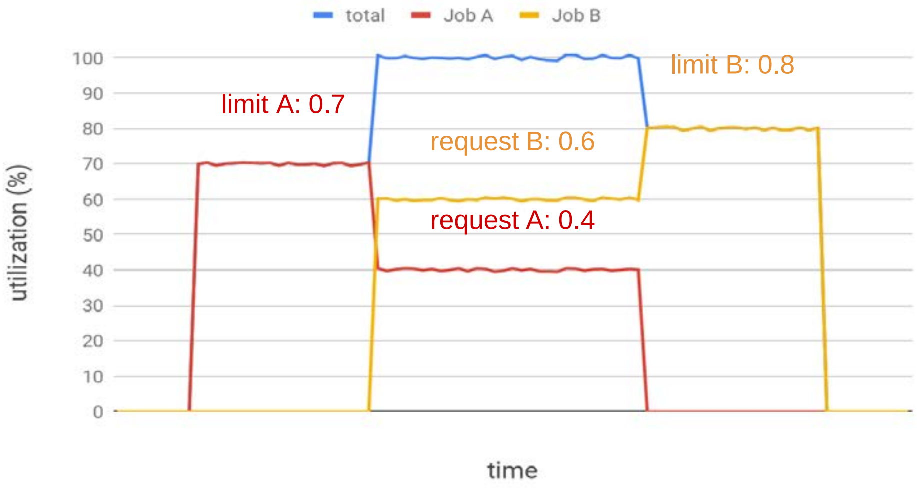
Intercepted CUDA Functions



Elastic Allocation

- More flexible resource allocation specifications for GPU time
  - **Request**: the **minimum** resource usage
  - **Limit**: the **maximum** resource usage
- Idle compute capacity can be shared without violating user requirements
  - **Achieve higher GPU utilization**

name	gpu_request	gpu_limit
Job A	0.4	0.7
Job B	0.6	0.8

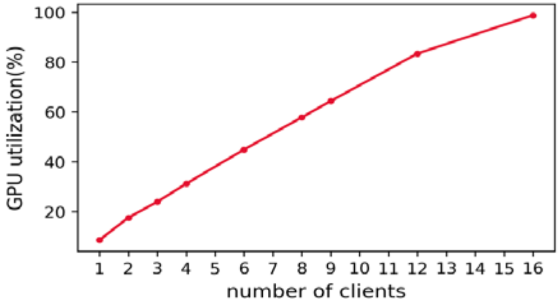


Outline

- Motivations & Objectives
- KubeShare Design & Implementation
- Experimental Evaluations
  - Experiment Setup
  - System Throughput Improvment
  - GPU Utilization Improvement
  - Mitigation of Performance Interference
  - Overhead & Scalability
- Conclusions

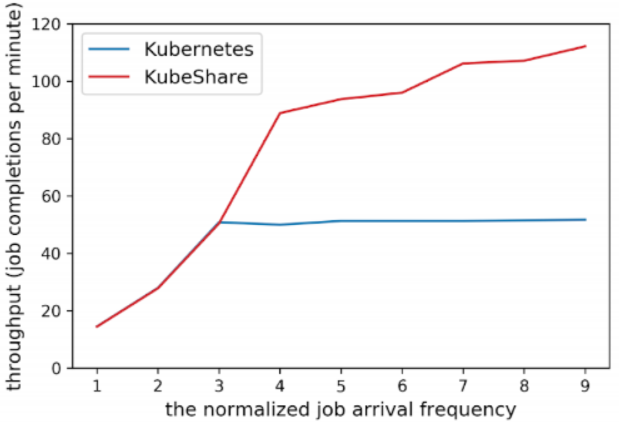
Experiment Setup

- Kubernetes clusters
  - 8 AWS p3.8xlarge instances
  - 36 cores (Intel Xeon E5-2686 v4), 244 GB RAM, 4 NVIDIA Tesla V100 16GB on each instance
- Compared container cloud platforms
  - **KubeShare**: Kubernetes with **KubeShare extension**
  - **Kubernetes**: Kubernetes native installation
- Workload: TensorFlow DeepLab V3 model inference
  - Its GPU consumption is positive correlative to #clients, so a single job may not fully utilize a GPU
  - We control the size (GPU utilization) of jobs by adjusting their concurrent client numbers
- Performance metrices
  - Application throughput: job completion per minutes
  - GPU utilization: average allocated GPU capacity

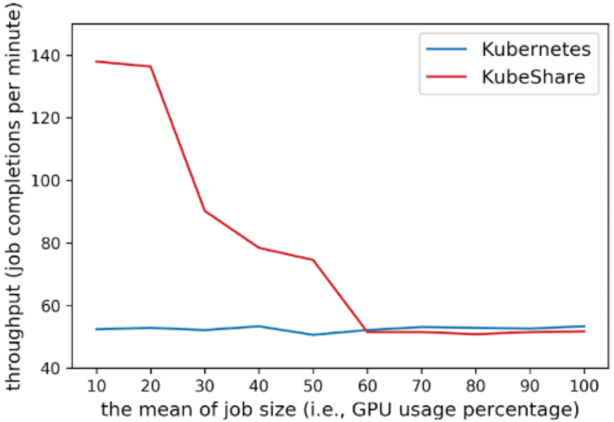


System Throughput Improvement

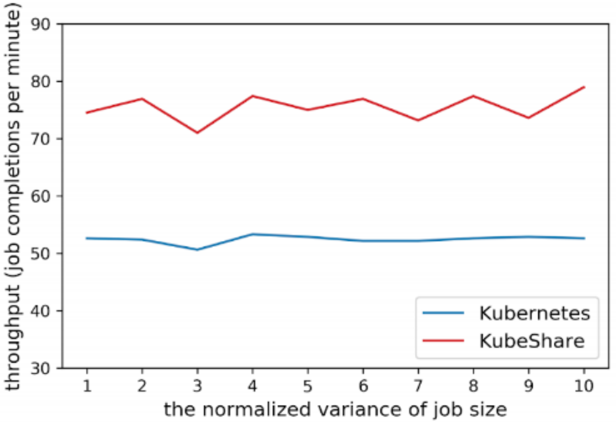
- Observe system throughput under various workload patterns



(a) Job frequency.



(b) GPU demand mean.



(c) GPU demand variance.

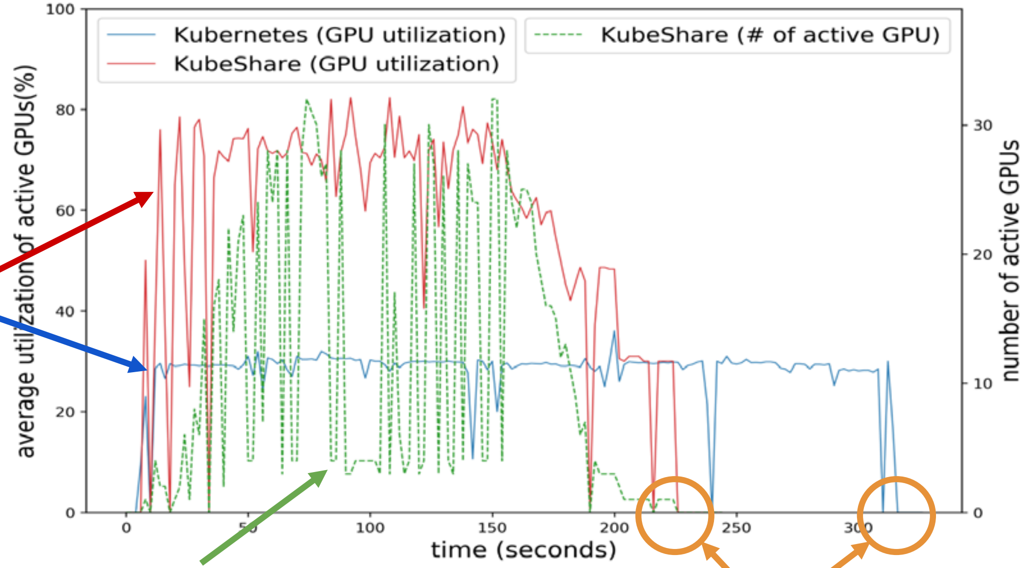
- KubeShare achieved higher throughput when workload is high enough to share GPUs
- More GPU sharing opportunities when job size is smaller
- Variance of job size doesn’t affect the throughput improvement much

GPU Utilization Improvement

- Observe average GPU utilization during workload execution

Workload:
GPU demand mean 30%
GPU demand variance 2

Average Utilization
KubeShare > Kubernetes
(>70%) (<30%)



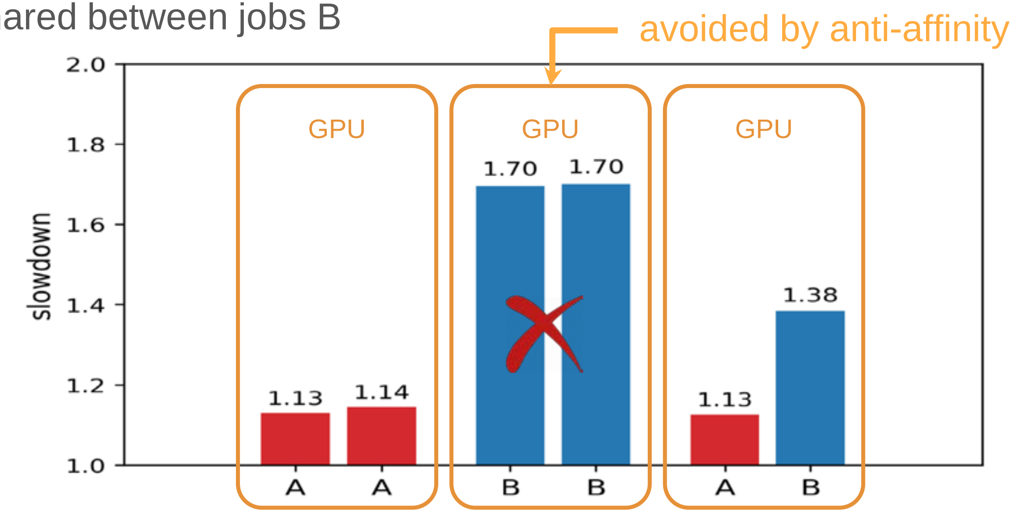
Number of active GPUs
KubeShare < Kubernetes
(< 30) (always 32)

Completion Time
KubeShare < Kubernetes
(<230s) (>310s)

Interference Mitigation: Workloads

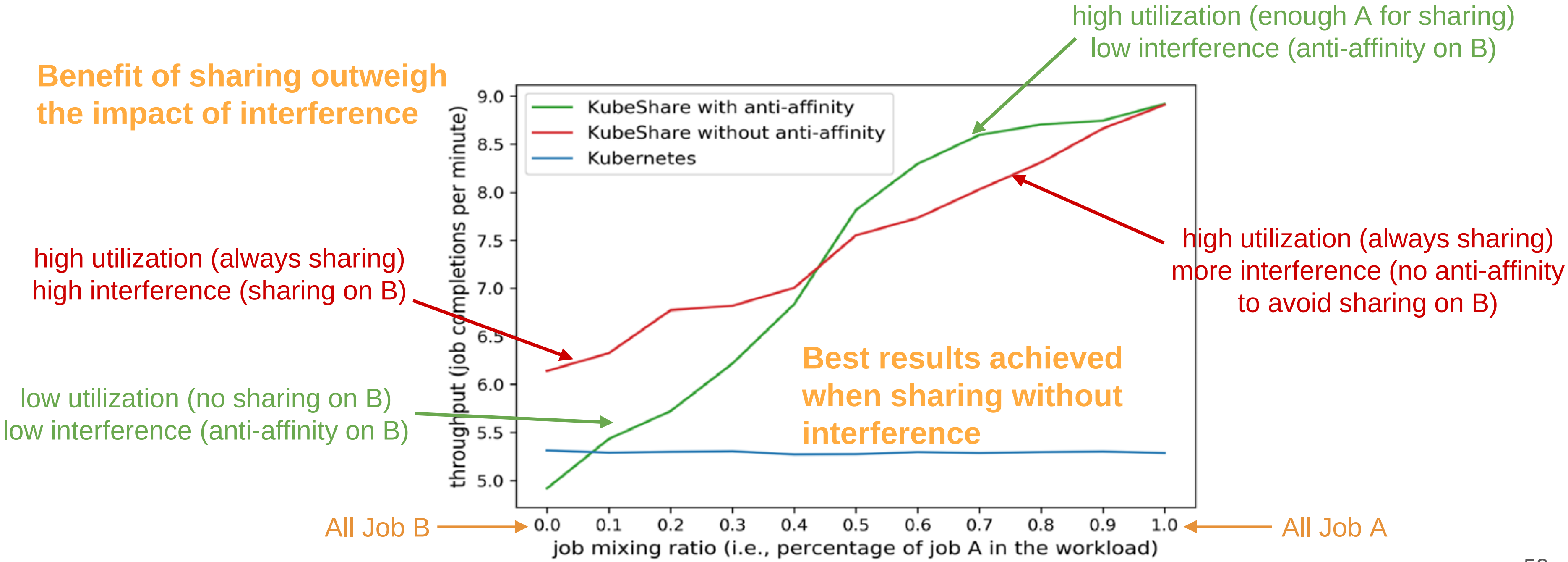
- **Anti-affinity** can be used to mitigate performance interference
  - Label all jobs B with the same color, and set the anti-affinity constraint on the label
  - Jobs B will not be scheduled on the same GPU
- But **Anti-affinity** will also reduce GPU sharing opportunities
  - GPU cannot be shared between jobs B

GPU demand
A: 50%, B: 50%



Interference Mitigation: Results

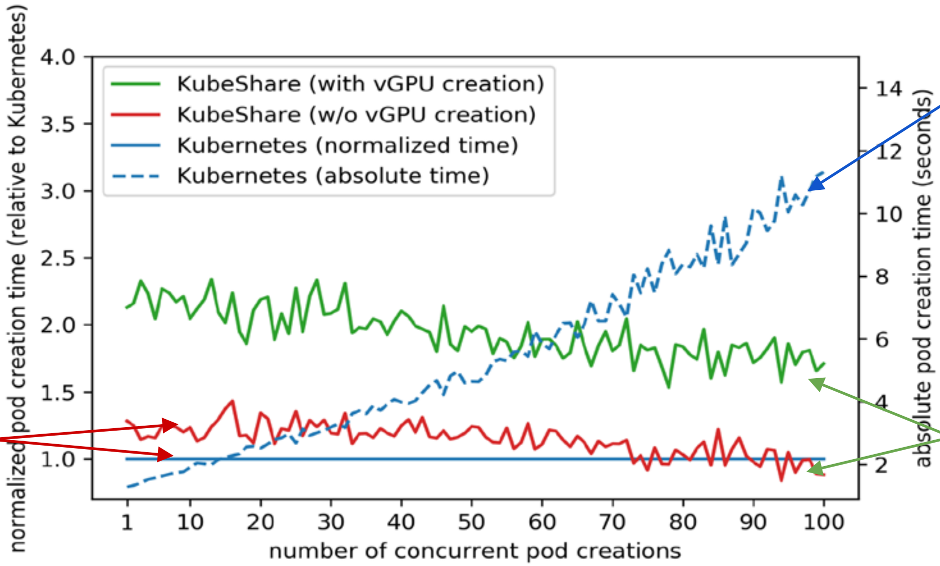
- Adjust the severity of interference by adjusting the job mixing ratio



Overhead on Pod Creation

- KubeShare needs to create vGPU before launching shared pod
    - **The overhead is bounded and can be reduced by vGPU reservation**
    - **Using reservation-based vGPU allocation can reduce the delay to only 15%**

KubeShare only take 15% more pod creation time than Kubernetes when vGPU is already created/reserved

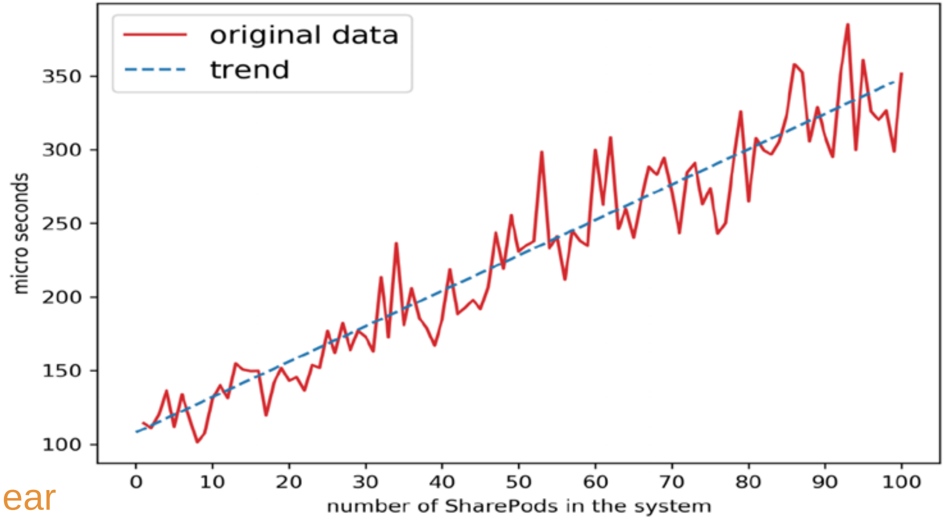


the actual time delay (with vGPU creation) still only takes less than a few seconds

KubeShare takes 2x pod creation than Kubernetw when vGPU needs to be created on-demand

Overhead on Scheduling

- Our scheduling algorithm is **scalable and efficient** for large-scale systems



the time for scheduling is less than 400 ms with 100 pods in the system

the time complexity of scheduling algorithm is linear to # of pods in the system

Conclusions

- **KubeShare** is the first work that makes GPUs become first-class and shared resources in Kubernetes to address the utilization and performance interference problems
- Users are able to specify their GPU resource requirements with usage, locality, identity constraints in KubeShare
- A series of resource management techniques were provided: on-demand vGPU creation, locality aware scheduling and elastic resource allocation
- Our design ensures KubeShare is compatible with existing Kubernetes components & NVIDIA GPU device plugin management
- Our experiments prove KubeShare can significantly improve GPU utilization and system throughput with little overhead
- Our implementation is available at https://github.com/NTHU-LSALAB/KubeShare