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

Ting-An Yeh
ta.yeh@lsalab.cs.nthu.edu.tw
National Tsing Hua University
Hsinchu, Taiwan R.O.C.

Hung-Hsin Chen
jim90247@gmail.com
National Tsing Hua University
Hsinchu, Taiwan R.O.C.

Jerry Chou
jchou@lsalab.cs.nthu.edu.tw
National Tsing Hua University
Hsinchu, Taiwan R.O.C.

ABSTRACT

Container has emerged as a new technology in clouds to replace virtual machines (VM) for distributed applications deployment and operation. With the increasing number of new cloud-focused applications, such as deep learning and high performance applications, started to reply on the high computing throughput of GPUs, efficiently supporting GPU in container cloud becomes essential. While GPU virtualization has been extensively studied for VM, limited work has been done for containers. One of the key challenges is the lack of support for GPU sharing between multiple concurrent containers. This limitation leads to low resource utilization when a GPU device cannot be fully utilized by a single application due to the burstiness of GPU workload and the limited memory bandwidth. To overcome this issue, we designed and implemented KubeShare, which extends Kubernetes to enable GPU sharing with fine-grained allocation. KubeShare is the first solution for Kubernetes to make GPU device as a first class resources for scheduling and allocations. Using real deep learning workloads, we demonstrated KubeShare can significantly increase GPU utilization and overall system throughput around 2x with less than 10% performance overhead during container initialization and execution.

KEYWORDS

Cloud computing, GPU, Container, Scheduling

ACM Reference Format:
Ting-An Yeh, Hung-Hsin Chen, and Jerry Chou. 2020. KubeShare: A Framework to Manage GPUs as First-Class and Shared Resources in Container Cloud. In *29th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '20), June 23–26, 2020, Stockholm, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3369583.3392679

1 INTRODUCTION

The emergence of containers has revolutionized cloud computing. Due to negligible runtime overhead and much higher deployment density on a physical machine, containers are perceived as a lightweight replacement of virtual machines (VMs) for resource allocation and service deployment. The agility of containers combining with microservice-style software architecture further advances the practice of DevOps [16] to achieve continuous software delivery, productivity, automation, cost saving, in the software delivery process. Therefore, containerization has been increasingly adapted and supported in cloud platform. One of the industry leading solution for containerization is Docker [21], and it has been used by many of the Internet's most predominate systems today [30].

As increasing number of applications are deployed and architected based on containers, the need of an orchestration systems for scheduling, deploying, updating and scaling such container-based applications becomes crucial. Many container-management systems have been developed by industry and open source community, including YARN [29], Mesos [13], Borg [30], etc. But Kubernetes [9] is the most popular one among all by far. Kubernetes is developed and used by Google to power their cloud container service. Kubernetes not only provides powerful tools for developer to manage loosely-coupled and stateless Docker containers without having to interact with the underlying infrastructure, but also has a highly configurable and extensible architecture to support custom cluster operators, including load balancing, container replication, rolling update, volume and network management, etc.

While Kubernetes has the strength to support container management, the only computing resources that can be natively recognized and allocated by Kubernetes are the CPU and memory. To attach any other custom devices to a container, including GPU, high-performance NICs, FPGA, a device plugin [14, 22, 27] must be developed and installed following the framework defined by Kubernetes to perform vendor specific initialization and setup for the devices. The device plugin framework successfully separates the vendor-specific code from Kubernetes for better system maintenance and extensibility, but it does not allow resource sharing or fractional allocation on custom devices. This limitation inevitably leads to lower resource utilization when a custom device cannot be fully utilized by a single application or container. The problem is further aggravated over the past decade by the demand surge and the growing price of high performance computing devices, like GPU. With the growing interests from High Performance Computing (HPC) community for container-based computing [5, 20, 33], the problem has drawn attentions from both research and industrial communities in recent years [2, 10, 15].

In this work, we present *KubeShare*, which extends Kubernetes to support GPU sharing with fine-grained allocation and first-class resource management. A first-class resource means that the resource entity can be explicitly identified and selected by both the resource manager and the users. As observed from our experiments, it is an essential property to address the performance interference problem in shared resource environment. It is a non-trivial task due to the lack of proper resource description, allocation policy,

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

ACM ISBN 978-1-4503-7052-3/20/06…$15.00
https://doi.org/10.1145/3369583.3392679

and architecture support in the existing Kubernetes framework. To the best of our knowledge, only a few attempts [2, 7, 10] have been made recently to support GPU sharing in Kubernetes, but none of them treats GPU as first-class schedulable entities. As a result, their GPU throughput and utilization can easily suffer from the resource fragmentation and performance interference problems. In contrast, KubeShare allows users to specify locality constraints on their allocated GPUs, so that a GPU can be shared among containers with less resource contention. Furthermore, we ensure the implementation of KubeShare is compatible to the existing Kubernetes architecture, and its performance overhead of GPU management is limited and scalable to the users requests and workload. We evaluated the performance and overhead of KubeShare comparing to the native Kubernetes implementation using a set of real deep learning GPU workloads. The results showed that KubeShare can significantly increase GPU utilization and overall system throughput by a factor of 2x with less than 10% performance overhead during the initialization and execution of a container.

The rest of the paper is structured as follows. Section 2 introduces the background on Kubernetes. Section 3 details the problems and requirements of supporting GPU sharing in Kubernetes. Section 4 presents the components and implementations of KubeShare. The evaluation on a real Kubernetes cluster is shown in Section 5. Finally, Section 6 and Section 7 are the related work and conclusions.

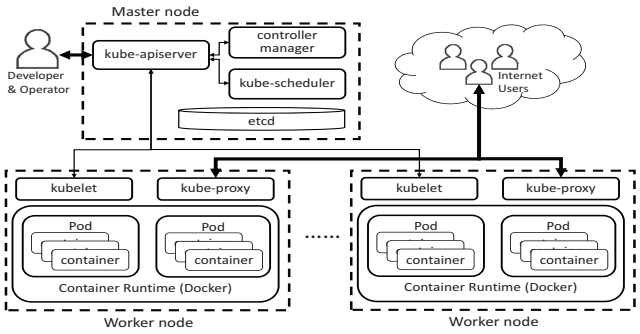
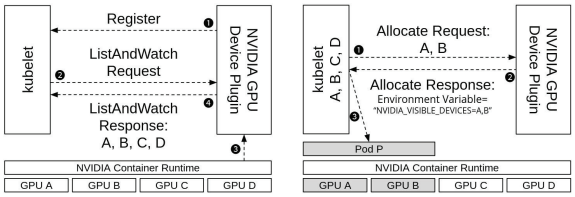


Figure 1: The system architecture of Kubernetes.



(a) The initialization phase. **(b) The allocation phase.**

Figure 2: The workflow of device plugin in Kubernetes.

2 BACKGROUND ON KUBERNETES

2.1 Architecture

Kubernetes has become increasingly popular as a reliable distributed system platform for running and managing containerized applications. The smallest deployable units of computing that can be created and managed in Kubernetes is called *pod*. A pod represents a logical host that contains one or more containers which are always co-located, co-scheduled, and run in a shared context. For the ease of discussion, we assume there is only one container per pod in this paper. So *container* and *pod* are interchangeable terms in the rest of the paper.

Kubernetes provides a modular API core to let users describe the specification of the desired behavior of a pod in a YAML or JSON object, which is also called *PodSpec*. The Kubernetes system constantly works to ensure that the pod object exists, and its actual state matches the desired state in PodSpec. The system architecture of Kubernetes is shown in Figure 1, and we briefly introduce its components as follows.

Master components

kube-apiserver: It provides the frontend to the shared state through which all other components interact. It also serves the CRUD (create, read, update, delete) operations on the Kubernetes objects, including the sharePod resource kind defined by KubeShare.

etcd: It is a distributed, highly-available key value data store that Kubernetes uses to store cluster configuration, and the metadata of all resources, such as the PodSpec objects.

kube-scheduler: It implements the logic for scheduling pods onto nodes by considering the resource requirements from users, and the resource availability across nodes. It plays a key role in resource allocation.

controller manager & controllers: Controllers are control loops that continuously ensure that the current state of the cluster matches the desired state. For instance, ReplicationController ensures the specified number of pod replicas are running at any one time. Kubernetes is highly configurable and extensible by allowing the cluster manager to define and implement their own controllers. Our solution, KubeShare, is also implemented by a set of custom controllers as described in Section 4.6.

Node components

kubelet: The primary "node agent" that runs on node to ensures that the containers in the pods assigned to the node are running and healthy.

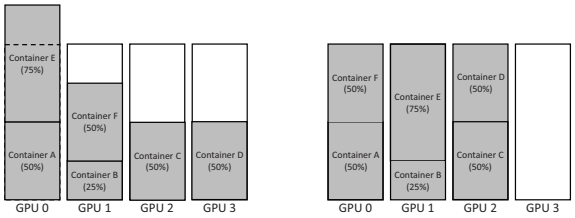
container runtime: The software that is responsible for running containers. In our case, it is Docker [21]), to instantiate the containers.

kube-proxy: A network proxy to support network communication among pods.

2.2 Custom Device

The only resource types that can be natively recognized by Kubernetes are the CPU and memory on physical machines. Hence, Kubernetes provides a device plugin framework to let kubelet attach other custom devices to pods, including GPU, high-performance NICs, FPGAs, etc. Instead of customizing the code for Kubernetes itself, vendors can implement a device plugin that helps kubelet to perform vendor specific initialization and setup for the devices.

Figure 2 illustrates the general workflow of device plugins at the initialization and allocation phases. In the initialization phase (shown in Figure 2a), the device plugin performs vendor specific initialization and setup to make sure the devices are in a ready state, and then registers itself to kebelet. Following a successful



(a) Round-robin scheduling. **(b) Locality aware scheduling.**

Figure 3: Resource fragmentation can cause resource over-commitment and under-utilization problems, if a scheduler is not aware of the identity of the GPU assigned to a container in a node.

registration, the device plugin sends kubelet the list of devices it manages, and then kubelet tracks the device status through the ListAndWatch request. Whenever a device state changes or a device disappears, its device plugin returns the new device list to kubelet. kubelet is then in charge of advertising those resources to the kube-apiserver as part of the kubelet node status update. Hence, users can request the custom devices as the resources on a node in PodSpecs.

When kubelet receives a request for custom device, it enters the allocation phase (shown in Figure 2b) by sending an Allocate request to device plugin. The device plugin then responses with the necessary information to let kubelet attach the requested device, such as environment variables, mount path, etc. Take GPU as an example, through the nvidia-docker2 package, kubelet can attach a GPU to containers by simply adding GPU's UUID to container's environment variable. Hence, the GPU device plugin only needs to return the NVIDIA_VISIBLE_DEVICES environment variable in the Allocate response.

3 PROBLEMS & REQUIREMENTS

While the device plugin framework described in Section 2.2 enables custom devices to be exposed and attached to pods, GPUs are neither shareable nor first class schedulable entities in the system. We discuss the limitations of existing work and propose the requirements of our solution in this section.

3.1 Resource Fragmentation

Device plugin framework is designed based on the assumption that custom devices cannot be fractionally allocated or over-committed. Therefore, the resource demand of custom device in PodSpec must be an integer value, and the allocate request cannot describe the requesting resource amount. Hence, once a device is attached to a pod by kubelet, it cannot be attached to another pod again.

A trivial solution that tricks the Kubernetes framework is to multiply the resource unit by a scaling-factor, such as 100. Then users can use an integer value to represent a fractional allocation request, and the scheduler can also count the fractional resources by integer. For instance, if a node has 2 GPU devices, its device plugin will register 200 device instances to kubelet instead. A user then can allocate a 50% GPU device by requesting 50 GPU device units in the PodSpec, and the kube-scheduler and kubelet can also correctly count the amount of residual resource capacity.

However, the above strategy can still suffer from resource fragmentation problem because kube-scheduler doesn't treat GPU as first class resource, and it only has the information on the aggregated resource capacity from a node not on individual devices. As illustrated by the example shown in Figure 3, a set of containers (Container A ~ Container F) are scheduled to a node with 4 GPUs. Without the ability to control the GPU location of a job, the jobs are assigned to GPUs in a round-robin fashion as shown in Figure 3a. As a result, some GPUs are over-committed, while others are under-utilized. In contrast, as shown in Figure 3b, a locality-aware scheduler not only can ensure job performance by avoiding resource over-commitment, but also maximize resource utilization by minimizing the number of active GPUs. Therefore, it is clear that the GPU resources cannot be properly allocated under sharing environment without treating GPUs as a first class resources, such that they have unique identifiers and usage status.

3.2 Implicit and Late Binding

As described in Section 2, kube-scheduler is responsible for assigning a pod to a node with sufficient custom device resources. However, the binding between custom device and pod is both implicit and late (i.e., meaning after the scheduling decision is made) from the view of kube-scheduler. As seen from the workflow of device plugin framework, kube-scheduler has no control of which device on the node to be assigned to a pod, and the binding decision is not made until kubelet creates the pod on a node. As a result, none of the Kubernetes components except kubelet can control the pod-to-device binding. However, kubelet shouldn't be modified for any custom resource management, because any change to the kubelet will affect all the pods running on a node, and kubelet is not a extensible component in the Kubernetes architecture. Therefore, a new architecture design is desired to support explicit binding and first-class resource management on GPU. Without it, we cannot control how a custom device is shared among pods in order to achieve predictable performance and to manage interference on concurrent usage.

3.3 Requirements

To summarize, the goal of our work is to enable GPU sharing in Kubernetes while satisfying the following requirements from the user point of view.

Fine-grained allocation: We should allow users to specify their resource demands on a shared GPU by any fractional values, and allocate resources with consideration of fragmentation mitigation.

Resource isolation: We should isolate the GPU usage among containers according to the user resource requirements on the computation time, and memory space on GPU.

Locality awareness: We should give users the capability to control the binding between GPUs and containers to mitigate performance interference on shared GPU.

Low overhead: Our implementation should limit performance overhead to all the operations on containers, including the instance initialization time and the application performance.

Platform compatibility: Our implementation should be compatible to the Kubernetes platform, specifically in the following aspects: (1) Its design and implementation must follow the extension

Table 1: Comparison of GPU sharing solutions for Kubernetes.

Property	Features	Deepomatic [7]	Aliyun [2]	GigaGPU [10]	KubeShare
Sharing	Multi-GPUs per node	No	Yes	Yes	Yes
	Fine-grained allocation	limited	limited	limited	Yes
Isolation	Memory	No	Yes	Yes	Yes
	Computation	No	No	Yes	Yes
Scheduling	First class with GPU identity	No	No	No	Yes
	locality constraint	No	No	No	Yes
Compatibility	Co-exist with kube-scheduler	No	No	No	Yes

```
apiVersion: kubeshare/v1
kind: SharePod
metadata:
  name: pod1
  annotations:
    "kubeshare/gpu_request": "0.5"
    "kubeshare/gpu_limit": "0.8"
    "kubeshare/gpu_mem": "0.5"
    "kubeshare/sched_affinity": "green"
    "kubeshare/sched_anti-affinity": "red"
    "kubeshare/sched_exclusion": "blue"
    "kubeshare/GPUID": "xyz"
spec: #PodSpec
  nodeName: node1
  containers:
  - name: example
    image: nvidia/cuda:9.0-base
    command: ["nvidia-smi", "-L"]
```

Script 1: An example of SharePodSpec.

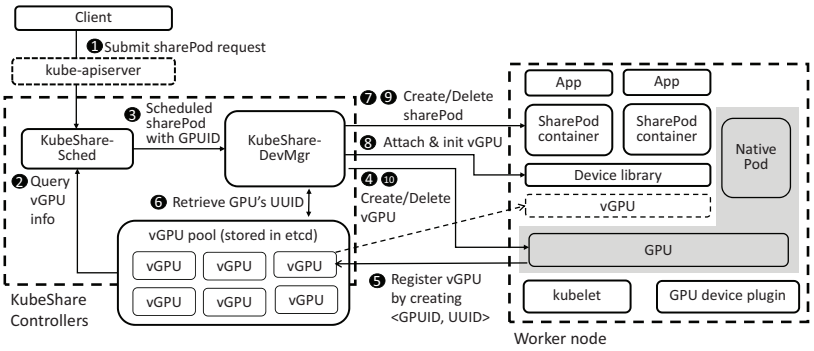


Figure 4: The components and workflow of KubeShare.

framework defined by Kubernetes. (2) It can co-exist and collaborate with the native kube-scheduler to manage GPU resources in a Kubernetes cluster. (3) It can be seamlessly integrated with other custom Kubernetes controllers to manage containers with fractional allocated GPUs.

4 KUBESHARE

In this work, we developed a new framework extension to support GPU sharing in Kubernetes, called ***KubeShare***. As summarized in Table 1, KubeShare has the strongest support for GPU sharing, isolation, scheduling, and compatibility comparing to all other known solutions with detailed discussions in Section 6. This section explains the design and implementation of KubeShare.

4.1 Architecture Overview

The role of KubeShare is to create and manage ***sharePod***, which is a custom resource kind we created in Kubernetes to represent the pod with ability to attach shared custom device on its containers. As shown in Script 1, the specification to create *sharePod* is called ***SharePodSpec***, which contains the information of the original podSpec, the resource usage requirements of the GPU, the identifier of a GPU (GPUID), and the nodeName of the GPU.

Figure 4 shows the architecture of KubeShare. To enable GPU sharing, KubeShare allocates the physical GPUs from Kubernetes, and then fractionally assigned them to sharePods. These shared GPUs managed by KubeShare is called, ***vGPU***. The physical location of these vGPUs can be spread over multiple nodes in the cluster when KubeShare acquires them from Kuberntes. Therefore, we use ***vGPU pool*** to represent the set of all vGPUs managed by KubeShare. When a GPU joins the vGPU pool, it is assigned with a unique identifier (***GPUID***), so that explicit GPU assignment and binding can be supported to address the fragmentation and interference problems. We introduce the KubeShare components and workflow as follows:

Client is a Kubernetes user who wants to allocate fractional GPU resources to its containers. Clients interact with KubeShare by submitting their resource specifications through the kube-apiserver API. The resource specification supported by KubeShare is detailed in Section 4.2.

KubeShare-Sched is the scheduler that decides the mapping between containers and vGPUs according to the current resource status and the resource requirements specified by *client*. KubeShare-Sched then generates the *SharePodSpec* with the GPUID value decided by a scheduling policy, and asks KubeShare-DevMgr to create the corresponding sharePod instance. Any scheduling algorithm can be implemented in this component. But our main goal is to design a resource locality aware scheduling algorithm while maximizing resource utilization as detailed in Section 4.3.

KubeShare-DevMgr creates sharePod objects, and then initializes the device environment in containers upon receiving the SharePodSpec from KubeShare-Sched. Specifically, it sets the NVIDIA_VISIBLE_DEVICES environment variable and installs a vGPU device library in containers to isolate their GPU usage. KubeShare-DevMgr is also responsible to manage the vGPU pool in a on-demand or reservation manner as detailed in Section 4.4.

vGPU Device Library: It is a library installed in each container to throttle and isolate the GPU usage by intercepting all the memory-related APIs and the computing-related APIs in the CUDA Library. More details about our control mechanism are given in Section 4.5.

4.2 First-Class Resource Specification

KubeShare treats GPUs as ***first class resources***, thus it allows users to specify scheduling requirements and constraints on a GPU in the specification as shown in Script 1. The role of KubeShare is to ensure GPU resources are allocated and assigned to pods without violating these user requirements and constraints.

The resource requirements include the computing and memory usage demand on GPU. As detailed in Section 4.5, GPU memory is shared by space, and GPU computing capacity is shared by time slice. For instance, `gpu_mem`=0.5 means that a container can allocate at most 50% of the total device memory space, while `gpu_request`=0.5 means that a container should have at least 50% of the kernel execution time in a sliding window. Similar to how Kubernetes manages CPU resource, KubeShare supports ***elastic resource allocation*** on GPU as well. That means KubeShare will guarantee the minimum resource allocation of a container specified by `gpu_request`, and allow a container to utilize the residual capacity on GPU as long as its usage doesn't exceed the value of `gpu_limit`. KubeShare allows all these resource demands to be specified in fractional numbers between 0 and 1.

The locality constraints are used to control the mapping between GPUs and containers. Currently, we support three types of constraints: ***exclusion***, ***affinity*** and ***anti-affinity***. All of them are specified by labels (i.e., an arbitrary string). *Exclusion* is used to exclude GPU sharing among containers with different labels. A use case scenario of exclusion is to allocate dedicate resources for a specific user or application by only giving the same exclusion label to their containers. The dedicate resource allocation strategy can avoid unexpected performance interference from other users' containers. Affinity forces the containers with the same label to be scheduled on the same GPU. A user may use this constraint to reduce the communication overhead among containers, or to designate a specific group of containers for sharing a single GPU device. Finally, *Anti-Affinity* is the opposite constraint of affinity, so it forces the containers with the same label to be scheduled on different GPUs. As demonstrated by our experiment in Section 5.5, the anti-affinity constraint can be used to force the jobs with resource contentions to be scheduled on separate GPUs for minimizing the impact of performance interference. Noted, Kubernetes also supports similar scheduling filters for node selection, but not on the device selection. Only KubeShare can support these constraints on device selection because it treats GPU as the first-class resource.

4.3 Resource & Locality Aware Scheduling

The role of KubeShare-Sched is to determine the mapping between vGPUs and containers. Our goal is to design an algorithm Algo. 1, that can ***satisfy the resource and locality constraints from users while maximizing resource utilization***. The algorithm inputs are r and D, where r is the requirements and constraints of a container from user request, and D is the list of available devices in vGPU pool. Table 2 summarizes the notations used in the algorithm. Noted, the affinity attribute of a device can be a set of labels because containers having different affinity labels can be placed on the same device simultaneously. The algorithm output is the GPUID value needed in the SharePodSpec.

Algorithm 1: Locality & Resource Aware Scheduling

Input : r: requirements of a container.
D: a list vGPUs available in vGPU pool.
Output: The GPUID of an assigned vGPU.

```
/* Step1: assign to device by affinity label */
if r.aff ≠ null then
    Find d ∈ D, such that r.aff ∈ d.aff ;
    if d ≠ null then
        if r.excl ≠ d.excl then return -1 ;
        if r.anti_aff ⊂ d.anti_aff then return -1 ;
        if r.mem > d.mem or r.util > d.util then return
          -1 ;
        d.anti_aff = d.anti_aff ⋃ r.anti_aff ;
        return d.id
    else
        Find d ∈ D, such that d.idle == true;
        if d == null then d = new_dev();
        d.aff = d.aff ⋃ r.aff ;
        d.anti_aff = d.anti_aff ⋃ r.anti_aff ;
        d.excl = r.excl;
        return d.id
    end
end
/* Step2: filter devices by the exclusion and
   anti-affinity labels */
S = D;
foreach d ∈ D do
    if d.idle == false then next;
    if (r.excl ≠ null or d.excl ≠ null) and
      (r.excl ≠ d.excl) then Remove d from S;
    if (r.anti_aff ≠ null and r.anti_aff ⊂ d.anti_aff ) then Remove
      d from S;
    if r.mem > d.mem or r.util > d.util then
        Remove d from S
    end
end
/* Step3: placement algo */
S' = {d' | d' ∈ S and d'.aff == ∅ };
d = best_fit(r, S');
if d == null then d = worst_fit(r, S \ S');
if d == null then d= new_dev();
d.excl = r.excl;
return d.id
```

Table 2: The notations used in Algo 1.

$r.util$	the `gpu_request` of a container
$r.mem$	the `gpu_mem` of a container
$r.aff$	the `sched_affinity` label of a container
$r.anti\_aff$	the `sched_abti-affinity` label on a container
$r.excl$	the `sched_exclusion` label of a container
$d.util$	the residual computing capacity of a device
$d.mem$	the residual memory space of a device
$d.aff$	the set of affinity labels on a device
$d.anti\_aff$	the set of anti-affinity labels on a device
$d.excl$	the exclusion label on a device
$d.idle$	whether any container is scheduled on the device

The algorithm has three steps. Step1(line1-14): If the container has an affinity label, we first check if a device with the same label already existed. If it exists, the container must be scheduled to the device due to the affinity constraint (line3). However, if the targeted device violates the resource or other locality constraints (i.e., exclusion and anti-affinity), we still have to reject the request (line4-6). If no device has the same affinity label at the moment, the container can be scheduled to any device. But we prefer to schedule it on an idle or new device without any workload (line9-14), so that there can be more space on the GPU to serve other containers with the same affinity label containers arriving in the future. Noted, an idle vGPU means that it already exists in the vGPU pool, but has no container scheduled on it. In contrast, a new vGPU means that it needs to be created and added into the pool by KubeShare-DevMgr. To ask KubeShare-DevMgr create a new vGPU, we only need to return a non-existed GPUID in vGPU pool (detailed in Section 4.4), so the new_dev() function here simply generates a device variable with a new hashed id (line10 and line24).

Step2(line15-20): If the container has no affinity label, we construct a candidate device list (S) by filtering out the devices that cannot satisfy the constraints of exclusion, anti-affinity, and resource demand (line18-20). Step3(line21-26): Make a valid scheduling decision by choosing a device from the candidate device list. We prioritize our choices in the following order: (1) devices without affinity label (line22); (2) devices with affinity label (line23); (3) new created devices (line24). We use best fit algorithm when choosing devices without affinity label, and use worst fit algorithm when choosing devices with affinity label. The intuition behind our policy is to keep more space on the device with affinity label for future requests with the same affinity label, and to utilize the resources of existing vGPUs as much as possible. Only when the container cannot fit into any existing vGPUs, we will create new vGPU to serve the request.

4.4 vGPU Lifecycle Management

vGPUs are resource objects managed by the KubeShare-DevMgr controller. The lifecycle of vGPU consists of four phases: ***creation***, ***active***, ***idle***, and ***deletion***. A vGPU is created when KubeShare-DevMgr receives a sharePod request containing an non-existent GPUID. If the GPUID in sharePod request already exists, KubeShare-DevMgr simply retrieves the corresponding UUID from vGPU pool without creating a new vGPU. A vGPU becomes active when it is attached to one or multiple sharePods. A vGPU is detached from a sharePod when the sharePod is deleted by users in Kubernetes. When a vGPU is not attached by any of the sharePods, it enters the idle state. An idle vGPU can become active again when it is assigned to another sharePod by the KubeShare-Sched or users. Finally, KubeShare-DevMgr decides when to delete an idle vGPU, and release the GPU to Kubernetes. We detail the actions taken by KubeShare-DevMgr in each phase as follows.

To convert a non-shareable GPU in Kubernetes into a shareable vGPU managed by KubeShare, KubeShare-DevMgr first has to allocate the GPU device from Kubernetes by creating a native Kubernetes pod that requests a GPU device on a node. The sole purpose of this pod is to allocate the GPU without running any workload. After the pod is launched as described in Section 2.2, KubeShare-DevMgr obtains the actual device UUID from the environment variable inside the launched container, and stores the mapping between GPUID and UUID.

To attach a vGPU in a container, KubeShare-DevMgr plays a similar role as the native Kubernetes device plugin by setting the NVIDIA_VISIBLE_DEVICES environment variable inside the container. But different from device plugin which doesn't know which container is requesting the device, KubeShare-DevMgr performs an explicit binding between container and device according to the GPUID in SharePodSpec. Noted, since GPUID is only a virtual identifier in KubeShare, KubeShare-DevMgr must convert the GPUID to UUID, and use the UUID in the NVIDIA_VISIBLE_DEVICES setting. In addition, KubeShare-DevMgr will install and initialize the device library inside the container to isolate its resource usage as described in Section 4.5.

Once all the sharePods attached to a vGPU are deleted by users, KubeShare-DevMgr can either keep the vGPU in vGPU pool as an idle device, or delete the vGPU immediately. If an idle vGPU is kept in the vGPU pool, we can reduce the time for acquiring GPU from Kubernetes when new vGPUs are needed. But an idle vGPU is still recognized as an allocated GPU by the kube-scheduler, so it cannot be utilized by the native Kubernetes pods until it is released from the vGPU pool. Therefore, the decision of when to releasing an idle vGPU presents a tradeoff between performance overhead and resource utilization. We said vGPU is managed in a *on-demand* manner when we release idle vGPUs immediately because a new vGPU must be created upon each request arrives. In contrast, we said vGPU is managed in a *reservation* manner when we tend to keep enough idle vGPUs to handle future vGPU requests. A hybrid strategy can also be designed by using both strategies to manage vGPUs at the same time. We chose the on-demand manner in our implementation because our experiments has shown that the overhead for acquiring vGPUs is quite low and should be acceptable for most use cases.

4.5 GPU Usage Isolation & Elastic Allocation

Resource over-commitment or overuse often leads to performance interference problem. More seriously, it could make applications failed or crashed, like the case when GPU device memory is over-allocated. Hence, besides enabling GPU sharing among containers, we also have to ensure the GPU resource usage of a container can be limited under its requested resource demand.

We isolate the GPU usage among containers in a time-sharing manner using an *token*. Conceptually, a container can only execute its code on GPU when it holds the valid token. The token is associated with a time quota. When the time quota is expired, the container must re-acquire the token again for execution. Hence, containers will utilize a GPU in turn by passing the token among them, and the usage rate of a container can be controlled by the amount of time we allow it to hold the token.

To achieve the aforementioned idea, our solution is implemented by a per-container frontend module and a per-node backend module. The frontend module is a dynamic linking library inside a container. It intercepts all CUDA Library APIs related to memory (e.g., cuMemAlloc, cuArrayCreate) and computing (e.g., cuLaunchKernel, cuLaunchGrid) through the Linux LD_PRELOAD mechanism,

which forces the applications to load our device library before the standard GPU CUDA library. If a container doesn't have a valid token, the frontend module blocks the intercepted CUDA calls until it re-acquires a valid token from the backend module.

The backend module is a standalone daemon running on host machine for managing the token among containers. Since each device is associated with its own token, only one backend module is needed on a host machine to manage the tokens of every devices independently. The backend module has three main tasks: (1) track the GPU usage time of each container; (2) schedule the token to one of the requested containers; (3) determine the time quota of the token.

The GPU usage rate of a container is measured by the time it holds the valid token within a sliding window timeframe. That can be easily tracked by the backend module from the timestamp when a token is assigned to a container. All the token requests from the frontend module is queued by the backend module. When the token is expired by itself or revoked by its holder, the backend module selects one of the requests from the queue and replies the request with a valid token.

Our token scheduling follows the following three steps to elastically allocate residual capacity to containers without violating the resource constraints specified by `gpu_request` and `gpu_limit`. (1) We filter the requests from the container whose usage rate already exceed their maximum usage demand, so the constraint of `gpu_limit` in user's resource specification can be guaranteed. (2) Then we select the request from the container whose usage rate is farthest from its minimum usage demand with the highest priority. Noted, because KubeShare-Sched uses the value of `gpu_request` to assign sharePod on GPUs, the sum of minimum usage demand from all the sharePods on a GPU should be less or equal to the total GPU computing capacity, and the actual usage of a sharePod can be less than its requested demand as well. Therefore, the constraint of `gpu_request` in user's resource specification can always be satisfied. (3) If all the containers requesting for token already reach their minimum usage demand, we pass the token to the one whose current usage is lowest. So the residual capacity can be more fairly allocated among containers.

Finally, we assign a fixed time quota to each valid token. Based on our experimental study in Section 5.2, 100ms time quota is a reasonable choice with sufficient control granularity and little performance overhead. Besides throttling the kernel execution time of containers, we also have to limit their memory allocation space on GPU. In this work, we don't allow memory space to be overcommitted which means the total memory demand from the containers sharing the same GPU must be less or equal to the physical GPU memory size. Therefore, in our current implementation, our frontend module simply throws out of memory exceptions when a container attempts to allocate more space than it requests. As discuss in related works, there are some existing approaches [4, 19, 32] to support memory over-commitment, and our work can be integrated with these solutions to support more flexible GPU memory sharing.

4.6 System Compatibility & Flexibility

Our implementation follows the operator pattern defined by Kubernetes. Operator is a software extension frameworks defined by Kubernetes to provide custom management to a specific kind of application and resource object without modifying the code or changing the behavior of the existing Kubernetes cluster. An operator is a combination of a custom resource and a custom controller. In our case, sharePod is the custom resource definition we added into the kube-apiserver, and KubeShare-DevMgr is the controller we implemented to create and manage sharePod. In comparison, all other existing GPU sharing solutions [2, 7, 10] implement their GPU scheduling and resource accounting logic as a kube-scheduler extender which will affect the whole cluster and monopolize the GPUs. So, we believe our design choice provides more compatibility and flexibility to cluster managers in the following ways.

First, our GPU sharing mechanism and implementation only applies to the sharePod objects that request our GPU sharing service. So KubeShare can co-exist with other GPU management controller or scheduler, and has no influence to the GPU outside the vGPU pool. For instance, in our extended Kubernetes cluster, users can either create a pod with non-shareable GPU through the native pod API or create a pod with shared GPU through our extended sharePod API. In contrast, other solutions based on scheduler extender force all the GPUs in a cluster to be controlled and scheduled by their extended mechanism.

Second, KubeShare decouples the scheduling implementation (i.e., KubeShare-Sched) and the GPU sharing implementation (i.e., KubeShare-DevMgr) into two separate controllers. Therefore, users can implement their own scheduling logic or algorithm to decide the GPUID in SharePodSpec, and interact with KubeShare-DevMgr directly to have the control of GPU locations. In contrast, scheduler extender solutions don't allow users to change or customize the scheduling implementations.

Third and most importantly, KubeShare makes vGPU become a first class resource in Kubernetes where vGPU has unique identity (i.e., GPUID) and can be explicitly requested by users. Because of that, KubeShare is able to support locality constraints in resource description and enable locality aware scheduling. In contrast, kube-scheduler only treats nodes as the first class schedulable entities. While kube-scheduler allows users to specify many scheduling configurations (i.e., such as affinity, anti-affinity, priority), these features only apply at the node level not at the device level. As a result, none of scheduler extender solutions has the ability to let users specify the locality constraints at device level, and thus hard for users to address performance interference issues.

Last but not least, our KubeShare controllers basically act like a wrapper over Kubelet to launch pods with shared GPU. Therefore, any higher level controllers (e.g., replication controller, deployment controller) can seamlessly integrate or adapt to our solution by requesting a sharePod instead of the native pod.

5 EXPERIMENTAL EVALUATION

5.1 Testbed & Workload

The implementation of KubeShare is available at [28]. We conducted our experiments on AWS cloud platform using a Kubernetes cluster consisting of 8 nodes (p3.8xlarge instance type). Each node is

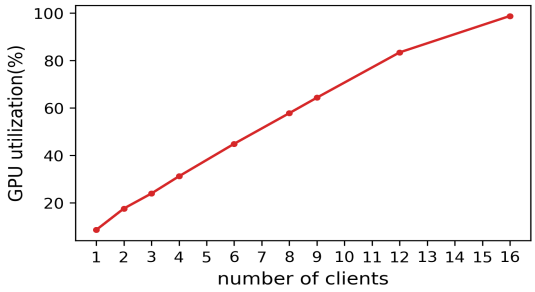


Figure 5: The positive correlation between the GPU usage and the number of client requests for TF-serving, a model inference application of Tensorflow.

Table 3: Deep learning workload.

Job type	Software	Application	Model	Adjusted parameter
Training	Tensorflow	Image Classification	ResNet-50 [12]	training steps
Inference	TF-Serving	Image Segmentation	DeepLab V3 [6]	request numbers

equipped with a 36-cores CPU (Intel Xeon E5-2686 v4), 244GB of RAM, and 4 Nvidia Tesla V100 GPUs with 16GB device memory. We evaluated KubeShare by a set of real deep learning application workloads using Tensorflow [1] as summarized in Table 3. The workloads include the computing jobs for both model training and model inference. Model inference runs the TF-serving application which loads the model into memory and computes the forward propagation upon each arrival client request. Therefore, its GPU usage is roughly proportional to the amount of client requests as shown in as shown Figure 5. This computing characteristic allows us to construct workloads with various GPU demand distributions by adjusting the client request rate of each job when evaluating the benefit of GPU sharing in Section 5.3. For the training jobs, we fixed all the training parameters, and adjusted the number of training steps to control the length of job execution time for the experiments in Section 5.2.

The main performance matrix of our evaluation is the system throughput and GPU utilization. The system throughput is the number of completed jobs per time interval. Since the total jobs is fixed in a workload, the job throughput is also inversely proportional to the overall execution time (i.e.,makespan) of a workload. The overall utilization of a GPU is measured by the GPU usage value time reported by the Nvidia NVML library tool (i.e., Figure 9), while the GPU utilization of individual container (i.e., Figure 6) is measured by the allocated usage time from our vGPU device library.

5.2 GPU Isolation

To ensure containers utilize a shared GPU following theirs requested demand, we must first verify the effectiveness of our GPU device library for GPU isolation. As mentioned in Section 4.5, our GPU device library aims to achieve two goals. One is to throttle the GPU usage of a container under its maximum demand specified

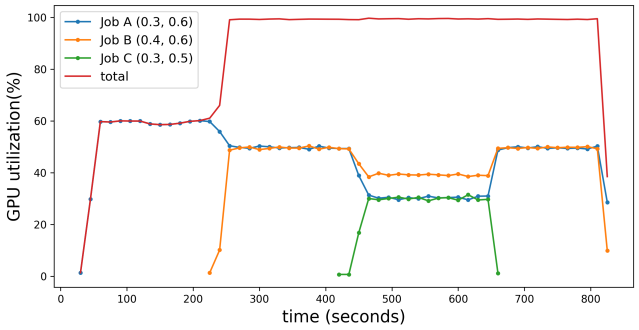


Figure 6: KubeShare ensures GPU isolation among containers according to their resource demands indicated in the bracket (`gpu_request`, `gpu_limit`).

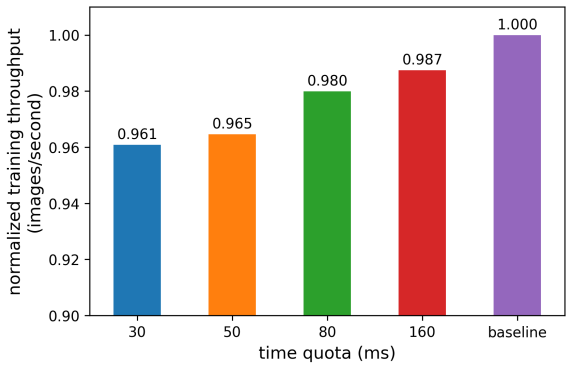
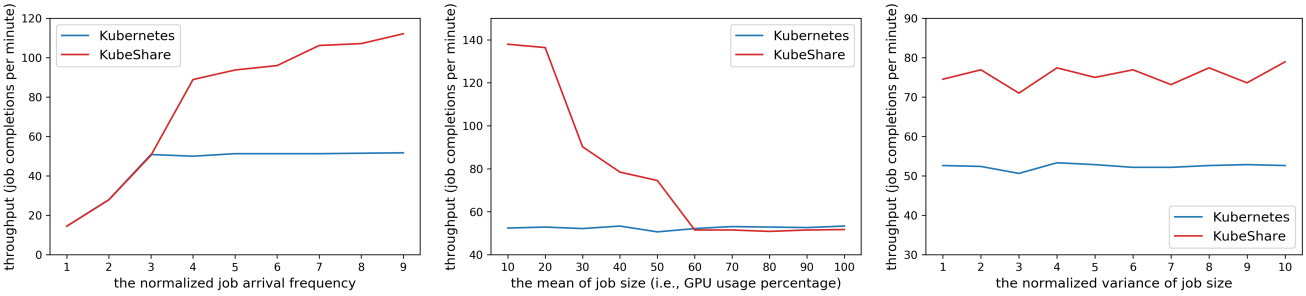


Figure 7: Performance impact from different time quota setting in the vGPU device library. The baseline is the job execution time without using our device library. The time quota setting indicates how often a container must reacquire the token from our device library for executing GPU kernels.

by `gpu_limit`. The other one is to elastically allocate the residual capacity among containers, so that the GPU utilization can be maximized. To prove the correctness and the effectiveness of our device library implementation, we conducted an experiment by running three TensorFlow training jobs on a single GPU. Each job is running inside its own container. Job A arrived at time 0s with the request `gpu_limit`=0.6, and `gpu_request`=0.3. Job B arrived at time 200s with the request `gpu_limit`=0.6, and `gpu_request`=0.4. Finally, Job C arrived at time 400s with the request `gpu_limit`=0.5, and `gpu_request`=0.3.

The actual GPU usage observed from the experiment over time are plotted in Figure 6. As shown by the results, Job A is the only job on the GPU at the beginning, so it could utilize the whole GPU. But our device library successfully limits the GPU usage of Job A to its `gpu_limit` value at 0.6. After Job B arrives at time 200s, the sum of `gpu_request` values from Job A and Job B is only 0.6, so our device library fairly splits the residual capacity among the two jobs without exceeding their `gpu_limit` values. Thus, Job A and Job B each obtains 0.5 GPU usage from time 200s to 400s. Then upon the arrival of Job C, our device library ensures all three jobs receive



(a) Job frequency. **(b) GPU demand mean.** **(c) GPU demand variance.**

Figure 8: Throughput improvement from GPU sharing under various workloads generated by adjusting the job frequency, and the mean and variance of GPU demand per job.

their minimum GPU requirements specified by their corresponding `gpu_request` values. Hence from time 400s to 660s, the GPU usage of Job A, Job B, and Job C are 0.4, 0.3, and 0.3, respectively. After Job C completes its computations at time 660s, we can observed that the residual capacity released from Job C quickly gets redistribute to the other two jobs. Therefore, the overall GPU utilization can be consistently maintained at 100% after time 200s.

From Figure 6, we can also observe that the GPU usage of a job slightly fluctuates at its requested demand. This is due the time quota setting of the token in our implementation. The fluctuation is larger with a larger time quota. But a smaller time quota could cause higher performance overhead because the increasing token exchanging time among containers. To show the performance overhead of our implementation is acceptable, Figure 7 plots the model training throughput (i.e., images/second) of a job under varied token quotes from 30ms to 160ms. The results are normalized to the training throughput of the baseline job execution without our vGPU device library. As shown, even with time quota=30ms, the performance slowdown is still within 5%. Considering the results in Figure 6 was getting from the setting of time quota=100ms, a smaller time quota shouldn't be necessary. Therefore, KubeShare can effectively isolate GPU usage among containers while maximizing the overall GPU utilization with limited overhead.

5.3 GPU Sharing Benefits

Here, we present the results from running a set of workloads on an AWS 8-node Kubernetes cluster equipped with a total of 32 GPUs in order to compare the overall system throughput and GPU utilization between the systems with and without KubeShare extension.

First of all, we compare the system throughput under varied workload patterns, and summarize the results in Figure 8. A workload is consisted of a set of model inference jobs. The job inter-arrival time follows a Poisson process, and the job GPU usage demand is randomly generated from a normal distribution. The throughput values plotted in the figure are the average of 5 experimental runs. Figure 8a is the result when we increase the job frequency by reducing the average job inter-arrival time. When the job frequency is really low, the system has sufficient residual capacity to let each job runs on its own GPU device. So we don't

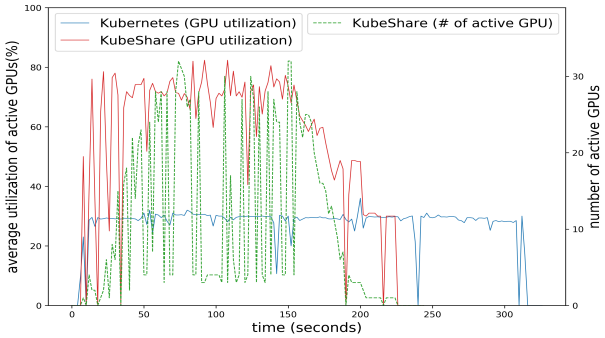


Figure 9: The average GPU utilization and the number of active GPUs over time. The number of active GPUs from Kubernetes is always 32, so it is not shown in the figure.

observe any throughput difference between Kubernetes and KubeShare. However, as the job frequency continues to grow, Kubernetes quickly reaches its maximum throughput at 50 jobs per minutes when job frequency is increased by a factor of 3. On the other hand, KubeShare allows multiple jobs to run on a single GPU as long as their aggregated GPU demand doesn't exceed the physical GPU capacity. Therefore, the maximum throughput of KubeShare can reach 110 jobs per minute, and the system is not saturate until job frequency is increased by a factor of 9. Therefore, GPU sharing can achieve much greater throughput improvement, especially for heavily loaded systems.

Figure 8b is the result when the mean of GPU demand distribution of jobs is varied from 10% to 60% of GPU capacity. The throughput of Kubernetes is agnostic to the change of GPU demand because each job occupies a GPU by itself. In contrast, the increasing GPU demand provides fewer opportunities for GPU sharing, so the throughput of KubeShare also drops as a consequence. But KubeShare achieves ~2.5x throughput improvement when the mean GPU demand is less than 20%. When the mean of GPU demand grows over 60%, there is little chance for GPU sharing among jobs. Therefore, KuberShare cannot provide any performance improvement when there is no GPU sharing opportunities. However, KubeShare doesn't introduce much performance overhead, and thus

the system throughput of KubeShare is almost better or similar to Kubernetes in any GPU workload.

Figure 8c shows that the variance of GPU demand doesn't affect the throughput of both KubeShare and Kubeneters. That can be expected for Kubernetes, because its GPU allocation is agnostic to the GPU demand. On the other hand, for KubeShare, it is likely because a GPU can still be shared between a high demand job and a low demand job regardless the demand variance. Overall, it is clear that the overall system throughput can be significantly improved when GPU sharing is possible.

Finally, we use the workload with GPU mean at 30%, and GPU variance at 2 as an example to show the benefit of GPU sharing on resource utilization. Figure 9 is the average utilization of 32 GPUs reported by Nvidia NVML library over time. We plot the results from a single experimental run in order to observe the utilization fluctuation more clearly. But similar behaviors can be observed from repeated experiments with different workloads. As observed from the plot, because KubeShare improves the GPU utilization of active GPUs, it can reduce the total execution time for completing the whole workload, and produce higher overall system throughput. Furthermore, KubeShare uses less than 32 GPUs in most of the time. In comparison, Kubernetes always uses all 32 GPUs, but still takes longer to complete the workload.

5.4 GPU Sharing Overhead

While GPU sharing provides many benefits, we also need to ensure KubeShare doesn't introduces too much time delay when launching containers. Here, we evaluate the GPU sharing overhead of KubeShare by comparing its end-to-end time delay on creating a user requested pod to the pod creation time of native Kubernetes. As discussed in Section 4.4, the pod creation time of KubeShare can be divided into two parts. The time to launch a native pod to create and register vGPU in vGPU pool, and the time to launch and setup device environment in a sharePod container. Therefore, we plot the pod creation time of KubeShare with and without vGPU creation time in Figure 10. The time of KubeShare without vGPU creation is only about 15% more than the time of native Kubernetes. The additional time delay is mainly caused by to the scheduling and vGPU info query operations. On the other hand, KubeShare with vGPU creation doubles the pod creation time from Kubernetes because it has to launch two pods in the process. However, as shown by the dash line in Figure 10, the actual time delay of creating a pod normally only takes less than a few second. So the pod creation overhead may have limited impact to the long running jobs. Also, the vGPU creation time can be avoided if a suitable vGPU candidate can be directly found from the vGPU pool. Furthermore, the vGPU creation overhead can be entirely ignored at runtime, if we use the reservation mode to let KubeShare pre-allocate all the GPUs in cluster. More importantly, as observed Figure 10, while the pod creation time for both KubeShare and Kubernetes increases with the number of concurrent pod creation requests, the overhead of KubeShare remains constant.

We also report the computing overhead of the scheduling algorithm in Figure 11. According to pseudo-code shown in Algorithm 1, we know its time complexity is $O(N)$, where N is the number of SharePods in the system. Hence, we also found the scheduling time

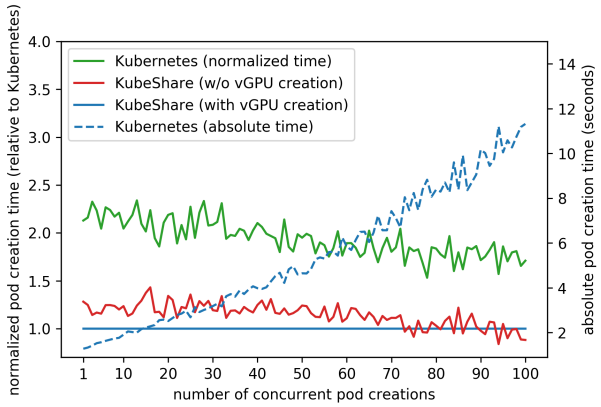


Figure 10: The overhead of KubeShare on pod creation.

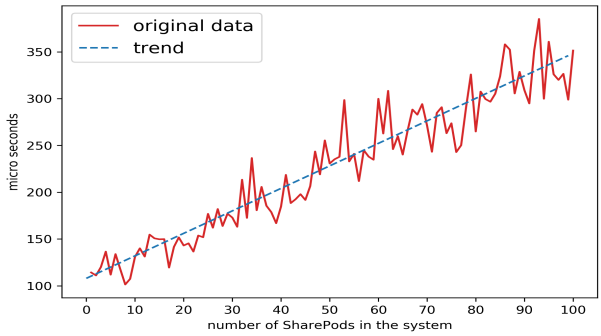


Figure 11: The scheduling time of KubeShare.

increases linearly to the number of SharePods as expected. More importantly, the scheduling time is still less than 400 ms with 100 SharePods. So, our scheduling algorithm is scalable to larger scale systems or workloads.

5.5 Job Interference

Lastly, we conduct experiments to show KubeShare's ability to mitigate performance interference problem with locality aware resource specification and scheduling. In this set of experiment, we construct workloads using two kinds of jobs. The GPU usage requested by Job A and Job B are both less than 50%, so they can share a GPU with each other. However, we let Job A request more GPU resource than it actually needs, so it is more resilient to performance interference caused by resource contention. On the contrary, we let Job B request less than its actual resource usage when it runs alone on a GPU. Therefore, as shown in Figure 12, if two Job Bs are scheduled on the same GPU, they could both suffer from performance interference and increase their execution time by a factor of 1.5x. In comparison, the other combinations involved Job A only cause less than 10% of the performance degradation.

Given the job characteristics, we compare the scheduling results of three cluster settings: Kubernetes, KubeShare without any locality constraint label, and KubeShare with anti-affinity label on Job B. To observe the performance tradeoff between GPU sharing

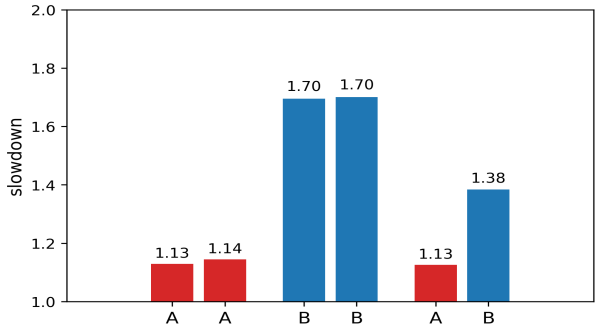


Figure 12: The performance slowdown on a shared GPU for different job combinations: A+A, B+B, and A+B. The slowdown of a job is computed by its execution time on a shared GPU divided by the time on a standalone GPU.

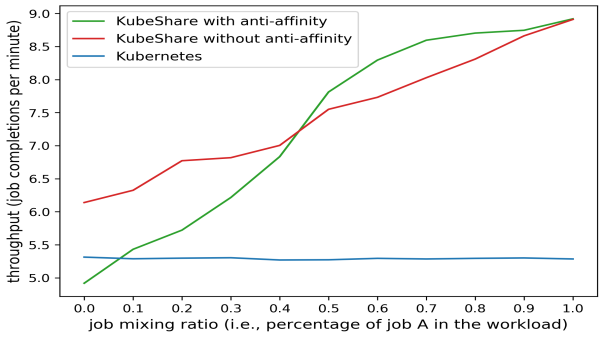


Figure 13: The system throughput comparison under performance interference workloads. Kubernetes doesn't allow GPU sharing between any jobs. KubeShare with anti-affinity doesn't allow GPU sharing between Job B. KubeShare without anti-affinity allows GPU sharing without restriction.

and locality constraints, we vary the ratio of Job A and Job B in the workload to conduct a set of experiments, and summarize the results in Figure 13. As shown in the plot, when all the jobs in workload are Job B (i.e., Job A ratio=0), the behavior of KubeShare with anti-affinity is the same as Kubernetes because both of them only allow a GPU to be scheduled to a single job. In this situation, although KubeShare without anti-affinity can suffer from the performance interference, it actually achieves the highest overall system throughput among the three settings because it fully utilizes the residual resource capacity by GPU sharing. When the ratio of Job A in the workload increases, the throughput of KubeShare with and without affinity both start to improve significantly, but for different reasons. KubeShare with affinity is because of the increasing opportunities of GPU sharing without violating the anti-affinity constraint. On the contrary, KubeShare without affinity is because the decreasing job interference effect from fewer number of Job B is scheduled on the same GPU. When the ratio of Job A in the workload grows over 50%, having the anti-affinity label results in the highest throughput because there is enough Job A to utilize all the GPU resources, and performance interference between Job B can be avoided at the same. Finally, when all the jobs in workload are Job A (i.e., Job A ratio=1), the behavior of two KubeShare settings become the same, and both of them significantly out-perform the native Kubernetes. Overall, we show that KubeShare has the ability to mitigate performance interference problem while sharing GPU, and users can use the locality constraints to achieve the higher system performance.

6 RELATED WORKS

To the best of our knowledge, only the following three existing solutions support GPU sharing in Kubernetes. All of them took a similar approach to support fractional allocation by multiplying the resource unit of a GPU by a scaling-factor. So, their allocation granularity is limited by the value of the scaling-factor. A larger scaling factor provide finer granularity, but also cost more management costs. Deepomatic [7] only support fractional allocation without addressing the fragmentation or resource isolation problems. So Deepomatic [7] can only be applied on a node with single GPU, and provide service quality guarantee. Aliyun [2] is a container service open source project developed by Alibaba. It solves the fragmentation problem by developing a scheduler-extender in Kubernetes. However, it only limits the GPU memory usage of a container, not the computation usage. Finally, GigaGPU [10] is a recent work that further extends Aliyun to support GPU usage isolation on kernel execution based on the LD_PRELOAD API interception technique, like our work. But none of the above solution aims to treat GPUs as first class resource or provide locality aware scheduling to mitigate performance interference problem. Also, all of them are implemented as a scheduler-extender instead of a custom controller in Kubernetes. Hence, their goal is to replace or extend the existing kube-scheduler mechanism, while we prefer KubeShare to co-exist and collaborate with the native kube-scheduler for better flexibility and capability.

GPU virtualization can be realized either by proprietary GPU drivers and kernel modules [17, 18, 24] that installed in OS, or wrapper libraries that dynamically linked to existing applications at runtime [8, 11, 15, 25]. For instance, TimeGraph [17] provides a new set of OS abstractions to support dataflow programming model, and Gdev [18] provides an explicit API set to share device memory among GPU contexts. On the other hand, wrapper libraries, also called API-remoting, perform customized GPU scheduling and management by intercepting GPU calls from an application, before the calls reach the GPU driver. Including our work, and many others, like ConVGPU [15], Pegasus [11], rCuda [23], and gVirtuS [8], are all based on this approach wihout re-implementing the GPU driver. In this work, we also choose to use the wrapper library approach, so no modification requirements on OS kernel or applications.

Supporting memory over-commitment on GPU is also important topic for GPU sharing, especially for memory-intensive jobs. Several approaches [4, 19, 32] have been proposed based on the virtual memory mechanism, so that the GPU memory content can be swapped to host memory when its GPU kernel is not running. Although these techniques can increase the chances of GPU sharing, they also have the risk to introduce more performance overhead from the memory swapping operations due to the limited memory

bandwidth between host and device. GPU kernel preemption is another topic related to GPU sharing. Because of the non-preemptive natural of CUDA kernel execution, a long running GPU kernel can potentially monopolize a shared GPU. Hence, several works aim to enable kernel preemption by breaking kernels into multiple sub-size kernels. Some of these approaches require users to rewrite their code in a new programming [26]; others rely on automated code generation at the compile time or kernel runtime [3]. For instance, FLEP [31] uses a compiler-time engine to generate sub-size kernel execution code, and further optimize the execution order these kernels according to builds a performance model. In general, all these approaches impose certain restrictions on applications, driver, or hardware. In this work, we mainly focus more on enabling GPU sharing and first class resource management in Kubernetes. Therefore, we chose to design and implement a GPU isolation solution that has less overhead and restrictions.

7 CONCLUSIONS

In this work, we developed KubeShare, a framework to support GPU sharing in Kubernetes. In order to address the GPU utilization and performance interference problems in shared resource environment, KubeShare manages GPUs as first-class resources, and provides a simple yet powerful resource specification to let users to request specific vGPU binding for their containers using the GPU identifier, and to control the location of their GPU by specifying the locality constraints. Through a device library implementation based on CUDA API interception, KubeShare not only can isolate the GPU usage of a container, but it also can elastically allocate residual GPU capacity to containers without exceeding their maximum demand. We followed the operator pattern defined by Kubernetes to implement KubeShare as a set of extended custom controllers in Kubernetes cluster, so KubeShare requires very little efforts for installation and it is compatible to other existing Kubernetes components without code modification. We extensively evaluated the performance and overhead of KubeShare using a set of real deep learning workloads. The results proved that KubeShare can significantly increase GPU utilization and throughput upto a factor of 2x with less than 10% performance overhead during container initialization and execution. Our implementation is open-source and available at [28].

REFERENCES

[1] Martín Abadi and et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Alibaba Cloud. GPU Sharing Scheduler Extender in Kubernetes. [Online]. Available: https://github.com/AliyunContainerService/gpushare-scheduler-extender.

[3] C. Basaran and K. Kang. Supporting preemptive task executions and memory copies in GPGPUs. In *Euromicro Conference on Real-Time Systems*, pages 287–296, July 2012.

[4] Michela Becchi, Kittisak Sajjapongse, Ian Graves, Adam Procter, Vignesh Ravi, and Srimat Chakradhar. A Virtual Memory Based Runtime to Support Multi-Tenancy in Clusters with GPUs. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, page 97–108, 2012.

[5] Maxim Belkin, Roland Haas, Galen Wesley Arnold, Hon Wai Leong, Eliu A. Huerta, David Lesny, and Mark Neubauer. Container solutions for hpc systems: A case study of using shifter on blue waters. In *Proceedings of the Practice and Experience on Advanced Research Computing*, 2018.

[6] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *CoRR*, 2016.

[7] Deepomatic. A shared GPU Nvidia K8S device plugin. [Online]. Available: https://github.com/Deepomatic/shared-gpu-nvidia-k8s-device-plugin.

[8] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, and Giuseppe Coviello. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing*, page 379–391. Springer-Verlag, 2010.

[9] Google. Kubernetes cluster management. [Online]. Available: http://kubernetes.io/.

[10] J. Gu, S. Song, Y. Li, and H. Luo. GaiaGPU: Sharing GPUs in Container Clouds. In *2018 IEEE Intl Conf on Parallel Distributed Processing with Applications, Ubiquitous Computing Communications, Big Data Cloud Computing, Social Computing Networking, Sustainable Computing Communications*, pages 469–476, Dec 2018.

[11] Vishakha Gupta, Karsten Schwan, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. Pegasus: Coordinated scheduling for virtualized accelerator-based systems. In *USENIX Annual Technical Conference*, page 3, 2011.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, 2015.

[13] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *USENIX Conference on Networked Systems Design and Implementation*, pages 295–308, 2011.

[14] Intel. Intel device plugin for kubernetes. [Online]. Available: https://github.com/intel/intel-device-plugins-for-kubernetes.

[15] D. Kang, T. J. Jun, D. Kim, J. Kim, and D. Kim. ConVGPU: GPU Management Middleware in Container Based Virtualized Environment. In *IEEE International Conference on Cluster Computing*, pages 301–309, Sep. 2017.

[16] H. Kang, M. Le, and S. Tao. Container and microservice driven design for cloud infrastructure devops. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*, pages 202–211, April 2016.

[17] Shinpei Kato, Karthik Lakshmanan, Ragunathan Rajkumar, and Yutaka Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *USENIX Annual Technical Conference*, page 2, USA, 2011. USENIX Association.

[18] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. Gdev: First-class GPU resource management in the operating system. In *USENIX Annual Technical Conference*, page 37, USA, 2012. USENIX Association.

[19] Jens Kehne, Jonathan Metter, and Frank Bellosa. GPUswap: Enabling Oversubscription of GPU Memory through Transparent Swapping. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, page 65–77, New York, NY, USA, 2015. Association for Computing Machinery.

[20] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. Singularity: Scientific containers for mobility of compute. *PLOS ONE*, 12(5):1–20, 05 2017.

[21] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[22] Nvidia. NVIDIA GPU Device Plugin for Kubernetes. [Online]. Available: https://github.com/NVIDIA/k8s-device-plugin/.

[23] Antonio J. Peña, Carlos Reaño, Federico Silla, Rafael Mayo, Enrique S. Quintana-Ortí, and José Duato. A complete and efficient cuda-sharing solution for hpc clusters. *Parallel Comput.*, 40(10):574–588, December 2014.

[24] Chris Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: Operating System Abstractions To Manage GPUs as Compute Devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, page 233–248, October 2011.

[25] Dipanjan Sengupta, Raghavendra Belapure, and Karsten Schwan. Multi-Tenancy on GPGPU-Based Servers. In *Proceedings of the 7th International Workshop on Virtualization Technologies in Distributed Computing*, page 3–10, 2013.

[26] Yusuke Suzuki, Hiroshi Yamada, Shinpei Kato, and Kenji Kono. GLoop: An Event-Driven Runtime for Consolidating GPGPU Applications. In *Symposium on Cloud Computing*, page 80–93, 2017.

[27] Tencent. Rdma device plugin for kubernetes. [Online]. Available: https://github.com/hustcat/k8s-rdma-device-plugin.

[28] Ting-An Yeh, Jerry Chou. Kubeshare. [Online]. Available: https://github.com/NTHU-LSALAB/KubeShare/.

[29] Vinod Kumar Vavilapalli et al. Apache hadoop yarn: Yet another resource negotiator. In *Symposium on Cloud Computing*, pages 5:1–5:16, 2013.

[30] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 18:1–18:17, 2015.

[31] Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang. Flep: Enabling flexible and efficient preemption on gpus. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, page 483–496, 2017.

[32] Mochi Xue, Kun Tian, Yaozu Dong, Jiacheng Ma, Jiajun Wang, Zhengwei Qi, Bingsheng He, and Haibing Guan. gScale: Scaling up GPU Virtualization with Dynamic Sharing of Graphics Memory Space. In *USENIX Annual Technical Conference*, pages 579–590, June 2016.

[33] A. J. Younge, K. Pedretti, R. E. Grant, B. L. Gaines, and R. Brightwell. Enabling diverse software stacks on supercomputers using high performance virtual clusters. In *IEEE International Conference on Cluster Computing*, pages 310–321, Sep. 2017.