KubeComp: A Resource-centric Composable Container Orchestrator for GPU Pooling

Hsu-Tzu Ting
National Tsing Hua University
Hsinchu, Taiwan
hsutzu.ting@lsalab.cs.nthu.edu.tw

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

Ming-Hung Chen
IBM Research
New York, US
minghungchen@ibm.com

I-Hsin Chung
IBM Research
New York, US
ihchung@us.ibm.com

Huaiyang Pan
H3 Platform Inc.
Taichung, Taiwan
brian.pan@h3platform.com

***Abstract*—Composable infrastructure, designed to allocate hardware from a disaggregated resource pool dynamically, aligns seamlessly with the resource-centric nature of cloud computing. Although a few GPU disaggregation systems for reconfigurable bare metal devices exist, the corresponding software stacks are lagging. In this work, we designed and implemented KubeComp, the first solution for Kubernetes to support GPU pooling based on a composable infrastructure. KubeComp makes Kubernetes aware of the underlying composable infrastructure and provides on-demand, optimized, and automated resource management for service deployment. With the support of KubeComp, resource utilization is boosted, and the job wait time is reduced by up to 89% for running jobs with diverse CPU-GPU ratios.**

***Index Terms*—Composable, GPU pooling, Container, Scheduling, Cloud Computing**

I. Introduction

Cloud computing provides on-demand access to memory, storage, and accelerators, allowing users to pay only for what they use. This pay-as-you-go cloud computing model frees businesses from the constraints of on-premise IT infrastructure, reduces operational expenses, and enhances agility. Although most accelerators are expensive, cloud computing makes them accessible to businesses. Cloud service providers manage the necessary infrastructure, making hardware efficiency a crucial concern. Higher hardware efficiency leads to greater resource utilization and the ability to serve more users.

Virtualization maximizes efficiency by enabling multiple virtual machines to run on a single physical server. However, it does not support flexible resource provisioning among nodes. In traditional data centers, resources are tightly coupled with nodes and cannot be dynamically reallocated. Such *server-centric* architecture causes resource fragmentation because resource allocation is limited within a node [15]. Although techniques like rCUDA [8] and DS-CUDA [32] are proposed to enable GPU sharing among nodes by redirecting CUDA API calls to remote GPUs, the network overhead is noticeable and requires complicated decisions when scheduling [1], [10].

Composable infrastructure solves this problem by providing a fluid environment where users can allocate resources dynamically to build up the systems [14]. It utilizes disaggregated resources, which are decoupled from the servers and form a resource pool shared among servers [2], [5], [24]. Unlike API remoting, composable infrastructure reconfigures hardware resources at the bare-metal level, eliminating the network concern [6], [7], [17], [19], [25], [34]. This resource-centric design empowers users to allocate resources based on their specific needs dynamically and efficiently.

Although composable infrastructure brings benefits, the modern resource management software does not support the resource-centric design well. In particular, traditional resource managers, such as job schedulers or resource allocators, are unaware of the underlying composable infrastructure and ignore the possibility of resource reconfiguration during system uptime. This design limits the benefit of the composable infrastructure to the construction level and does not bring the benefit to the end user. The end user cannot compose the environment they need promptly because the resource manager doesn't reflect the latest resource allocation after reconfiguring the underlying resources.

Seeing the lack of software support for composable infrastructure, we aim to develop a resource management middleware that enables resource-centric management on the composable infrastructure. We designed and implemented our proposed solution, *KubeComp*, on Kubernetes [13], one of the most popular container orchestrators on the cloud. We initially focus on GPU resources due to the rapid growth of the machine learning workloads in the cloud, which often require a mix of CPU and GPU usage [16] with diverse resource demands during training and inference [35]. This variability poses significant challenges to resource utilization for cloud service providers. KubeComp addresses these challenges by providing three crucial features: **on-demand**, **optimization**, and **automation**. On-demand resource provision satisfies the cloud service agility, reconfiguration optimization enhances resource utilization through intelligent config-

uration decisions, and deployment automation simplifies resource management for users.

To the best of our knowledge, we are the first to present the idea of resource-centric management in Kubernetes. We summarize the contributions of our paper as follows:

- We proposed KubeComp, which introduces resource-centric management to Kubernetes atop non-vendor-specific composable infrastructure.
- We followed the KubeComp framework and implemented a GPU pooling solution on top of the PCIe fabric-based GPU pooling infrastructure.
- We elaborated an optimized reconfiguration mechanism for GPU pooling that is automated and on-demand to meet the user workload requirement and maximize resource utilization.
- We ran diverse workloads with various CPU-GPU ratios on KubeComp to show a notable improvement in job waiting time and system resource utilization compared with the default server-centric Kubernetes environment.
- We conducted simulations using real cluster traces to validate the performance gain of resource-centric management on composable systems.
- We open-sourced KubeComp implementation and is publicly available. [27].

II. Background

KubeComp aims to let Kubernetes become a resource-centric cluster manager for orchestrating a disaggregated resource pool. Hence, we illustrate an overview of Kubernetes and then introduce the composable infrastructure we used for the GPU pool, namely, the PCIe fabric GPU chassis.

A. Kubernetes Overview

Many companies embrace microservices that break down an application into multiple components with specific functions. These components can be managed independently and are commonly run in containers due to scalability, portability, and agility. The increasing number of containers highlights the need for containerized application orchestration, which features scheduling, deploying, updating, and scaling the containerized applications. Kubernetes is a popular and reliable distributed system platform for container orchestration on the cloud. It runs workloads by placing containers into *pods* to run on *nodes*. Nodes can be either virtual or physical, playing the role of worker machines. Pods are the smallest deployable computing units in Kubernetes, comprising one or more containers. Our work is based on the scheduling and device plugin framework defined by Kubernetes. Hence, we describe them in more detail below.

1) Scheduling Framework: When the kube-scheduler detects a newly created pod without an assigned node, it starts a process to find an appropriate node, following the scheduling framework [22]. It includes the scheduling the binding cycle. During the scheduling cycle, nodes that do not meet the pod's requirements are filtered out, and the remaining nodes are scored based on scoring functions. The node with the highest score is then selected and assigned to the pod during the binding cycle. The scheduling framework allows customization through extension points such as the `score` extension for alternative ranking criteria and the `permit` extension to prevent or delay the pod from entering the binding process.

2) Device Plugin: Kubernetes natively supports CPU and memory discovery and avoids integrating specific hardware for sustainability [11]. Instead, Kubernetes contains a device manager inside Kubelet, which interacts with device plugins developed by the hardware vendor. These plugins are gRPC servers with specific gRPC interfaces, enabling the device manager to manage the hardware. Through device plugins, devices such as GPUs, high-performance NICs, FPGAs, etc., can be integrated into the Kubernetes cluster.

The device plugin is arranged in `Registration`, `ListAndWatch`, and `Allocate`. During `Registration`, the device plugin advertises its presence to Kubelet and starts the gRPC server. After successful registration, Kubelet can interact with the device plugin through the gRPC interfaces like `ListAndWatch` and `Allocate`. Kubelet discovers the devices via `ListAndWatch`, and the device plugin also notifies Kubelet via `ListAndWatch` whenever the device state changes or a device disappears. `Allocate` is called by Kubelet when the containers are about to be created. The device plugin may do device-specific preparation and response to the gRPC call with container runtime configurations. For example, to force the container to use a specific GPU, `Allocate` can set the container's environment variables `NVIDIA_VISIBLE_DEVICES` to GPU UUID [31].

B. PCIe Fabric GPU Chassis

GPU chassis disaggregates GPUs from the servers, and servers attached to the chassis share the GPUs inside, forming a GPU pool. The interconnect between servers and GPU chassis is based on Peripheral Component Interconnect Express (PCIe) [29], a serial expansion bus standard for connecting peripheral devices to a computer. Using PCIe fabrics for the resource pool improves GPU utilization by deploying the exact amount of GPUs to the server for the workload. It requires no change to the software stack, allowing the user programs to access GPU transparently.

The example topology of the GPU chassis is shown in Fig. 1. Inside the GPU chassis is a PCIe switch with an ARM processor. The PCIe switch's upstream ports (USP) are connected to hosts, while the downstream ports (DSP) are connected to GPUs. The ARM processor maintains a Device Lookup Table (DLUT) that maps device ports to host ports. Whenever the GPUs are reallocated, the ARM processor updates DLUT, allowing the host to recognize the new GPU.

To reconfigure GPU allocation on-demand without rebooting machines, both the operating system and de-

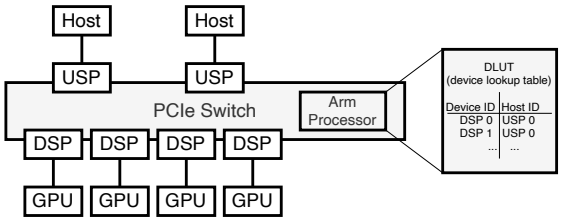


Fig. 1: The example topology of the GPU chassis.

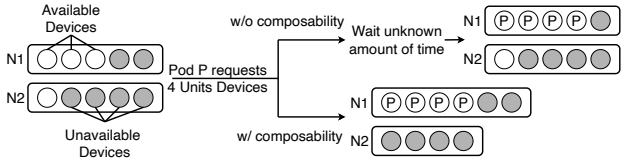


Fig. 2: Comparison between the systems with and without composability.

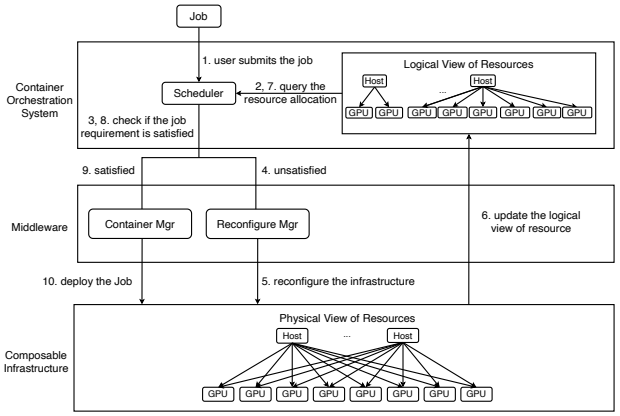


Fig. 3: The integration and workflow of the composable infrastructure. The reconfiguration is driven by user requests, and the deployment is automated.

vice driver have to support PCIe hot-plugging. Most of the system can satisfy the requirement since the Linux kernel completed the PCIe hot-plug around 2020, and the NVIDIA driver enables GPU hot-plugging when the persistent mode is turned off.

III. Resource Centric Design

A. Motivation

The key advantage of a resource-centric composable system is its flexibility to reconfigure or re-distribute resources among nodes during system runtime. As illustrated in Fig. 2, if the system is composable and applies resource-centric management, the pod can be scheduled once the resource is reconfigured by moving one available device from *N2* to *N1*. In contrast, the server-centric scheduler without composability doesn't consider the potential of resource reallocation, leaving the pod unschedulable.

To obtain the strength of composable infrastructure, we need the resource manager to be aware of the composability and allocate resources in a resource-centric manner. Traditional cluster managers, such as the native Kubernetes scheduler, don't know when and how to manage composable infrastructure and assume the resources on a node are fixed. Existing solutions often require system administrators to manually reconfigure the infrastructure, followed by rebooting the nodes to detect changes. The loose coupling between container orchestrators and composable infrastructure results in long response time for end users and unwanted service interruptions.

B. Workflow

We propose KubeComp and implement a resource-centric resource management middleware to integrate container orchestrators and the composable infrastructure. Fig. 3 illustrates the workflow of our proposed resource-centric management middleware. After the user submits jobs (step 1), the resource-centric scheduler queries the resource allocation (step 2) to decide whether the job can be scheduled (step 3). If the current resource configuration cannot satisfy the job requirement (step 4), the reconfiguration process will be triggered. The resource-centric container orchestrator carefully selects resources from the pool and reallocates them to the target server through the interfaces provided by the composable infrastructure vendors (step 5). After the reconfiguration, the logical view of resources is updated accordingly (step 6). Once the scheduler detects the change in resource allocation (step 7), which matches the job requirement (step 8 and 9), the job is deployed successfully (step 10).

Our proposed solution achieves three primary design goals: **on-demand**, **optimized**, and **automated**. The resource reconfiguration is on-demand driven by the user requests (steps 1-4); an optimized reconfiguration plan is given by our manager to maximize resource utilization and minimize reconfiguration cost (step 5); finally, the entire process from job submission to infrastructure reconfiguration is automated without service or system interruption (steps 6-10).

We noticed that only a few works (elaborated in Section VII-B) attempt to integrate the composable infrastructure with the container orchestrator, and none of them achieves three primary design goals. Seeing this incompleteness, we designed KubeComp to bridge the gap between the underlying composable infrastructure and the upper-layer container orchestration with strategic reconfiguration. To the best of our knowledge, we are the first to propose the middleware solution for Kubernetes to support resource-centric management.

IV. Design & Implementation

KubeComp is a framework that follows the resource-centric design presented in Section III. It introduces composability to the Kubernetes cluster, and we used GPU as the initial case, as GPU is a critical resource for the increasingly popular machine learning workloads in the cloud. At the end of the section, we also elaborate on how to adapt KubeComp to other disaggregated resources.

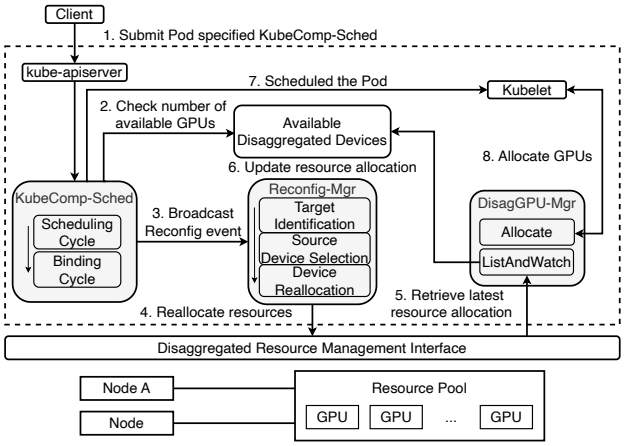


Fig. 4: The components and workflow of KubeComp.

A. Challenges

To realize the composable Kubernetes for GPU pooling, we must address the following three challenges:

Kubernetes Compliance. To keep the core of Kubernetes simple and maintainable, Kubernetes is not encouraged to be modified arbitrarily. KubeComp complies with the Kubernetes architecture by enabling features with the scheduling plugin and the device plugin.

Composability Awareness. Without composability, kube-scheduler selects the node for the pod to run on according to the node status at the time the pod enters the scheduling cycle. However, KubeComp allows reconfiguration, so the scheduler has to consider the node status after reconfiguration. The resolution of this challenge is presented in Section IV-C and Section IV-D.

Disaggregated Device Support. Kubernetes natively supports CPU and memory, while GPU is exclusive. Although Kubernetes allows device vendors to develop device plugins, the NVIDIA device plugin [30] doesn't match well with composability. It constantly holds GPU resources, which prevents us from reconfiguring. We designed our own device plugin presented in Section IV-E to overcome this challenge.

As our work aims to bring composability into Kubernetes, some aspects are currently not our goal. First, we don't handle the overhead brought from the hardware. Hardware-specific optimization is outside our scope as we aim to deal with diverse hardware. Second, we target to maximize resource utilization by resource reconfiguration and do not consider job-level scheduling. In this paper, we operate jobs in the first-come-first-serve order.

B. Architecture Overview

The main difference between the traditional Kubernetes and KubeComp is that the GPUs can be reallocated. KubeComp composes the desired environment for pods on demand, causing a decrease in wait time and an increase in overall resource utilization. KubeComp includes three components: *KubeComp-Sched*, *DisagGPU-DevMgr*, and *Reconfig-Mgr*. They overcome the challenges of GPU pooling in Kubernetes and bring composability.

Fig. 4 shows the workflow of KubeComp. After the client submits a pod intended to use the disaggregated GPUs by specifying the specialized scheduler (step 1), KubeComp-Sched is responsible for finding a suitable node with adequate resources for scheduling. If no node meets the requirement, KubeComp-Sched checks whether a suitable node can be composed by GPU reallocation (step 2). If feasible, a reconfiguration event is broadcasted (step 3), triggering the reconfiguration process managed by Reconfig-Mgr (step 4). Subsequently, DisagGPU-DevMgr retrieves the updated resource allocation (step 5) and reflects the correct allocation to the kubelet (step 6). KubeComp-Sched notices the resource change and lets the pod transition from the scheduling cycle to the binding cycle (step 7). Finally, as the pod is about to run, the kubelet interacts with DisagGPU-DevMgr for GPU allocation (step 8).

C. KubeComp-Sched

When KubeComp-Sched notices an incoming pod, it first checks if the total number of GPUs in the cluster is sufficient. If this requirement is met, KubeComp-Sched does not filter out the node with insufficient GPU because the desired number of GPUs can be composed later. KubeComp-Sched assigns scores to the remaining nodes and selects the one with the highest score as the designated node for running the pod.

If the designated node does not own sufficient GPUs for the pod, KubeComp-Sched will broadcast the *Reconfig Event* and annotate the pod with the designated node and the GPU demand. Suppose pod *P* requests 4 GPUs but is assigned to node *N1* with 2 GPUs attached, it will be annotated with "`dst_node: N1`" and "`demand: 2`". KubeComp-Sched confirms the completion of the reconfiguration process by verifying if the number of GPUs attached to the designated node matches the pod's requirement. The pod is permitted to proceed to the binding cycle only when the node obtains sufficient resources.

D. Reconfig-Mgr

To enhance Kubernetes with composability, we deploy Reconfig-Mgr to reconfigure the allocation of GPU. Reconfig-Mgr is a daemon that detects the reconfiguration demand by monitoring the Kubernetes events. After detecting the *Reconfig Event* broadcasted by KubeComp-Sched, it will start the reconfiguration process, which includes target identification, source GPU selection, and GPU reallocation.

Target identification specifies the number of GPUs to reallocate and their new location. When Reconfig-Mgr detects the Reconfig Event, it first decodes the event to know which pod is involved. Next, it inspects the pod's annotations, `dst_node` and `demand`, indicating that the Reconfig-Mgr has to reallocate `demand` GPUs to node `dst_node`.

Source device selection decides the GPUs to choose from when reallocating GPUs. It maintains a sorted list of

source GPU candidates. Reconfig-Mgr ensures that these GPUs are available and can be reallocated.

Device reallocation is carried out via the exposed method provided by reconfigurable devices. The composable infrastructure vendors provide common interfaces for consumers to manage their devices, typically including APIs for allocating and deallocating resources. Reconfig-Mgr iterates through the source GPU candidates and performs the reconfiguration until the GPU demand is satisfied.

E. DisagGPU-DevMgr

Kubernetes inherently supports resources like CPU and memory, but a device plugin is necessary to expose additional resources like GPUs. As we consider GPUs to be our initial disaggregated devices in the framework, we have tried the NVIDIA device plugin. However, it does not match our requirements due to the following two reasons:

1) The NVIDIA device plugin utilizes the NVIDIA Manage Library (NVML), which retains control of the GPU even when the GPU is not computing. Once NVML is initialized, the GPUs are held by the library and can not be reallocated.
2) The NVIDIA device plugin does not anticipate changes in GPU allocation after the plugin is initialized. It lacks support for regular GPU allocation updates except for the health check. Moreover, GPUs in an unhealthy state cannot recover unless restarting the device plugin, which also does not match our goal.

These reasons motivate us to design our device plugin, DisagGPU-DevMgr. It does not use the library holding GPU resources and views the allocation as reconfigurable. It follows the Kubernetes device plugin design framework with interfaces `ListAndWatch` and `Allocate`.

DisagGPU-DevMgr maintains a device table that stores GPUID and UUID as key-value pairs. GPUID represents the identifier within the GPU pool, while UUID is the identifier of the physical GPU. When `ListAndWatch` is called, DisagGPU-DevMgr polls the common interface provided by the composable infrastructure vendor to retrieve GPU allocation as well as GPUID and UUID. The GPUID is returned to the kubelet, and both values are stored in the device table.

`Allocate` function is called when a container is about to be created. The kubelet specifies the GPUID it aims to allocate to the container, and DisagGPU-DevMgr queries the device table to retrieve the corresponding UUID. It then sets GPUID and UUID as environment variables within the container to ensure that the pod utilizes the specific GPU.

F. Compatibility

KubeComp framework can expand Kubernetes support for various disaggregated resources and composable infrastructure vendors if the following three conditions are met. First, the resource can be reconfigured without a system restart. Second, the resource reconfiguration and allocation retrieval methods are exposed by the vendors. Third, the resource demand can be discretely described and is exclusively used by a single container, as required by the device plugin framework.

KubeComp has high compatibility due to its design, which leverages composable infrastructure with exposed resource management APIs. DisagGPU-DevMgr uses the resource allocation retrieval API, while Reconfig-Mgr uses the resource reconfiguration API. Therefore, KubeComp is not limited to specific resources or interconnections of the composable infrastructure; these are managed by vendors, allowing KubeComp to focus on reconfiguration decisions.

V. Resource Management

This section demonstrates the resource management strategies with optimization, analyzes the overall time complexity, and discusses strategies for extending resource management from a single resource pool to multiple pools.

A. Management Strategies

The strategies aim to handle reconfiguration correctly and efficiently. When KubeComp-Sched selects the node for pods, it adapts *best-fit* or *least-demand* based on the availability of resources for a shorter job wait time. As for Reconfig-Mgr, it carefully selects the GPU candidates for reallocation to reduce GPU fragments.

1) Node Selection: KubeComp-Sched selects a node for the pod to run on, and a good decision should deliver the desired environment promptly. When choosing the node for the pod that requires GPU resources, KubeComp-Sched employs *best-fit* when the node has sufficient resources and employs *least-demand* when reconfiguration is necessary. These strategies are implemented as the node scoring function presented in Equation 1. $reqGPU$ denotes the requested number of GPUs for the pod requests, and $availGPU$ denotes the available GPUs on the node.

$$score(reqGPU, availGPU) = \begin{cases} \frac{reqGPU}{availGPU} \times 100\%, & \text{if } availGPU \geq reqGPU \\ availGPU - reqGPU, & \text{otherwise} \end{cases} \tag{1}$$

The *best-fit* strategy aims to minimize the need for future reconfiguration. KubeComp selects the node with the least yet sufficient GPUs, ensuring that nodes with higher GPU capacities remain available for GPU-demanding pods. When reconfiguration is inevitable, KubeComp shifts to the *least-demand* strategy. After filtering out the nodes lacking sufficient CPU and memory, KubeComp-Sched selects the node with the least GPU demand. This strategy triggers fewer GPU reallocation API calls, leading to faster environment composition and reduced wait times for pod execution.

2) GPU Candidates Identification: Reconfig-Mgr is responsible for moving the desired number of GPUs to the designated node. Deciding which GPU to move is similar to the Kubernetes scheduling framework. We first select GPU candidates and then apply the GPU scoring function. Candidates with higher scores will be reallocated.

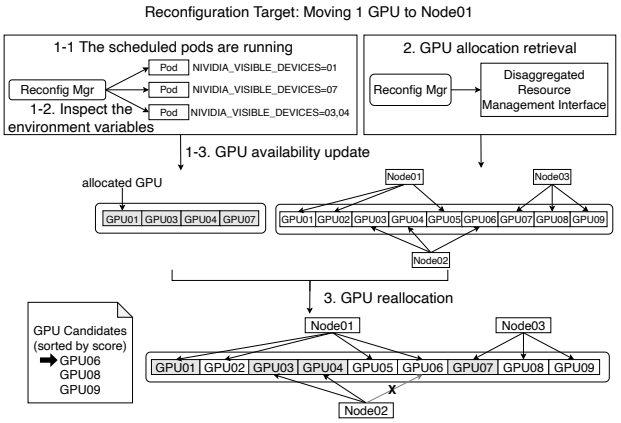


Fig. 5: The process of GPU candidate identification. GPU06 is reallocated from Host02 to Host01 in this example.

GPU candidates are the available GPUs not attached to the designated node yet. The process of GPU candidate identification is shown in Fig. 5. DisagGPU-Mgr passively receives the resource allocation request from the kubelet and declares `NVIDIA_VISIBLE_DEVICES` to specify GPU usage for pods. When Reconfig-Mgr needs to determine GPU availability, it queries pod environment variables to identify allocated GPUs (step 1). The disaggregated resource management interface layer, utilizing common operations provided by the PCIe fabric chassis vendors via RESTful or Redfish APIs, provides information about all GPUs in the GPU pool (step 2). The difference between these two sets yields available GPUs, and the GPU candidates are those not located on the designated node. Reconfig-Mgr sorts the GPU candidates using the GPU scoring function and selects GPU for reallocation accordingly (step 3).

The GPU scoring function aims to minimize GPU fragments, which are the available GPUs allocated to the same node but insufficient to serve jobs immediately. The GPU scoring function is presented in Algorithm 1. Given a list of GPUs denoted as $gpuCandidates$, it assigns a score for each GPU. Each GPU has properties *node*, indicating the node that the GPU is currently allocated to, and *score*, which is calculated by the GPU scoring function. The function first iterates through $gpuCandidates$ to count GPUs on each node and then assigns the score based on the negative value of this count. The GPU scoring function prioritizes the GPUs on nodes with fewer GPUs because they are more likely to become fragments.

B. Time Complexity

The reconfiguration process can be summarized into five phases, and this section analyzes the complexity of each phase. Throughout the analysis, the notation n is the number of nodes, while the notation m is the number of reallocated GPUs.

Node Selection. KubeComp-Sched selects a suitable node by filtering out the unsuitable ones and assigning scores to the remaining using the $O(1)$ node scoring function (Equation 1). Therefore, the time complexity of this phase is $O(n)$.

Algorithm 1: GPU Scoring Function

```
/* Assign the score for gpuCandidates */
nodeGPUCnt[numOfNode] ← {0, 0, ..., 0}
foreach gpu ∈ gpuCandidates do
    nodeGPUCnt[gpu.node] ← nodeGPUCnt[gpu.node] + 1
end
foreach gpu ∈ gpuCandidates do
    gpu.score ← −nodeGPUCnt[gpu.node]
end
```

GPU availability update. Reconfig-Mgr inspects the running pods in the cluster to identify GPU availability. As the maximum number of pods per node is limited according to Kubernetes best practice [23], the time complexity of this phase is $O(n)$.

GPU allocation retrieval. As far as we know, the PCIe fabric chassis from all vendors provides an API to retrieve GPU allocation. Regardless of the number of GPU chassis, retrieval can be performed in parallel, leading to $O(1)$ time complexity.

GPU reallocation. Reconfig-Mgr applies GPU scoring functions on each GPU as shown in Algorithm 1 and reallocates the demanded number of GPUs. Given that the number of GPUs is linear with the number of nodes due to the design of the GPU chassis, and the reallocations have to be serialized, the time complexity of this phase is $O(n) + O(m)$.

Scheduler detection and binding. After the reconfiguration, the time it takes for the scheduler to notice depends on the design of Kubernetes, which can be expected to be bounded.

The overall time complexity is $O(n) + O(m)$. Considering n as a constant since the cluster scale is fixed, the reconfiguration time of our method is only related to the reallocated GPUs. The number of reallocated GPUs can be reduced by carefully selecting the designated node.

C. Multi-Pool Scenario

The capacity of the GPU chassis is constrained by hardware limitations, such as the length of the PCIe cable and the tolerable error rate in PCIe links. For example, each H3 Falcon 4005 GPU expansion box only connects to four GPUs at most, Liqid SmartStack has a maximum capacity of 30 GPUs, and the GigaIO SuperNODE supports up to 32 GPUs. To accommodate more GPUs, multiple GPU chassis are required, as illustrated in Fig. 6. Each pool is isolated and connected to a limited number of nodes. The node can only access GPUs that belong to the same GPU pool.

To extend the management strategies of a single GPU pool to the multi-pool scenario, the node selection strategy requires a slight change. Under the multi-pool scenario, the node selection process involves filtering and scoring. Nodes are filtered out if the GPU pool they

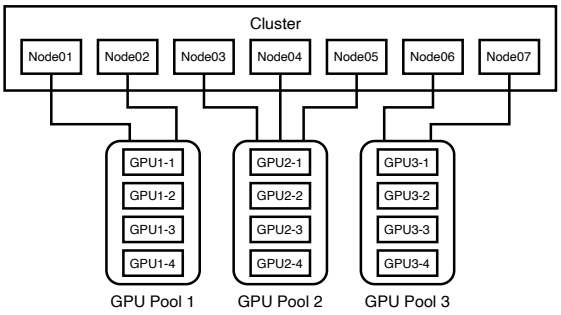


Fig. 6: The composable infrastructure for multi-pool. GPUs inside the GPU Pool 1 can be reallocated to either Node01 or Node02, but not the others.

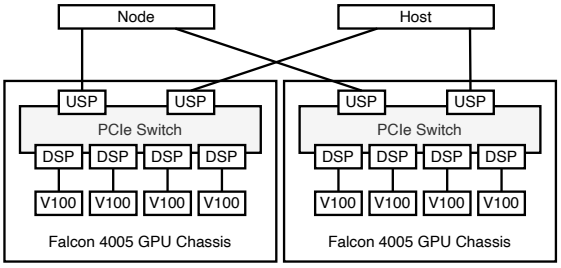


Fig. 7: The topology of our KubeComp implementation.

are connected to lacks sufficient available GPUs. The nodes that remain after filtering are scored by Equation 1, which is the same scoring function used in the single pool scenario. After the target node is selected, the GPU pool is fixed. Therefore, the rest of the reconfiguration process is identical to the single pool condition.

VI. Experimental Evaluation

A. Experiment Setup

1) Testbed: Fig. 7 is the topology of our testbed. It comprises two nodes equipped with AMD EPYC 7642 48-core processors. Each node has 2 sockets, 48 cores per socket, and 2 threads per core, forming 192 logical CPU cores in total. The nodes are connected to the H3 Platform Falcon 4005 PCIe 4.0 GPU chassis, which can support multi-host connections and enable GPU dynamic allocation among nodes. 8 NVIDIA Tesla V100 GPUs are inserted in the two chassis sets, forming a single 8-GPU pool shared by two nodes.

The nodes run on Ubuntu 20.04.6 LTS with Linux kernel version 5.4, which supports in PCIe device hotplug. The NVIDIA Driver version 520.61.05 is installed, allowing GPU hotplug when persistence mode is disabled. Key components of KubeComp are developed in Go. For reconfiguration, we implemented a disaggregated resource management interface layer as shown in Fig. 4 to invoke the restful APIs exposed by the Falcon 4005 chassis.

2) Simulator: The simulator models the cluster comprising homogeneous nodes equipped with GPUs, assuming uniform job performance across all nodes. Given the node configuration, which includes each node's GPU allocation and the GPU pool that the node connects to,

	Request CPU	Request GPU(w/ 80 SM)	CPU/GPU
Job A	80	1	1
Job B	40	1	1/2
Job C	40	2	1/4
Job D	40	4	1/8
Job E	40	8	1/16

TABLE I: Details of jobs. CPU/GPU is the ratio between CPU cores and total GPU SMs.

	Job A	Job B	Job C	Job D	Job E
Workload 1	50%	50%	-	-	-
Workload 2	50%	-	50%	-	-
Workload 3	25%	25%	25%	25%	-
Workload 4	24%	24%	24%	24%	4%

TABLE II: Details of workloads. Workloads are composed with jobs in Table I.

the simulator can emulate either the configurable cluster with composability or the preconfigured one without composability. The simulator is trace-driven. Given the input traces, including job ID, requested CPU and GPU, job duration, and the intervals between successive jobs, jobs are executed in a First-Come-First-Serve order.

3) Workloads: The workloads consist of real cluster traces and manipulated ones with mixtures of jobs in Table II. We adopt SenseTime Helios traces [18], using the data of the GPU cluster Earth from 2020 June to 2020 August. The manipulated workloads are composed of jobs in Table I with different proportions. Each job runs for five minutes and has diverse requests for CPU and GPU.

B. Wait Time Reduction

The experiment on our testbed demonstrates that KubeComp's resource-centric design reduces job wait times by dynamically reconfiguring the cluster. We executed Workloads 1-4 under three different settings: sharing GPUs among nodes (KubeComp), concentrating all GPUs on one node, and evenly distributing GPUs.

The result is shown in Fig. 8, and sharing GPUs among nodes (KubeComp) brings the shortest wait time. Workload 1 solely consists of single GPU jobs, so equally distributing GPUs yields similar performance to KubeComp. Workload 2 and Workload 3 request less than four GPUs and can run on all three settings, in which KubeComp presents the least wait time. Workload 4 contains a diverse range of jobs, including those requesting 8 GPUs. Though they account for merely 4%, the cluster that only contains nodes with 4 GPUs fails to complete these tasks. While concentrating GPU on the same node performs comparably to KubeComp, it still worsens by 25%. Although concentrating GPUs on a single node may suit high GPU demand jobs, it proves unsuitable for single GPU jobs requiring high CPU demand. KubeComp has the strength to handle any job distribution well. The result shows that the job wait time is possibly reduced by up to 89%.

C. GPU Utilization

The experiment on our testbed shows that KubeComp boosts GPU utilization by reducing GPU fragments

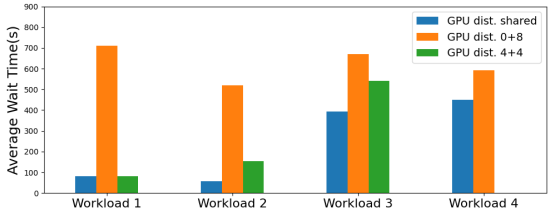


Fig. 8: Average wait time under different cluster settings of GPU distribution and compositions of the workloads.

TABLE III: The correlation coefficient of the duration of each phase and the reallocated GPUs.

	R
Node Selection	0.0021
GPU Availability Update	0.0010
GPU Allocation Retrieval	0.0357
GPU Reallocation	0.992
Scheduler Detection & Binding	0.0815
Total	0.970

through reconfiguration. We ran Workload 1-4 with three kinds of GPU distributions, the same as in the previous section. Fig. 9 presents the cumulative distribution of used GPU. KubeComp typically has a steeper curve as the used GPU increases, indicating efficient GPU utilization. In contrast, the clusters with fixed GPU allocation initially display steeper curves, implying frequent underutilization of GPUs.

D. Overhead

We measured the KubeComp overhead regarding CPU usage and reconfiguration time on our testbed. To assess CPU overhead, we monitored CPU usage every 30 seconds via Prometheus. The CPU usage for DisagGPU-DevMgr, Reconfig-Mgr, and KubeComp-sched was found to be less than 1.7×10^{-3}, 9×10^{-4} and 1.4×10^{-2} respectively, which is considered negligible. To analyze the time overhead, we measured the duration of each reconfiguration phase. Table III shows the correlation coefficient (R) between the duration and the reallocated GPUs. Among the five phases, only GPU reallocation correlates highly with the reallocated GPUs, aligning with our inference in Section V-B. Additionally, the total time of reconfiguration is associated with the reallocated GPU because GPU reconfiguration takes the majority of the time. This result emphasizes the importance of node selection.

E. Simulation

Fig. 10 shows the average job completion time of Workload 1-4 on the simulator. The variance between our physical machine and the simulator is generally less than 5%, and both show GPU pooling brings a shorter job completion time. The variance is likely caused by the simplified scheduler of the simulator, which does not perfectly mimic the behavior of Kubernetes. For example, the kube-scheduler employs mechanisms to prevent continuous retries for unscheduled pods, while our simulator lets pods be scheduled immediately once the required resources become available.

1) Single Pool Simulation: The simulator emulates a four-node cluster with a GPU pool of size 24, which is aligned with the current design of GPU chassis accommodating 20 to 30 GPUs at most. We ran the mix of Job A (single GPU job) and Job E (eight-GPU job) on the simulator to demonstrate the trend in wait time. In Fig. 11, KubeComp with shared GPU consistently has the shortest wait time. When Job A is dominant, spreading GPUs among nodes performs similarly to KubeComp, as Job A requires only a single GPU and can be executed as long as sufficient CPU resources are available. However, in Job E-dominant scenarios, KubeComp and fixed allocations with eight GPUs per node outperform because nodes with four GPUs cannot run Job E. The experiment highlights two key points: First, prior knowledge of the workload is essential to preconfigured clusters. Second, In unpredictable workloads, similar to real-world conditions, composable clusters provide excellent flexibility for just-in-time resource reconfiguration.

2) Multi-Pool Simulation: Multiple GPU chassis are required to accommodate hundreds of GPUs in real data centers. To verify KubeComp's real-world benefits, we ran SenseTime Helios traces on the simulator, emulating the multi-pool scenario. Each GPU pool in our simulator consists of 24 GPUs connected to four nodes. The configurable cluster allows flexible allocation within the pool, while the preconfigured cluster has a fixed allocation: 2 nodes with 4 GPUs each and 2 nodes with 8 GPUs each. Fig. 12 shows the wait time normalized to that of the preconfigured cluster with a single GPU pool. The configurable cluster reduces wait time by approximately 30% compared to the preconfigured setup due to efficient resource reallocation.

F. Resource Management

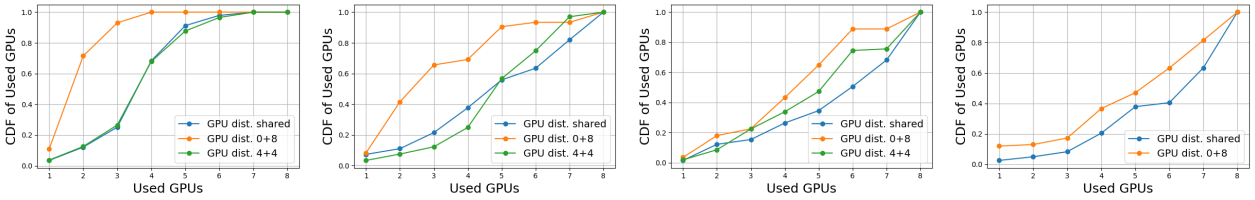
KubeComp applies several policies to select nodes for pods, aiming to reduce the number of reallocated GPUs and thus lower reconfiguration overhead. The experiment was conducted on the simulator to demonstrate these benefits.

The simulator contains five nodes, each with 6 GPUs attached, forming a GPU pool with 30 GPUs. There are 50 input jobs with an equal distribution of Job A-E. Job intervals are modeled by the Poisson distribution, with averages of 30, 60, and 120. Fig. 13 shows the result. Best-Fit and Least-Demand decrease the reallocated GPU in all cases and applying both policies brings the most significant decrease.

VII. Related Works

A. Resource Pool

Resource pools are the foundation of the composable system, allowing resource sharing across hosts. rCUDA [8], a user-mode library, allows user programs to access remote GPUs, reducing the number of GPUs in the cluster and saving energy. Device lending [21] and



(a) Workload 1. (b) Workload 2. (c) Workload 3. (d) Workload 4.

Fig. 9: The cumulative distributions of used GPUs under different cluster settings and workloads.

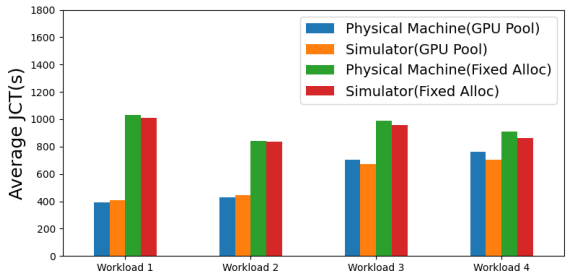


Fig. 10: Average job completion time on our physical machine and the simulator.

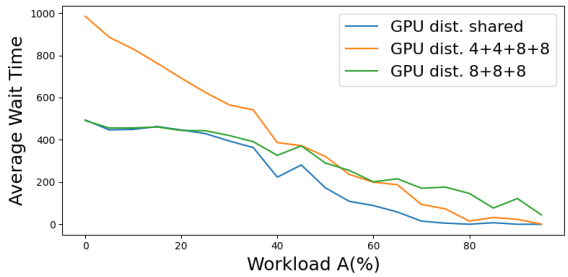


Fig. 11: Average wait time under different cluster settings of GPU distribution and compositions of Job A and E.

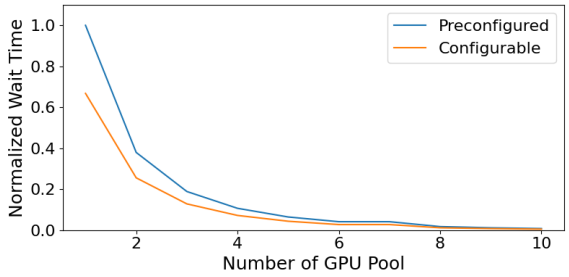


Fig. 12: Normalized wait time of the preconfigured and configurable cluster under different cluster scales.

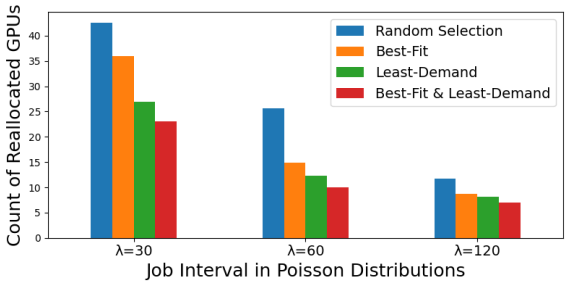


Fig. 13: Count of reallocated GPUs under different job intervals and when different policies are applied.

SmartIO [28] are the resource pool based on PCIe non-transparent bridges (NTB). PCIe devices, such as network cards and GPUs, can be shared among computers. DxPU [16] also creates GPU pool based on PCIe but overcomes the PCIe scale limitations with `DxPU_PROXY`, converting PCIe for transmission over network fabric to expand the scale of the resource pool. These techniques are essential for composable infrastructure and are promising to integrate with workload orchestrators.

B. Disaggregated GPU Solutions

DGSF [9], DRMaestro [1], and rCUDA integration with Slurm [20] allows processes to utilize disaggregated GPUs by API remoting. DGSF, designed for serverless functions, directs requests to available GPUs based on current GPU utilization. DRMaestr facilitates GPU disaggregation on Kubernetes using HFCUDA, optimizing workload placement to enhance resource efficiency and avoid interference. The rCUDA integration with Slurm allows clients to request the ”rgpu” device to access any GPU node using rCUDA. However, these solutions present disaggregated GPUs as virtual GPUs with limited functionality rather than assigning them as bare metal.

COMPaaS DLV [3], [4], [26], GigaIO’s Fabrex integration with Slurm [12] and OpenShift composability resource operator [33] enable containers to access bare-metal GPUs in a composable infrastructure. Nevertheless, these orchestrators are vendor-specific, and their publications do not discuss the reconfiguration policies. COMPaaS is used in academia, accessed via the instance of JupyterLab, but is tailored for Liqid fabric. Besides, when the hardware is reconfigured, Kubernetes deployments require a restart. GigaIO’s FabreX with Slurm dynamically composes nodes to accommodate workload requests. However, the node needs a reboot, preventing seamless deployment. As for OpenShift composability resource operator, it relies on the composable infrastructure facilitated by Liqid and does not handle on-demand resource provision, potentially resulting in resource fragmentation if manual operations are delayed. In contrast, KubeComp offers greater compatibility and achieves on-demand resource provision, reconfiguration optimization, and deployment automation, surpassing these disaggregated GPU solutions.

VIII. Conclusions

In our work, we developed KubeComp, the first solution for integrating composable infrastructure with the container orchestration system with on-demand resource provision, full deployment automation, and optimized resource management. KubeComp serves as a framework adaptable to various disaggregated resources and infrastructure vendors, and we choose GPU as the initial case. We successfully had Kubernetes support disaggregated GPU pooling, with experiments showing improvements in resource utilization and job wait times by up to 89%. KubeComp is open-source and publicly available.

References

[1] Marcelo Amaral, Jordà Polo, David Carrera, Nelson Gonzalez, Chih-Chieh Yang, Alessandro Morari, Bruce D'Amora, Alaa Youssef, and Malgorzata Steinder. Drmaestro: orchestrating disaggregated resources on virtualized data-centers. *Journal of Cloud Computing*, 10(1), 2021.

[2] Maciej Bielski, Ilias Syrigos, Kostas Katrinis, Dimitris Syrivelis, Andrea Reale, Dimitris Theodoropoulos, Nikolaos Alachiotis, Dionisios Pnevmatikatos, EH Pap, and George Zervas. dredbox: Materializing a full-stack rack-scale system prototype of a next-generation disaggregated datacenter. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1093–1098. IEEE, 2018.

[3] Maxine Brown, Luc Renambot, Lance Long, Timothy Bargo, and Andrew E Johnson. Compaas dlv: Composable infrastructure for deep learning in an academic research environment. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)*, pages 1–2. IEEE, 2019.

[4] Zhongyi Chen, Luc Renambot, Lance Long, Maxine Brown, and Andrew E. Johnson. Moving from composable to programmable, 2022.

[5] I-Hsin Chung, Bulent Abali, and Paul Crumley. Towards a composable computer system. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, pages 137–147, 2018.

[6] Cisco. Unified computing system (ucs). [Online]. Available: https://www.cisco.com/site/us/en/products/computing/servers-unified-computing-systems/index.html, 2023.

[7] Dell. Poweredge mx7000 modular chassis. [Online]. Available: https://www.dell.com/en-us/shop/ipovw/poweredge-mx7000, 2023.

[8] José Duato, Antonio J Pena, Federico Silla, Rafael Mayo, and Enrique S Quintana-Ortí. rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In *2010 International Conference on High Performance Computing & Simulation*, pages 224–231. IEEE, 2010.

[9] Henrique Fingler, Zhiting Zhu, Esther Yoon, Zhipeng Jia, Emmett Witchel, and Christopher J Rossbach. Dgsf: Disaggregated gpus for serverless functions. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 739–750. IEEE, 2022.

[10] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation, 2016.

[11] Renaud Gaubert and Jiaying Zhang. Device manager proposal. [Online]. Available: https://github.com/kubernetes/design-proposals-archive/blob/main/resource-management/device-plugin.md, 2021.

[12] GigaIO. Slurm workload manager: Efficient cluster management - gigaio. [Online]. Available: https://gigaio.com/slurm/, 2024.

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

[14] Anubhav Guleria, J Lakshmi, and Chakri Padala. Emf: Disaggregated gpus in datacenters for efficiency, modularity and flexibility. In *2019 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, pages 1–8. IEEE, 2019. "GPU disaggregation [39] technique has been proposed".

[15] Anubhav Guleria, J Lakshmi, and Chakri Padala. Quadd: Quantifying accelerator disaggregated datacenter efficiency. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 349–357. IEEE, 2019.

[16] Bowen He, Xiao Zheng, Yuan Chen, Weinan Li, Yajin Zhou, Xin Long, Pengcheng Zhang, Xiaowei Lu, Linquan Jiang, and Qiang Liu. Dxpu: Large scale disaggregated gpu pools in the datacenter. *ACM Transactions on Architecture and Code Optimization*, 2023.

[17] HPE. Hpe synergy blades composable infrastructure platform. [Online]. Available: https://www.hpe.com/us/en/integrated-systems/synergy.html, 2023.

[18] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. Characterization and prediction of deep learning workloads in large-scale gpu datacenters. In *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2021.

[19] Intel. Intel® rack scale design (intel® rsd) architecture. Report, Intel, 2018.

[20] Sergio Iserte, Adrian Castello, Rafael Mayo, Enrique S. Quintana-Orti, Federico Silla, Jose Duato, Carlos Reano, and Javier Prades. Slurm support for remote gpu virtualization: Implementation and performance study. In *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*. IEEE, 2014.

[21] Lars Bjørlykke Kristiansen, Jonas Markussen, Håkon Kvale Stensland, Michael Riegler, Hugo Kohmann, Friedrich Seifert, Roy Nordstrøm, Carsten Griwodz, and Pål Halvorsen. Device lending in pci express networks. In *Proceedings of the 26th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 1–6, 2016.

[22] Kubernetes. Scheduling framework. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/.

[23] Kubernetes. Considerations for large clusters. [Online]. Available: https://kubernetes.io/docs/setup/best-practices/cluster-large/, 2024.

[24] Chung-Sheng Li, Hubertus Franke, Colin Parris, Bülent Abali, Mukil Kesavan, and Victor Chang. Composable architecture for rack scale big data computing. *Future Gener. Comput. Syst.*, 67:180–193, 2017.

[25] Liqid. Gpu on-demand. [Online]. Available: https://www.liqid.com/products/gpu-on-demand, 2023.

[26] Lance Long, Timothy Bargo, Luc Renambot, Maxine Brown, and Andrew E Johnson. Composable infrastructures for an academic research environment: Lessons learned. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1209–1214. IEEE, 2022.

[27] LSALAB. Kubecomp. [Online]. Available: https://github.com/NTHU-LSALAB/KubeComp, 2024.

[28] Jonas Markussen, Lars Bjørlykke Kristiansen, Pål Halvorsen, Halvor Kielland-Gyrud, Håkon Kvale Stensland, and Carsten Griwodz. Smartio: Zero-overhead device sharing through pcie networking. *ACM Transactions on Computer Systems (TOCS)*, 38(1-2):1–78, 2021.

[29] David Mayhew and Venkata Krishnan. Pci express and advanced switching: Evolutionary path to building next generation interconnects. In *11th Symposium on High Performance Interconnects, 2003. Proceedings.*, pages 21–29. IEEE, 2003.

[30] NVIDIA. Nvidia gpu device plugin for kubernetes. [Online]. Available: https://github.com/NVIDIA/k8s-device-plugin/.

[31] NVIDIA. Specialized configurations with docker. [Online]. Available: https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/docker-specialized.html?fbclid=IwAR3v0Q8NpqLzbSR1_o6PLJB-9BP5U9zPznvqPwmTeWyWrbh0d6DynEiLZbA#, 2023.

[32] Minoru Oikawa, Atsushi Kawai, Kentaro Nomura, Kenji Yasuoka, Kazuyuki Yoshikawa, and Tetsu Narumi. Ds-cuda: a middleware to use many gpus in the cloud environment. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1207–1214. IEEE, 2012.

[33] Christian Pinto, Michele Gazzetti, and Michael Johnston. Composable systems with openshift. [Online]. Available: https://research.ibm.com/blog/composable-systems-openshift, 2023.

[34] H3 Platform. Composable gpu expansion chassis pcie gen4 switch box. [Online]. Available: https://www.h3platform.com/product-list/10, 2019.

[35] H3 Platform. Composable gpu system- alternating gpu resources to fulfill both ai training and inference requirements. [Online]. Available: https://www.h3platform.com/blog-detail/27, 2022.