Transparent GPU Sharing in Container Clouds for Deep Learning Workloads

Bingyang Wu and Zili Zhang, *Peking University;* Zhihao Bai, *Johns Hopkins University;* Xuanzhe Liu and Xin Jin, *Peking University*

https://www.usenix.org/conference/nsdi23/presentation/wu

This paper is included in the Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation.

April 17–19, 2023 • Boston, MA, USA

978-1-939133-33-5

Transparent GPU Sharing in Container Clouds for Deep Learning Workloads

Bingyang Wu* Zili Zhang* Zhihao Bai† Xuanzhe Liu* Xin Jin*

*Peking University †Johns Hopkins University

Abstract

Containers are widely used for resource management in datacenters. A common practice to support deep learning (DL) training in container clouds is to statically bind GPUs to containers in entirety. Due to the diverse resource demands of DL jobs in production, a significant number of GPUs are underutilized. As a result, GPU clusters have low GPU utilization, which leads to a long job completion time because of queueing.

We present TGS (Transparent GPU Sharing), a system that provides transparent GPU sharing to DL training in container clouds. In stark contrast to recent application-layer solutions for GPU sharing, TGS operates at the OS layer beneath containers. Transparency allows users to use any software to develop models and run jobs in their containers. TGS leverages adaptive rate control and transparent unified memory to simultaneously achieve high GPU utilization and performance isolation. It ensures that production jobs are not greatly affected by opportunistic jobs on shared GPUs. We have built TGS and integrated it with Docker and Kubernetes. Experiments show that (i) TGS has little impact on the throughput of production jobs; (ii) TGS provides similar throughput for opportunistic jobs as the state-of-the-art application-layer solution AntMan, and improves their throughput by up to 15× compared to the existing OS-layer solution MPS.

1 Introduction

Containers [1–3] are widely used for resource management in datacenters. Containers provide lightweight virtualization, and can significantly reduce the complexity and cost of deployments and managements in datacenters.

Deep learning (DL) is an important workload in datacenters. With recent advancements in deep neural networks (DNNs) [4] and the burst of big data space, DL models have been increasingly integrated into applications and online services. Large enterprises build multi-tenant GPU clusters that are shared by many teams to develop and train DL models.

A common practice to support DL training in container clouds is to statically bind complete GPUs to containers. When a GPU is allocated to a container, the container has exclusive access to the GPU, which provides performance isolation for production jobs. But it means that other containers

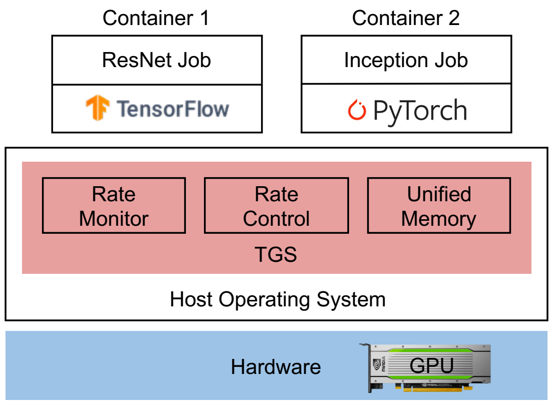


Figure 1: TGS architecture.

on the same machine cannot use the GPU when the GPU is under-utilized or is even completely idle.

The major limitation of this approach is *low resource utilization*. A recent study on a production GPU cluster by Microsoft shows that the mean GPU utilization is only 52% [5]. Another measurement on a production GPU cluster at Alibaba shows even lower GPU utilization—the median GPU utilization is no more than 10% [6]. However, due to exclusive GPU allocation, incoming jobs have to wait in the queue to be scheduled even when many GPUs are not fully utilized. This causes a long job completion time for subsequent jobs.

This is a known problem in production GPU clusters [5, 6]. The problem can be addressed by GPU sharing to increase GPU utilization. In production environments [6–8], DNN training jobs are typically classified into two classes: *production* jobs, which must run without much great performance degradation caused by other jobs, and *opportunistic jobs*, which utilize spare resources. It is natural to share GPUs between the two classes of jobs to improve GPU utilization. Yet, it is critical for production environments to ensure that the impact of GPU sharing on production jobs is minimized.

GPU sharing solutions can be realized at either the application layer or the OS layer. AntMan [6] is a state-of-the-art application-layer solution. While AntMan can provide high GPU utilization and performance isolation, it modifies DL frameworks non-trivially and restricts users to use particular versions of given frameworks. NVIDIA Multiple Process

Sharing (MPS) [9] is an OS-layer solution. MPS requires application knowledge to set resource limits for performance isolation and does not support GPU sharing under GPU memory oversubscription. It merges several processes into a single CUDA context, leading to fate sharing between jobs.

We present TGS, a system that provides transparent GPU sharing to DL training in container clouds. Unlike application-layer solutions, TGS works at the OS layer and realizes the benefits of application-layer solutions at the OS layer without the limitations of existing OS-layer solutions. Transparency allows users to choose any version of any DL framework (either TensorFlow, PyTorch or a custom framework) to develop models and run jobs in containers.

The core of TGS is a lightweight indirection layer between containers and GPUs. It intercepts the system calls from containers to GPUs and regulates the GPU resource usage for concurrent jobs. TGS enables GPU sharing between the production job and the opportunistic job, but largely isolates the production job from contention.

There are two primary technical challenges in realizing an OS-layer GPU sharing solution with performance isolation. The first challenge is to share GPU compute resources between containers adaptively without application knowledge. Inaccurately setting resource limits for each container would either degrade job performance or leave resources unused. MPS and MIG require application knowledge to manually set resource limits. TGS applies an adaptive rate control approach to address this challenge without application knowledge. It monitors the performance of production jobs at runtime, and adaptively updates the resource allocation to opportunistic jobs. The control loop automatically converges to the point that opportunistic jobs utilize as many resources as possible without much affecting production jobs.

The second challenge is to enable transparent GPU memory oversubscription. GPUs have their own memory to keep the application state. MPS fails when the total GPU memory required by containers exceeds the GPU memory size. AntMan uses a custom memory management component in DL frameworks to manage memory swapping between GPU memory and host memory at the application layer. We design a transparent unified memory mechanism based on CUDA unified memory to enable unified memory at the OS layer, obviating the need to explicitly modify applications. This mechanism manages memory swapping underneath when the GPU memory is oversubscribed. TGS leverages placement preferences to ensure that GPU memory is prioritized for production jobs to protect their performance.

In summary, we make the following contributions.

- We propose TGS, a system that provides transparent GPU sharing for DL training in container clouds.
- We design adaptive rate control and transparent unified memory mechanisms to simultaneously achieve high GPU utilization and performance isolation.
- We implement TGS and integrate it with Docker and Kubernetes. Experiments show that (i) TGS has little impact on the throughput of production jobs; (ii) TGS provides similar throughput for opportunistic jobs as state-of-the-art application-layer solution AntMan and improves their throughput by up to 15× compared to existing OS-layer solution MPS.

2 Background and Motivation

In this section, we first introduce containers, deep learning training, and the current practice to support deep learning training in container clouds. Then, we show the limitations of existing solutions to motivate TGS.

2.1 Container Clouds

Containers [1–3] (e.g., Docker) are used widely to manage resources and deploy workloads in datacenters, and provide *portability* and *isolation*. A container is a standalone software package including everything needed to run an application. A containerized application can run across various environments without any modifications. Such portability enables developers to use the tools and application stacks of their choice to develop and run their applications, without worrying about deployment environments. Applications in different containers are isolated by using independent namespaces.

Containers are lightweight, compared with virtual machines. Virtual machines use a guest OS, but containers use the host OS kernel. Thus, applications can achieve bare metal performance when running in containers. Cloud operators use a container orchestration platform [10, 11] to provision, manage and update containers on many machines in a datacenter.

2.2 DL Training Workloads

DL training uses a dataset to train a DNN model. A training job contains many iterations. Each iteration uses a batch of samples from the dataset to train the DNN model. An iteration includes a forward pass and a backward pass. The forward pass uses the DNN model to compute the labels of the samples in the batch. A loss is computed based on the output labels and the actual labels using a loss function. The backward pass propagates the loss from the last layer to the first layer of the DNN model and computes the gradients for each weight. The DNN model is updated based on the gradients using an optimizer. DL training is compute-intensive, so GPUs are typically used. However, widely-adopted exclusive GPU allocation leads to low GPU utilization in production, as reported by Microsoft [5] and Alibaba [6].

2.3 Limitations of Existing Solutions

A natural way to increase GPU utilization is GPU sharing. If a single container cannot utilize all the GPU resources, a GPU can be shared by multiple containers to increase GPU utilization. However, containers on a shared GPU will compete for compute and memory resources of the GPU, and the interference can slow down the jobs.

	AntMan [6]	Salus [12]	PipeSwitch [13]	MPS [9]	MIG [14]	TGS
Transparency				✓	✓	✓
High GPU utilization	✓	✓				✓
Performance isolation	✓	✓	✓	✓	✓	✓
Fault isolation	✓		✓		✓	✓

Table 1: Comparison between TGS and existing GPU sharing solutions.

GPU sharing can be done either at the application layer or the OS layer. The primary drawback of application-layer solutions [6, 12, 13] is that they are not *transparent* to users, i.e., they require significant modifications to DL frameworks. Users are restricted to use the set of supported versions of given frameworks and have to wait for the integration if a newer version of a particular DL framework comes. This approach loses the advantage of allowing users to use any tools to develop and run applications in containers.

NVIDIA MPS [9] is an OS-layer solution for GPU sharing. It requires application knowledge to properly set the resource limit for each process to ensure performance isolation. More importantly, MPS requires the total GPU memory of the processes to fit within the GPU memory capacity and relies on applications to handle memory swapping between GPU memory and host memory. Another limitation of MPS is that it does not provide *fault isolation*. MPS merges the CUDA contexts of multiple processes into a single CUDA context to share the GPU. When a process fails, it leaves the MPS server and other processes in an undefined state and may result in process hangs, corruptions, or failures.

NVIDIA Multi-Instance GPU (MIG) [14, 15] is another OS-layer solution. MIG requires GPU hardware support and is currently only available on three high-end GPUs, i.e., NVIDIA A100, NVIDIA A30, and NVIDIA H100. MIG cannot *arbitrarily* partition a GPU based on application needs; it only supports GPU partitioning for a given set of configurations. For example, an NVIDIA A100 GPU can be partitioned into *GPU instances* with separate compute and memory resources for different DL training jobs, but MIG only provides seven fixed configurations for each GPU instance and each GPU instance cannot use more than 4/7 of the GPU compute resources or half of the GPU memory resources. Furthermore, it cannot *dynamically* change GPU resources owned by GPU instances if there are running jobs on the GPU even if the GPU usage of a container changes. Reconfiguration of MIG can only happen when the GPU is idle. MIG does not support memory oversubscription.

3 TGS Overview

TGS is a GPU sharing system for deep learning training in container clouds that is designed to meet the following goals. Table 1 compares TGS with existing GPU sharing solutions regarding these four goals.

- **Transparency.** The system should be transparent to applications so that users can use any software to develop and train DNN models in containers.
- **High GPU utilization.** The system should achieve high GPU utilization for both compute and memory resources.
- **Performance isolation.** The system should provide performance isolation for DL jobs. Production jobs should not be significantly affected by opportunistic jobs.
- **Fault isolation.** Application faults should be isolated by containers. The fault of an application in one container should not crash applications in other containers.

Architecture. Figure 1 shows that TGS is an OS-layer approach: it sits between containers and GPUs. Containers and applications are unaware of TGS. Users can use any custom framework to develop and train DNN models. A GPU is exposed as a regular GPU to the containers. The processes in the containers issue GPU kernels, i.e. functions executed on the GPU, to the GPU as they do with a dedicated GPU. TGS uses a lightweight indirection layer to share the GPU between workloads of several containers. The indirection layer intercepts the GPU kernels from containers and regulates these GPU kernels to control the resource usage of each container.

Key ideas. TGS leverages an adaptive rate control mechanism and a transparent unified memory mechanism to tackle two challenges in providing transparent GPU sharing at OS layer. The first challenge is to adaptively share GPU compute resources between containers without application knowledge. To address this challenge, the rate monitor of TGS monitors the performance of each container, and provides the number of *CUDA blocks* (a basic scheduling and execution unit on the GPU) as a real-time signal for the control loop. Based on the signal, the rate control of TGS adaptively controls the rate of sending GPU kernels to the GPU for each container. The control loop automatically converges to the point that opportunistic jobs utilize as many remaining resources as possible to achieve high GPU utilization without greatly affecting the performance of production jobs.

The second challenge is to enable transparent GPU memory oversubscription. AntMan [6] modifies DL frameworks to swap GPU memory when GPU memory is oversubscribed. OS-layer solution MPS does not support GPU memory oversubscription, and relies on applications to handle memory swapping. These approaches are not transparent. To address this challenge, TGS exploits CUDA unified memory [16] which unifies GPU memory and host memory in a single memory space. TGS intercepts and redirects GPU memory allocation calls from containers to the CUDA unified memory space. When the GPU memory is oversubscribed, TGS can automatically evict some data of opportunistic jobs to the host memory, and change the mapping of the corresponding virtual addresses to the new data locations in the host memory. The entire process is transparent to applications. To ensure performance isolation, TGS uses memory placement preferences to prioritize allocating GPU memory for production jobs over opportunistic jobs.

The design of TGS has two other benefits. First, the architecture is *lightweight*. TGS has low overhead and conforms with the principle of containers. Second, TGS provides the same *fault isolation* property as regular containers. The containers in TGS use separate GPU contexts, as opposed to MPS which merges the CUDA contexts of the containers into one. Therefore, an application fault in one container does not affect or terminate other containers.

4 TGS Design

In this section, we present the design of TGS. We first describe the adaptive rate control mechanism to share GPU compute resources. Then we describe the unified memory mechanism to share GPU memory resources.

4.1 Sharing GPU Compute Resources

Application code is encapsulated into functions to be executed on a GPU, which are known as GPU kernels. GPU kernels are highly optimized based on the particular architecture and execution model of the GPU. A small DNN training job may not use all the compute resources of a GPU. In this case, the GPU has low utilization if it is exclusively allocated to the container of the job. TGS improves GPU utilization by GPU sharing. In TGS, a GPU can be exposed to and shared by multiple containers to increase GPU utilization.

TGS ensures the performance of production jobs is not greatly affected by opportunistic jobs. Opportunistic jobs use no more than the resources left by production jobs. To achieve this, we need to solve two problems. First, we need to estimate how many resources are left by production jobs. Second, we need to control opportunistic jobs to use no more than the remaining resources.

Strawman solution: priority scheduling. A strawman solution is priority scheduling. It intercepts the GPU kernels from containers and puts them into a production queue and an opportunistic queue based on the priority of the job. The kernels in the opportunistic queue are only scheduled to the GPU when the production queue is empty. In this solution, whether there are remaining resources is estimated by checking whether the production queue is empty, and controlling the resource usage of opportunistic jobs is achieved by prioritizing the scheduling of the kernels in the production queue. This is a canonical solution to performance isolation and high utilization, and has been widely used in computer systems.

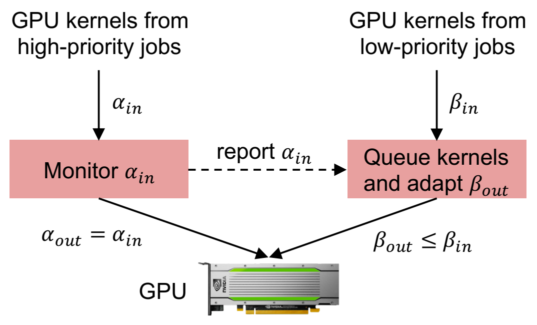

GPU kernels from high-priority jobs
GPU kernels from low-priority jobs
α_{in}
β_{in}
Monitor α_{in}
report α_{in}
Queue kernels and adapt β_{out}
$\alpha_{out} = \alpha_{in}$
$\beta_{out} \leq \beta_{in}$
GPU

Figure 2: Adaptive rate control.

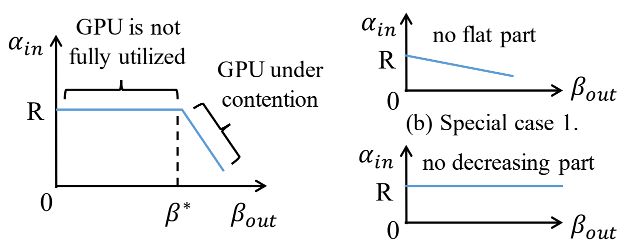
However, this solution is not suitable for GPU sharing. An empty production queue for GPU jobs does not mean production jobs are not using the GPU. A GPU kernel is an optimized GPU function that runs for some time. The GPU kernels scheduled in the past may still be running on the GPU, while the production queue is empty. Similarly, an empty queue also cannot tell how many resources on left on the GPU. Therefore, if the kernels in the opportunistic queue are sent to the GPU and the production jobs are using most of the GPU resources, then the GPU kernels from both jobs would contend with each other, which incurs large overhead for production jobs. Keeping track of GPU kernels running on the GPU is also not feasible, because the state of the GPU is not fully visible.

It may be possible to implement a priority scheduler into the GPU device driver, so that the scheduler can have full visibility of the resource usage and can perform fine-grained control. This solution is not general. It is tightly tied to the low-level GPU specifics and requires deep integration with each type of GPU based on their architecture and execution model. Some GPUs are blackboxes and do not expose such control to the OS.

Our solution: adaptive rate control. TGS uses an adaptive rate control approach (Figure 2). The main idea is to carefully control the dequeuing rate of the kernels in the opportunistic queue based on the kernel arrival rate, so that opportunistic jobs can use up the remaining compute resources without greatly affecting the production job. This is a general OS-layer approach: it is decoupled from low-level GPU specifics and does not require access to GPU internal control.

This approach requires a feedback signal to tell the control loop whether the dequeuing rate of the opportunistic queue can be increased to use more resources or should be decreased to avoid degrading production jobs. Ideally, we want to use the application performance, i.e., the training throughput for DL training workloads, as the feedback signal, because this is the metric we ultimately care about. However, we cannot directly obtain the training throughput, because this requires application knowledge, and we aim to design an OS-layer solution that is transparent to applications.

One choice of the signal is GPU utilization, i.e., increase the rate if the GPU utilization is below 100%. While this choice



(a) General case. (b) Special case 1. (c) Special case 2.

Figure 3: Relationship between the rates of production and opportunistic jobs.

seems natural, it has two drawbacks. First, the definition of GPU utilization is hardware-specific and is often vague [17]. Today's GPUs contain different types of compute units on a single chip, e.g., Tensor cores and CUDA cores for different data types on NVIDIA GPUs. GPU utilization reported by GPU drivers (if supported) often lacks a precise definition. Even if it does (e.g., the percentage of stream processors that are used), it is unclear what a single utilization value actually means for a GPU with several types of compute units. Second, GPU utilization is only loosely coupled with the application performance. Even when the reported GPU utilization is below 100%, it does not mean we can increase the dequeuing rate of the opportunistic queue without slowing down production jobs. For example, a production job and an opportunistic job may compete for the same type of compute units that are already used up by the production job alone, though there are other types of compute units that are idle; two jobs may also compete for other resources than the one captured by GPU utilization.

In TGS, we use the kernel arrival rate of production jobs (i.e., the rate that TGS receives kernels from the containers) as the feedback signal. A DL training job constructs a compute graph based on the DNN model for its training process. It uses the compute graph to generate and send kernels to the GPU to perform training. The compute graph captures the dependencies between the kernels. The kernel arrival rate directly corresponds to the training throughput. If the training is slowed down, the kernels are finished slower, the dependencies are satisfied slower, and the kernel arrival rate drops. Therefore, TGS uses a rate monitoring module to monitor the kernel arrival rate of production jobs, and uses it as the feedback signal to control the kernel dequeuing rate of opportunistic jobs. Note that any contention between production jobs and opportunistic jobs can be captured by this kernel arrival rate, including GPU cache contention, CPU contention and network contention. Some of them are beyond what a GPU hardware design can control, and TGS uses rate control as a knob to control all of them. Since there can be a small variance in the kernel arrival rate, TGS uses a moving average to smooth the estimation of the kernel arrival rate. For the kernels from production jobs, TGS only performs a simple counting operation to estimate the kernel arrival rate. It does not queue the kernels and directly passes them to the GPU, to minimize the impact on the performance of production jobs.

Rate adaptation algorithm. The rate adaptation algorithm controls the kernel dequeuing rate of the opportunistic queue, so that the kernel arrival rate of production jobs is not greatly affected and the kernel dequeuing rate of opportunistic jobs is maximized. Formally, let α_{in} and α_{out} be the rates that the kernels of production jobs arrive at and departure from TGS respectively, and β_{in} and β_{out} be those of the opportunistic jobs. TGS only monitors, but does not limit the rate of production jobs. So $\alpha_{in} = \alpha_{out}$. Let the kernel arrival rate of production jobs when the GPU is not shared be R. The rate control algorithm is to maximize β_{out} so that $\alpha_{in} = R$. In the formulation, β_{out} is the variable controlled by the algorithm and α_{in} is dependent on β_{out}. Let f be the function that captures the relationship between α_{in} and β_{out}, i.e., $\alpha_{in} = f(\beta_{out})$. Then the algorithm has to solve the following optimization problem.

$$\begin{aligned} max \quad & \beta_{out} && (1) \\ s.t. \quad & \alpha_{in} = f(\beta_{out}) \geq R && (2) \\ & \beta_{out} \geq 0 && (3) \end{aligned}$$

The exact shape of $f(\beta_{out})$ is unknown, but we know its rough shape by the nature of the problem. Specifically, $f(\beta_{out})$ is flat and is equal to R when β_{out} is small, and is monotonically decreasing when β_{out} is large, as illustrated in Figure 3(a). The intuition is that when β_{out} is small, the GPU is not fully utilized and executing the kernels of opportunistic jobs does not affect the performance of production jobs, resulting in a flat line; after the tipping point β^*, opportunistic jobs start to compete with production jobs for GPU resources, causing the performance of production jobs to drop. Note that the monotonically decreasing part is not necessarily linear; Figure 3(a) illustrates the general trend that α_{in} decreases when β_{out} increases. The goal of the algorithm is to find the tipping point β^* from which $f(\beta_{out})$ starts to decrease.

Figure 3(a) is the general case. There are two special cases. Figure 3(b) is the special case where the GPU is already fully utilized by production jobs, so that even executing a small number of kernels for opportunistic jobs would degrade the performance of production jobs. In this case, the line does not have a flat part. Figure 3(c) is the special case where the demand of opportunistic jobs is very small, so that even when the dequeuing rate is not limited, the performance of production jobs is not affected. In this case, the line does not have a monotonically decreasing part.

To approximate the optimal β_{out}, we use the canonical additive increase multiplicative decrease (AIMD) method to control the rate β_{out}, as shown in Algorithm 1. Specifically, TGS first measures the rate R of a production job on a GPU before it adds an opportunistic job to the GPU for sharing (line $1-3$). After the opportunistic job is added, TGS additively increases β_{out}, if α_{in} is greater than or equal to R (line $24-$

25), or multiplicatively decreases β_{out}, if α_{in} is below R (line $29-30$). AIMD ensures that β_{out} can approximately converge to the tipping point β^*. To accelerate convergence, a slow start phase is adopted (line $17-22$). Experiments in §6 shows that the convergence is fast. When the production job changes its resource usage pattern, TGS detects that the variance of R is beyond a threshold. In this case, the rate control module suspends the opportunistic job and measures new R (line $26-28$). When R becomes stable, the rate control module uses AIMD to adjust β_{out} to the tipping point. We have the following theorem to ensure the convergence of the adaptive rate control algorithm at most cases.

Theorem 1 *Assuming DL jobs are stable during the profiling phase and the convergence phase, the adaptive rate control algorithm converges in $O(B \log B)$ function calls, where B is the throughput limit of jobs in the GPU.*

The proof of the theorem is in Appendix A. The proof is based on the stability of the deep learning training workload. For readers familiar with congestion control in computer networking, our problem resembles the bandwidth allocation problem when multiple flows compete for the bandwidth resources of a shared link. In bandwidth allocation, each flow uses a congestion control algorithm to control its own rate, and after the system converges, each flow gets a fair share of the link bandwidth. Our problem is subtly different from bandwidth allocation in that we do not limit the rate of production jobs, and only control the rate of opportunistic jobs to ensure that the performance of production jobs is not greatly affected by resource sharing.

Algorithm 1 Adaptive Rate Control Algorithm

```
 1: procedure INIT
 2:     $R = measure\_high\_prio\_job\_rate()$
 3:     $\beta_{out} = 0$
 4:     $state = SLOW\_START$
 5:
 6: procedure UPDATE_HIGH_RATE
 7:     $R_{avg} = avg(high\_rate\_window)$
 8:     $dR = |R - R_{avg}|/R$
 9:     if $dR < R\_threshold$ then
10:         $R = max(R, R_{avg})$
11:     else
12:         $R = measure\_high\_prio\_job\_rate()$
13:
14: procedure UPDATE_LOW_RATE_LIMIT
15:     $d\alpha = |R - \alpha_{in}|/R$
16:     switch $state$ do
17:         case $SLOW\_START$ :
18:             if $d\alpha < threshold_{slow\_start}$ then
19:                 $\beta_{out} *= \delta_{SS}$
20:             else
21:                 $\beta_{out} /= \delta_{SS}$
22:                 $state = CA$
23:         case $CA$ :
24:             if $d\alpha < threshold_1$ then
25:                 $\beta_{out} += \delta_{AI}$
26:             else if $d\alpha > threshold_2$ then
27:                 $\beta_{out} = 0$
28:                 $state = SLOW\_START$
29:             else
30:                 $\beta_{out} *= \delta_{MD}$
```

4.2 Sharing GPU Memory Resources

GPUs have GPU memory that is separated from the host memory. The memory size in modern GPUs ranges from a few GB to tens of GB. GPU memory stores the state and data needed by applications to perform their computation on the GPU. The compute units in the GPU can access the GPU memory much faster than the host memory. The GPU device driver exposes the GPU memory to users with an API, which is similar to the memory management API for the host memory. Users use the API to allocate and manage GPU memory for their GPU programs, e.g., `cudaMalloc` for GPU memory allocation on NVIDIA GPUs. Similar to GPU compute resources, the GPU memory can be shared by multiple containers when a single container cannot utilize all the GPU memory resources.

Strawman solution: pass-through allocation. A strawman solution is to directly pass the GPU memory allocation calls from containers to the GPU. In this way, the GPU memory is fully utilized as long as there is enough demand from containers. The major limitation of this solution is that it has large overhead for production jobs. In this solution, when production jobs do not use all the GPU memory, opportunistic jobs can obtain the remaining memory. Later, if the production job wants to allocate more GPU memory, they would not be able to do so because the remaining memory has been allocated to opportunistic jobs. Without sufficient GPU memory, production jobs may run at a lower speed, or even fail, which violates fault isolation.

Another limitation of this solution is that it does not consider the characteristics of DL frameworks. When starting a job, some DL frameworks (e.g., TensorFlow) claim all the available GPU memory even if the training job does not request that much memory. These DL frameworks typically have a memory pool that caches all the allocated memory, and give the memory to the training job on demand. They do not free and return the allocated memory back to the GPU when some memory is not used. This is an optimization in these DL frameworks to avoid the overhead of frequently calling GPU memory to allocate and release during a job.

This optimization introduces challenges to sharing the GPU memory. Application-layer solutions like AntMan [6] can directly modify DL frameworks to obtain the memory usage of training jobs and disable unnecessary memory caching to return unused GPU memory back to the GPU. However, to design a transparent OS-layer solution, modifications on DL frameworks or applications are not allowed.

Our solution: unified GPU and host memory. Modern GPUs provide a feature called *unified memory* which unifies GPU memory and host memory in a single address space. Unified memory is traditionally used by applications to simplify GPU memory management. TGS applies CUDA unified memory [16] in a novel way: it uses CUDA unified memory allocation as an *indirection* of GPU memory allocation, in order to achieve transparency and performance isolation for GPU memory sharing. Specifically, TGS exposes CUDA unified memory as pseudo GPU memory to containers. When a container issues a GPU memory allocation call, whether the call is for regular GPU memory or CUDA unified memory, TGS intercepts this call and allocates the memory requested by the call in the CUDA unified memory space. When production jobs do not use up the GPU memory, opportunistic jobs can obtain the remaining GPU memory.

Pseudo GPU memory refers to that the allocated memory appears to be normal GPU memory to containers and applications, while it can actually come from either GPU memory or host memory depending on availability. Note that we do not change the virtual memory system. Pseudo memory is still virtual memory, and applications use virtual memory addresses to access allocated pseudo memory. A GPU/host virtual memory address is translated to a GPU/host physical memory address by the GPU/host memory management unit.

The transparent unified memory in TGS is different from the original CUDA unified memory in two aspects, which are (i) performance isolation and (ii) transparent oversubscription of GPU memory. To provide performance isolation, TGS uses placement preferences in CUDA unified memory to prioritize the allocation of GPU memory to production jobs. When the GPU memory is not full, the memory allocation requests from any job get the GPU memory. When the GPU memory is full, TGS tries to place the blocks of production jobs in the GPU memory, and evict the blocks of opportunistic jobs to the host if necessary. This is transparent to the containers, as the containers still use the same virtual memory addresses to access their allocated memory space. The virtual memory addresses are translated to physical memory addresses at different locations. This mechanism also does not introduce additional out-of-memory (OOM) faults, because in the view of DL training jobs, the GPU memory capacity is the same as the size of the original GPU memory.

The transparent unified memory in TGS also addresses the issue of overclaiming the GPU memory in existing DL frameworks, without modifications to DL frameworks. When the DL framework claims all the available GPU memory, TGS allocates the requested amount of memory from the CUDA unified memory space. The actually used memory would trigger GPU page faults and be swapped to the GPU memory when it is used for the first time, and then would reside in the GPU memory. Consequently, only the portion actively used by the training job is in the GPU memory; the remaining portion is in the host memory. This allows opportunistic jobs to efficiently share the GPU memory.

5 Implementation

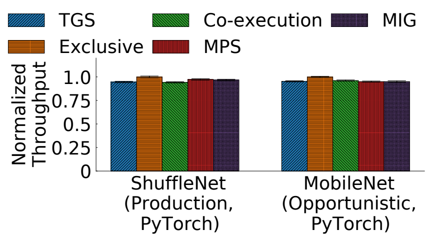
We have implemented a system prototype for TGS with $\sim$3000 lines of code in C++ and Python, and integrated it with Docker and Kubernetes. A coordinator process takes charge of resource management and leverages the indirection layer of TGS to enable GPU sharing between containers. Specifically, the adaptive rate control and the transparent unified memory provided by TGS are used for GPU sharing. The code of TGS is open-source and is publicly available at https://github.com/pkusys/TGS.

Adaptive rate control. TGS intercepts CUDA driver API calls related to CUDA kernel launch from containers for rate monitoring and rate control. Because CUDA kernel launch may be evoked by multiple threads in the container, TGS uses a global counter to record the number of CUDA blocks launched in a given time period. A CUDA block is a group of threads that must execute in the same SM (Streaming Multiprocessor) and different CUDA blocks can run independently in parallel. As the number of a CUDA block that a kernel contains is specified in the CUDA driver API call, the number of pending CUDA blocks can be treated as a real-time signal to estimate the performance of production jobs. For a production container, a standalone thread serves as the rate monitor, which reads this counter of the TGS periodically and sends the value to the rate-control component of the opportunistic container on the same GPU. For an opportunistic container, a rate control thread is created when the CUDA driver starts to work. The rate control thread adjusts the rate limit of the opportunistic container according to the received statistics. To keep the kernel launch rate of the opportunistic container at a desirable value, all CUDA kernel launch API calls are redirected to the rate control component first. The rate control component accesses statistics generated by the rate monitor to examine whether the rate limit is satisfied and defers the kernel launch if the rate of the opportunistic container exceeds the rate limit.

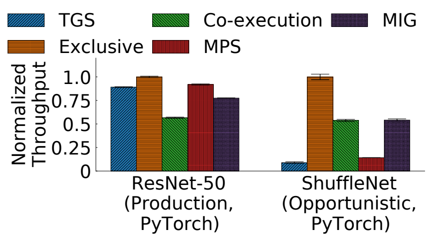
Unified memory management. To implement transparent memory sharing, TGS intercepts CUDA driver API calls related to GPU memory allocation, such as `cuMemAlloc`, and replaces these calls with unified memory allocation calls using `cuMemAllocManaged`. We use `cuMemAdvise` to prioritize the allocation of GPU memory for production containers. Specifically, we use `cuMemAdvise` to set the preferred location of memory allocation as the current GPU to avoid eviction for production containers. When the production container finishes, the indirection layer in the opportunistic container would use CUDA driver API `cuMemPrefetchAsync` to prefetch memory located in the host memory transparently.

6 Evaluation

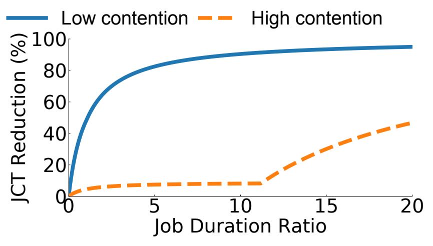
Setup. Most experiments are conducted on a server machine configured with an Intel Xeon Silver 4210R CPU, two NVIDIA A100 40 GB PCIe GPUs and 126 GB host memory. AntMan [6] only open-sourced one particular version based on TensorFlow 1.15.4 and the version is not compatible



(a) Low-contention scenario.

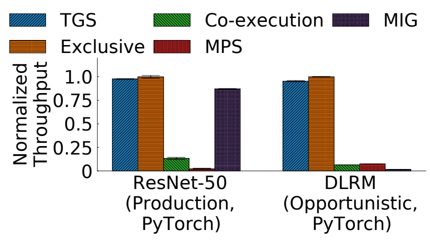


(b) High-contention scenario.

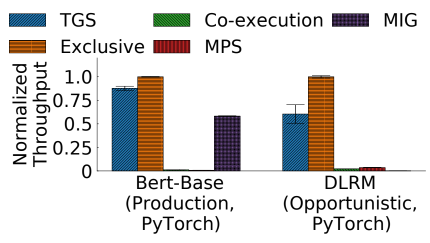


(c) JCT reduction of TGS over Exclusive.

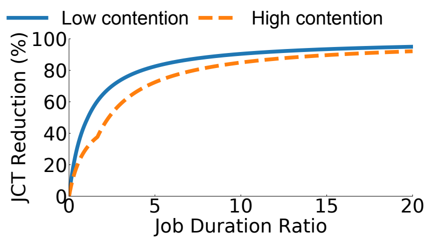
Figure 4: Throughput of production and opportunistic jobs for different model pairs when GPU memory is sufficient.



(a) Low-contention scenario.



(b) High-contention scenario.



(c) JCT reduction of TGS over Exclusive.

Figure 5: Throughput of production and opportunistic jobs for different model pairs under GPU memory oversubscription.

with A100. Therefore, all experiments involved in TensorFlow are conducted on an AWS p3.2xlarge instance which is configured with eight Intel Xeon Scalable (Skylake) vCPUs, one NVIDIA V100 16 GB Tensor Core GPU and 61 GB host memory. The software environment includes NVIDIA driver 460.91.03, CUDA 11.2, Docker 20.10.5, PyTorch 1.9.0, TensorFlow 1.15.4, torchvision 0.10.0 and scipy 1.6.3.

Workloads. We use various models for evaluation. The models include ShuffleNet, MobileNet, GCN (Graph Convolutional Network), ResNet-50, BERT-Base, DLRM (Deep Learning Recommendation Model) and ESPnet2. These models are representative and widely-used, and are standard benchmarks for evaluating DL systems. They vary in terms of GPU resource usage, which allows us to evaluate TGS under different levels of GPU resource contention.

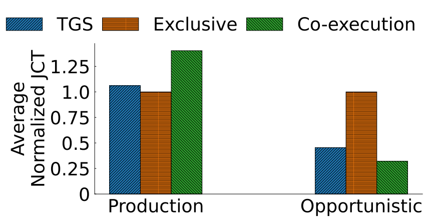
Comparison. To demonstrate the benefits of TGS, we compare the following mechanisms in the experiments. Each job runs in a separate container. We use throughput (iterations per second) as the main metric to evaluate the performance of different mechanisms, because it is a direct metric of a job's speed. We run at least 100 seconds for each case to measure the variance of the throughput, which typically includes 2000 iterations of a DL training job. Because a DL training job performs the same computation for each iteration (only the input data is different), the variance is low. We also use job completion time (JCT), but it depends both on the throughput and the number of iterations. The latter is configured by the user and varies from job to job.

- **TGS.** This is the proposed system.
- **Exclusive.** The production and opportunistic jobs are given exclusive access to a GPU when they run.
- **Co-execution.** The production job and the opportunistic job are executed concurrently without TGS.
- **NVIDIA MPS.** The production job and the opportunistic job run concurrently with NVIDIA MPS. We manually find the appropriate resource limit to set for each job in MPS to ensure that the performance of the production job is not affected by the opportunistic job.
- **NVIDIA MIG.** We manually set the best configuration to partition GPUs into different GPU instances so that the performance degradation of the production job brought by the opportunistic job is minimal.

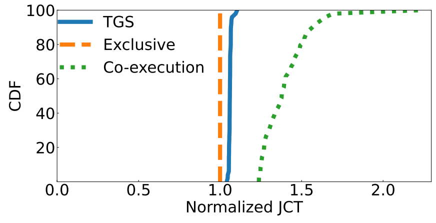
Due to the compatibility issue of AntMan [6], we compare it with TGS in §6.7.

6.1 Adaptive Rate Control

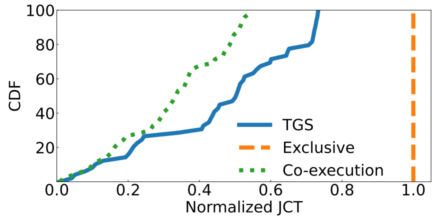
TGS uses an adaptive rate control approach to allocate GPU compute resources between containers in order to simultaneously achieve high GPU utilization and performance isolation. In this experiment, we show that TGS packs an opportunistic job with a production job on a GPU to increase GPU utilization when the production job cannot use up all the GPU resources, and that the overhead of the production job is 5% to 10.8%. We use two different pairs of DNN models for the production job and the opportunistic job to evaluate TGS under different scenarios of resource contention. In this experiment, the total required GPU memory of the two jobs does not exceed the GPU memory capacity. This allows us to focus on evaluating the effectiveness of adaptive rate control. In the experiment, the two jobs arrive at the same time, and we measure the throughput for each job. To clearly show the difference between the five mechanisms, we normalize the throughput of each mechanism to that of Exclusive.



(a) Average JCT.



(b) CDF of production jobs.



(c) CDF of opportunistic jobs.

Figure 6: Performance comparison under a mixed workload job stream.

Figure 4a compares the performance of the five mechanisms when the production job trains ShuffleNet with batch size 4 and the opportunistic job trains MobileNet with batch size 4. These two models are small, so this case has low resource contention, and the throughput of the production job and the opportunistic job is almost the same for the five mechanisms. The overhead of TGS is 5%.

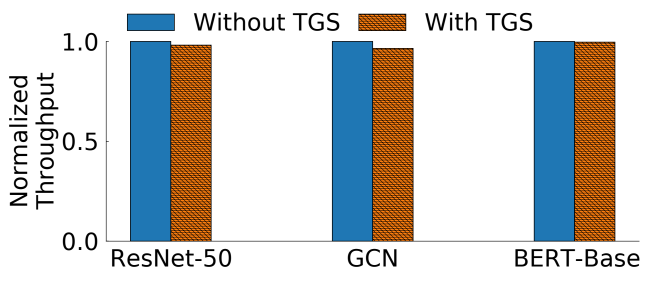
Figure 4b shows the results when the production job trains ResNet-50 with batch size 24 and the opportunistic job trains ShuffleNet with batch size 64. Both models are more computation-intensive than the models in Figure 4a. Thus, this case has a higher resource contention. TGS and MPS provide higher performance of the production job compared to Co-execution, because TGS and MPS control the resource allocation. Co-execution does not provide performance isolation, so the contention with the opportunistic job causes the throughput of the production job to reduce to 57% of that under Exclusive. The opportunistic job gets more resources than it should get by contending with the production job under co-execution. Thus the throughput of the opportunistic job under co-execution is high. TGS incurs 10.8% overhead for the production job although the resource contention is high. The performance provided by MPS is also comparable with TGS, although MPS sacrifices fault isolation. MIG only provides limited configurations for each GPU instance. On an NVIDIA A100 GPU, each GPU instance can only use at most one half of the total GPU memory and 4/7 of total SMs for GPU computation when a GPU is partitioned into two instances. In the high contention scenario, when the production job needs more GPU SMs than 4/7 for computation, the performance of the production job suffers, and is reduced to 77% of that under Exclusive. The opportunistic job gets more resources than it should, so its throughput is quite high.

While TGS protects production jobs from high contention caused by the opportunistic job, some sharing overhead is inevitable. In terms of throughput, Exclusive slightly outperforms TGS, because Exclusive runs DL models exclusively on the GPU. However, in this case, opportunistic jobs have to wait until the completion of the production job before execution. This leads to longer JCT for opportunistic jobs. Figure 4c shows that as the ratio of the job duration of the production job to that of the opportunistic job becomes larger, TGS can significantly reduce the queuing delay and thus speed up the opportunistic job over Exclusive. When the ratio is 20, TGS can reduce the JCT of the opportunistic job by 95% than Exclusive at the low-contention scenario and by 47% at the high-contention scenario.

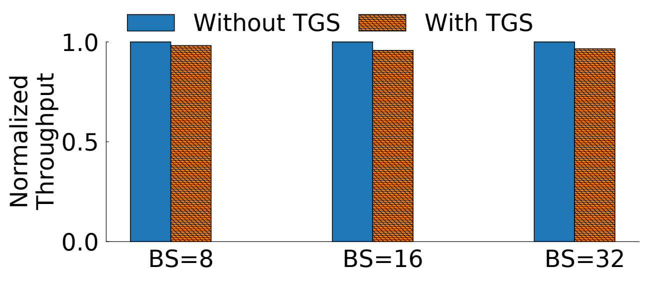
6.2 Unified Memory Management

In this experiment, we show that TGS provides high GPU utilization and performance isolation for GPU sharing even when the GPU memory is oversubscribed. We use two different pairs of DNN models to evaluate TGS under different scenarios. To oversubscribe the GPU memory, we use DLRM as the model of the opportunistic job for both pairs. DLRM is a large recommendation model with high GPU memory consumption. Similar to previous experiments, two jobs arrive at the same time, and we measure the throughput of each job. To clearly show the differences between the five mechanisms, we normalize the throughput of each mechanism to that of Exclusive for each job. Because MPS and Co-execution do not support GPU memory oversubscription, we modify the DL frameworks to use unified memory to evaluate them.

Figure 5a compares the performance of the five mechanisms when the production job trains ResNet-50 with batch size 16 and the opportunistic job trains DLRM with batch size 2048. The overhead of TGS is 2.3% compared to Exclusive. Co-execution has lower throughput due to resource contention. While MPS can set resource limits for SM usage, it cannot prioritize GPU memory allocation, and the two jobs contend for GPU memory resources when the GPU memory is oversubscribed. This causes significant memory swapping between GPU memory and host memory for both jobs, which degrades the performance of the production job under GPU memory oversubscription. MIG can partition the GPU memory resources, but it cannot provide sufficient GPU SMs with the production job due to the configuration constraints. Therefore, the performance of the production job under MIG is lower than that of Exclusive and TGS. In terms of the opportunistic job, Co-execution and MPS have lower throughput due to GPU memory contention. TGS improves the throughput by 7.8× over MPS for the opportunistic job by prioritizing memory allocation. MIG cannot partition GPU memory flexibly. The GPU instance of the opportunistic job can only use one half of the GPU memory to maintain performance of the production job. Therefore, the throughput of the opportunistic



(a) Different DNN models.



(b) Different batch sizes.

Figure 7: System overhead of TGS.

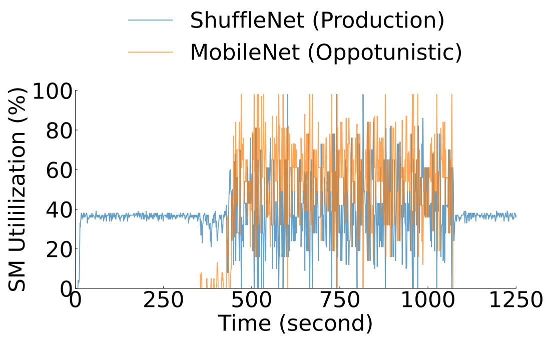
job under MIG is even lower than that of Co-execution and MPS.

Figure 5b shows the results when the production job trains BERT-Base with batch size 4 and the opportunistic job trains DLRM with batch size 256. BERT-Base is more computation-intensive than ResNet-50, and thus there is heavier contention. TGS maintains the performance as Exclusive with 12.3% overhead for the production job. Due to heavier contention, Co-execution and MPS perform worse for the production job. Due to more GPU compute resource demand, MIG performs also worse. TGS improves the throughput by 36× over Co-execution, 72× over MPS, and 1.5× over MIG for the production job. TGS also performs the best for the opportunistic job compared to MIG, MPS and Co-execution. They are slower due to resource contention and simply use unified memory without leveraging priority information. For the opportunistic job, TGS improves the throughput by 24×, 15× and 259×, compared to co-execution, MPS, and MIG, respectively.

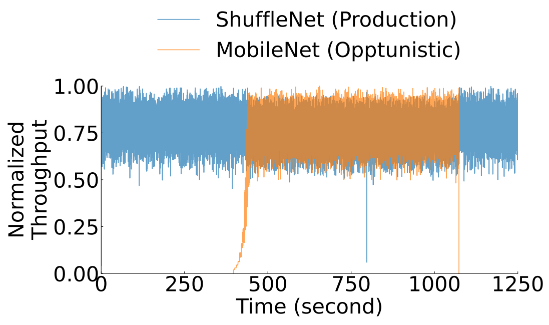
Exclusive provides all GPU resources to the production job, even though GPU resources are not fully utilized. As a result, the opportunistic job has a long queuing time—it has to wait for the production job to finish before it can be executed. As shown in figure 5c, when the ratio of the job duration of the production job to that of the opportunistic job reaches 20, TGS reduces the JCT of the opportunistic job by 95% over Exclusive at the low-contention scenario and by 92% at the high-contention scenario.

6.3 Mixed Workload Job Stream

In this experiment, we compare TGS with Exclusive and Co-execution when sharing a GPU between a mixed workload job stream. The DNN models used in the trace are consistent with previous experiments, including ResNet-50, MobileNet,



(a) GPU utilization.



(b) Training throughput.

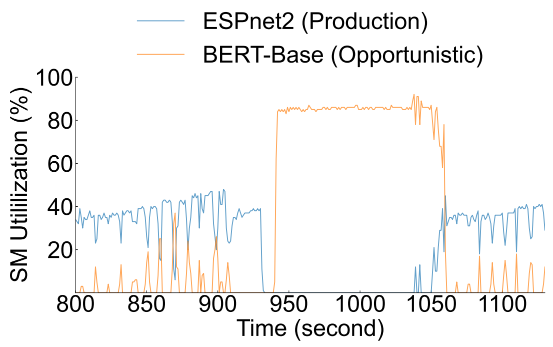
Figure 8: Convergence under dynamic job arrival.

ShuffleNet, GCN, BERT-Base, and DLRM. The running time of the jobs are from a production DL training job trace of Microsoft [5]. The job stream contains 100 jobs, where half are production jobs and the other half are opportunistic jobs. We use fast-forwarding [18] to speed up the experiment. NVIDIA MIG and NVIDIA MPS cannot dynamically change GPU resources allocated to a DL training job, so we do not compare them in this experiment.

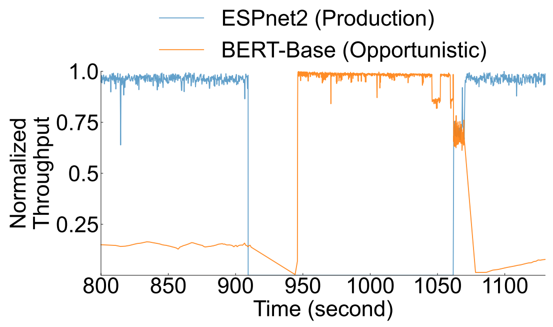
Figure 6a shows the average JCT when executing the trace. For fair comparison, we normalize the JCT of each mechanism to that of Exclusive for each job. As shown in figure 6b, because Co-execution cannot protect production jobs from contention caused by GPU sharing, the average normalized JCT of production jobs under Co-execution is 135% of that under Exclusive, while TGS only incurs 6% overhead. Compared to Exclusive, Figure 6c shows that TGS can significantly reduce the JCT of opportunistic jobs. This is because TGS can reduce the queueing time of opportunistic jobs, as they can use remaining GPU resources not used by production jobs, instead of waiting for production jobs to complete. TGS reduces the average normalized JCT of opportunistic jobs to 48% of that under Exclusive.

6.4 System Overhead

TGS monitors the rate of production jobs, and relies on the monitoring to decide whether a GPU can be shared and how many resources can be allocated to opportunistic jobs. When a GPU is shared, experiments in previous sections have demonstrated that opportunistic jobs do not greatly affect production

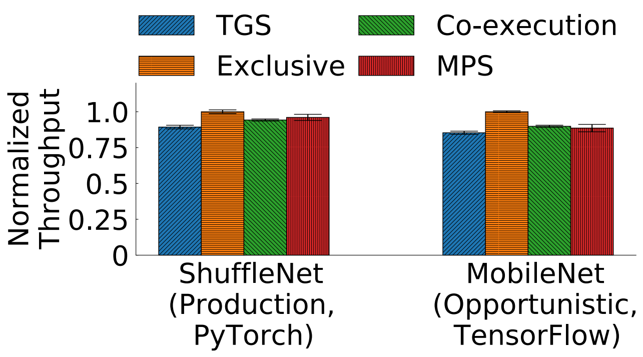


(a) GPU utilization.

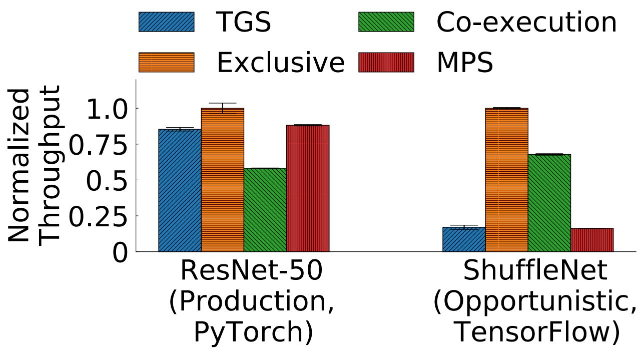


(b) Training throughput.

Figure 9: Convergence under dynamic resource usage.



(a) Low-contention scenario.



(b) High-contention scenario.

Figure 10: GPU sharing between different DL frameworks.

jobs. In this experiment, we explore the system overhead of the rate monitoring component in TGS. We measure the throughput of a job with and without TGS for different configurations, and normalize the throughput to that without TGS.

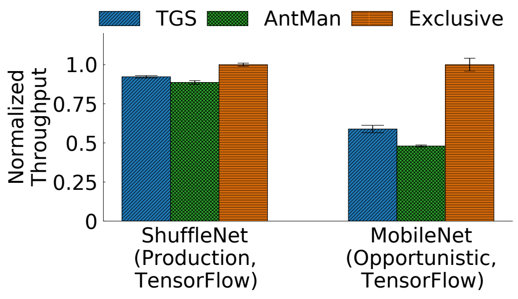
Figure 7a shows the throughput under different DNN models. The throughput is almost the same with and without TGS for ResNet-50, GCN and BERT-Base. Figure 7b shows the throughput under different batch sizes. We use ResNet-50 as the DNN model. Similarly, the JCT is almost the same with and without TGS for batch size 8, 16 and 32. The results demonstrate that the rate monitoring component of TGS incurs 0.3% to 5% overhead for production jobs.

6.5 Convergence

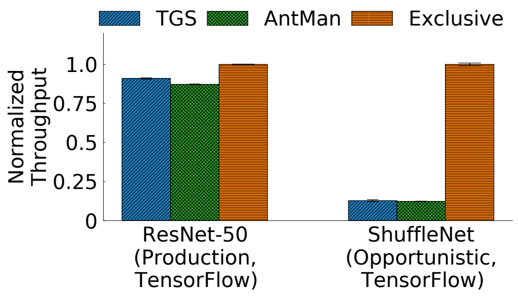
We evaluate the convergence of TGS in different scenarios. The first scenario evaluates the convergence under *dynamic job arrivals*, i.e., a job arrives in the middle to share the GPU with an existing job. In this scenario, the production job training ShuffleNet with batch size 4 is running in the beginning. The opportunistic job training MobileNet with batch size 4 is started after 350 seconds and runs for 240 seconds before it finishes. Figure 8a and Figure 8b show the time series of the GPU utilization and normalized throughput, respectively. As shown in Figure 8a, there are still idle GPU resources when the production job runs, so the total GPU utilization increases when the two jobs run concurrently and share the GPU. Figure 8b shows that the throughput of the opportunistic job increases when it is launched at 350 seconds. At the same time, GPU sharing does not affect the throughput of the production job.

The second scenario evaluates the convergence under *dynamic resource usage*, i.e., a job dynamically switches between high and low GPU utilization, and the other job utilizes the unused GPU resources. In this scenario, the production job trains ESPnet2 with batch size 1 and the opportunistic job trains BERT-Base with batch size 16. ESPnet2 has several phases, so it changes GPU utilization periodically. Figure 9a and Figure 9b show the time series of the GPU utilization and normalized throughput during a transition, respectively. When ESPnet2 needs more GPU resources, the production job keeps its maximum throughput. Between 910 and 940 seconds, ESPnet2 does not train, but runs validation in the GPU. Thus ESPnet2 still utilizes GPU but the throughput is zero. After 940 seconds, ESPnet2 runs into a phase that primarily uses CPU, and Figure 9a shows that the GPU utilization of ESPnet2 decreases to 0. TGS detects the change and dynamically allocates more GPU resources to the opportunistic job. After 1060 seconds, the production job starts using GPU again and reclaims all GPU resources. TGS ensures that the production job is not greatly affected by the opportunistic job.

In summary, these experiments demonstrate that TGS can converge in different scenarios. On the contrary, MIG cannot change GPU resource allocation to each GPU instance whenever there is a job running on the GPU, and MPS cannot change GPU resources allocated to a job after the job begins.



(a) Low-contention scenario.



(b) High-contention scenario.

Figure 11: Comparison between TGS and AntMan.

6.6 Supporting Different DL Frameworks

The experiments in previous sections are based on PyTorch, because TensorFlow-like frameworks claim all GPU memory by default when DL models start and the baselines cannot be directly used for GPU sharing for these frameworks. Specifically, Co-execution does not support GPU memory oversubscription or GPU memory allocation on demand. When one job claims all GPU memory, another job cannot use any GPU memory and would be aborted under Co-execution. MPS also suffers from this behavior. To compare TGS with them, we modify DL frameworks to use CUDA unified memory and enable dynamic GPU memory allocation.

Figure 10a compares the performance of the four mechanisms when the production job trains ShuffleNet with batch size 4 on PyTorch and the opportunistic job trains MobileNet with batch size 4 on TensorFlow. The result is similar to that of Figure 4a.

Figure 10b compares the performance of the four mechanisms in the high contention scenario. The production job trains ResNet-50 with batch size 16 and the opportunistic job trains ShuffleNet with batch size 32. Similar to figure 4b, TGS reduces the throughput of the production job by 14% compared to Exclusive, while Co-execution reduces the throughput by 41%. MPS achieves comparable performance, but it has to be manually tuned and breaks fault isolation.

6.7 Comparison with AntMan

In this experiment, we compare TGS with AntMan [6], which is a state-of-the-art application-layer solution for GPU sharing. AntMan is closely coupled with DL frameworks and uses an application-layer metric, iteration time, to control the opportunistic job. The open-sourced GitHub repository of AntMan is not fully functional. It does not include the logic to dynamically allocate resources to jobs. We contacted the authors of AntMan and followed their instructions to add necessary code in order to run AntMan. Figure 11a and 11b show the comparison under low-contention (ShuffleNet with batch size 4 and MobileNet with batch size 4) and high-contention scenarios (ResNet-50 with batch size 8 and ShuffleNet with batch size 4), respectively. Although AntMan uses application-layer knowledge and controls the jobs at the application layer, TGS still achieves similar performance to AntMan. The throughput of the production job under TGS is 104.1% to 104.3% than that under AntMan, while the throughput of the opportunistic job using TGS is 103% to 122% than that of AntMan. Compared to AntMan, TGS provides the same benefit of GPU sharing and is transparent to DL frameworks.

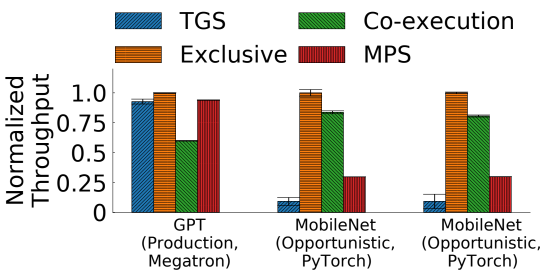


Figure 12: GPU sharing with the large model.

6.8 GPU Sharing for Large Model Training

In §6.2, we have shown that even if a large model (e.g., DLRM) with large batch size (e.g., 2048) and large memory consumption (e.g., 38 GB) runs on a GPU, TGS can still mostly maintain the performance of the production job, while providing the remaining GPU resources to the opportunistic job. In this experiment, we show that although it is not a common scenario, TGS can provide GPU sharing capability when training a bigger model (e.g., GPT). We train a GPT with batch size 32 using two NVIDIA A100 GPUs as the production job, while running two single-GPU opportunistic jobs training MobileNet with batch size 4. Figure 12 shows that TGS still can achieve comparable performance compared to MPS, while MPS breaks fault isolation and Co-execution breaks performance isolation. NVIDIA MIG does not support multi-GPU jobs when a GPU is partitioned into several GPU instances, so it is not evaluated in this case.

7 Discussion

Distributed training. Many solutions have been proposed to achieve high GPU utilization for distributed training jobs [19–23]. With these solutions, it is unlikely that a distributed training job would leave substantial GPU resources unused; otherwise, the job should reduce its GPUs. Therefore, there is little need for TGS. It is most suitable for sharing GPUs between single-GPU jobs, which is also how GPU sharing is used in previous solutions [6, 12, 13]. Yet, TGS can be applied to increase GPU utilization for unoptimized distributed jobs, by controlling the GPU resource usage of an opportunistic job on each GPU as for single-GPU jobs.

GPU cluster scheduling. Many solutions [7, 18, 24–28] have been proposed to minimize job completion time and provide fairness for a GPU cluster. GPU cluster scheduling is orthogonal and complementary to TGS. TGS provides the mechanism for transparent GPU sharing, which can be used by cluster schedulers when they schedule and place jobs. We note that some schedulers [18, 28] pack multiple jobs on a GPU, which are at the application level and require modifications to DL frameworks. Also, they do not support GPU memory oversubscription. These schedulers can benefit from TGS.

Space sharing and time sharing. The concepts of GPU compute sharing and memory sharing are orthogonal to space sharing and time sharing. Sharing GPU compute resources can be done either in space sharing or in time sharing. The adaptive rate control mechanism and transparent unified memory mechanism of TGS can be used either in space sharing or in time sharing. GPU space sharing needs hardware support and is not well supported. Current space sharing solutions reduce performance isolation (e.g. MIG) or fault isolation (e.g. MPS). Therefore, TGS currently uses time sharing.

8 Related Work

Deep learning systems. Many DL frameworks have been proposed for developing and running DNN models [29–36]. Some works optimize communication to improve distributed training performance [19–23]. Some works use memory swapping to handle the GPU memory problem for training large DNN models [16,37–39]. They focus on improving the performance of a single training job, while TGS provides a solution for improving the GPU utilization of running many jobs in a cluster. Some works [40, 41] propose algorithms for inter-job GPU memory management, but they are not transparent to applications and require modifications to DL frameworks. GPUswap [42] proposes a transparent GPU memory swapping system, but it needs to modify GPU drivers. However, most current commercial GPU drivers, such as NVIDIA GPU drivers, are not open-source. Open-source GPU drivers are not as high performance as the commercial ones, so they are not widely used for DL training workloads. MIG-Serving [43] tries to find better configurations to use MIG for GPU sharing. However, MIG itself has limitations as described above. We compare MIG with the best configuration and TGS in the evaluation section, and show the benefits of TGS. There are many solutions for optimizing DL inference workloads [44, 45]. We focus on GPU clusters for training workloads in this paper. Several scheduling algorithms have been designed to schedule DL training jobs in a GPU cluster [7, 18, 24–28]. These works are orthogonal to TGS.

Containers. Containers provide lightweight virtualization for applications. Due to the benefits of portability, isolation and performance, containers are widely used in datacenters. Major public cloud services, such as AWS, Microsoft Azure and Google Cloud, offer containers as a service [46–48]. Many container runtimes (e.g., Docker) and orchestration systems (e.g., Kubernetes) are developed and deployed [1–3, 10, 11, 49, 50]. Some work is proposed to provide high-performance networking with isolation [51–56]. These solutions are orthogonal to TGS, which focuses on improving GPU utilization.

GPU sharing. Several solutions have been proposed for GPU sharing. Early solutions [57–65] explored OS-layer techniques like driver call interception and application-layer techniques like introducing new programming APIs, for sharing GPU between applications. They focus on jobs with a few kernels, and are not specifically designed for DL training that typically has hundreds of kernels. With the emergence of DL applications, recent solutions [6, 12, 13] have been designed for GPU sharing of DL training. AntMan [6] is the state-of-the-art application-layer solution for GPU sharing. Salus [12] uses centralized GPU memory management and kernel scheduling for GPU sharing. It requires all the applications to fit in the GPU memory. PipeSwitch [13] provides fast context switching for DNN jobs, but only one job can run at each time. They all modify DL frameworks. MPS [9] is an OS-layer solution, but it requires application knowledge to correctly set resource limits, does not support GPU memory oversubscription and does not provide fault isolation. Planaria [66] is an accelerator designed for the multi-tenant scenario. In comparison, TGS is a software solution that can be used for sharing a variety of hardware.

9 Conclusion

We have presented TGS, a system that transparently shares GPUs for DL workloads to improve GPU utilization in container clouds. TGS is distinguished from state-of-the-art application-layer solutions in that it enables users to use any DL framework and library to develop and train DNN models in containers. Shared GPUs are exposed to containers as regular GPU devices, and TGS transparently runs multiple containers on a GPU when a single container cannot utilize all GPU resources. TGS achieves both high utilization and decent performance isolation.

Acknowledgments. We sincerely thank our shepherd John Wilkes and the anonymous reviewers for their valuable feedback. This work was supported by the National Key Research and Development Program of China under the grant number 2020YFB2104100, the National Natural Science Foundation of China under the grant number 62172008 and the National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas). Xin Jin is the corresponding author. Bingyang Wu, Zili Zhang, Xuanzhe Liu and Xin Jin are also with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

References

[1] "containerd." https://containerd.io/.

[2] "cri-o." https://cri-o.io/.

[3] "Docker." https://www.docker.com/.

[4] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, 2015.

[5] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of large-scale multi-tenant GPU clusters for DNN training workloads," in *USENIX ATC*, 2019.

[6] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia, "Antman: Dynamic scaling on GPU clusters for deep learning," in *USENIX OSDI*, 2020.

[7] W. Qizhen, X. Wencong, Y. Yinghao, W. Wei, W. Cheng, H. Jian, L. Yong, Z. Liping, L. Wei, and D. Yu, "MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters," in *USENIX NSDI*, 2022.

[8] H. Zhao, Z. Han, Z. Yang, Q. Zhang, F. Yang, L. Zhou, M. Yang, F. C. Lau, Y. Wang, Y. Xiong, and B. Wang, "HiveD: Sharing a GPU cluster for deep learning with guarantees," in *USENIX OSDI*, 2020.

[9] "CUDA Multi-Process Service." https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[10] "Kubernetes." https://kubernetes.io/.

[11] "Docker Swarm." https://docs.docker.com/engine/swarm/.

[12] P. Yu and M. Chowdhury, "Salus: Fine-grained GPU sharing primitives for deep learning applications," in *Conference on Machine Learning and Systems*, 2020.

[13] Z. Bai, Z. Zhang, Y. Zhu, and X. Jin, "Pipeswitch: Fast pipelined context switching for deep learning applications," in *USENIX OSDI*, 2020.

[14] "Nvidia multi-instance GPU (MIG)." https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

[15] "Nvidia multi-instance GPU user guide." https://docs.nvidia.com/datacenter/tesla/mig-user-guide/.

[16] "CUDA Unified Memory." https://devblogs.nvidia.com/unified-memory-cuda-beginners/.

[17] J. Gleeson, S. Krishnan, M. Gabel, V. J. Reddi, E. de Lara, and G. Pekhimenko, "RL-Scope: Cross-stack profiling for deep reinforcement learning workloads," in *Conference on Machine Learning and Systems*, 2021.

[18] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou, "Gandiva: Introspective cluster scheduling for deep learning," in *USENIX OSDI*, 2018.

[19] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[20] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed DNN training acceleration," in *ACM SOSP*, 2019.

[21] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters," in *USENIX OSDI*, 2020.

[22] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "PipeDream: generalized pipeline parallelism for DNN training," in *ACM SOSP*, 2019.

[23] G. Wang, S. Venkataraman, A. Phanishayee, J. Thelin, N. Devanur, and I. Stoica, "Blink: Fast and generic collectives for distributed ML," in *Conference on Machine Learning and Systems*, 2020.

[24] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo, "Tiresias: A GPU cluster manager for distributed deep learning," in *USENIX NSDI*, 2019.

[25] H. Zhang, L. Stafman, A. Or, and M. J. Freedman, "Slaq: quality-driven scheduling for distributed machine learning," in *ACM Symposium on Cloud Computing*, 2017.

[26] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," in *EuroSys*, 2018.

[27] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla, "Themis: Fair and efficient GPU cluster scheduling," in *USENIX NSDI*, 2020.

[28] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, "Heterogeneity-aware cluster scheduling policies for deep learning workloads," in *USENIX OSDI*, 2020.

[29] "TensorFlow." https://www.tensorflow.org/.

[30] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for large-scale machine learning," in *USENIX OSDI*, 2016.

[31] "PyTorch." https://pytorch.org/.

[32] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, 2019.

[33] "MXNet." https://mxnet.apache.org/.

[34] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," in *LearningSys at Neural Information Processing Systems*, 2015.

[35] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *USENIX OSDI*, 2014.

[36] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. a. Ranzato, A. Senior, P. Tucker, K. Yang, Q. Le, and A. Ng, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems*, 2012.

[37] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[38] C.-C. Huang, G. Jin, and J. Li, "SwapAdvisor: Pushing deep learning beyond the GPU memory limit via smart swapping," in *ACM ASPLOS*, 2020.

[39] G. Wang, K. Wang, K. Jiang, X. LI, and I. Stoica, "Wavelet: Efficient dnn training with tick-tock scheduling," in *Conference on Machine Learning and Systems*, 2021.

[40] X. Peng, X. Shi, H. Dai, H. Jin, W. Ma, Q. Xiong, F. Yang, and X. Qian, "Capuchin: Tensor-based gpu memory management for deep learning," in *ACM ASPLOS*, 2020.

[41] G. Lim, J. Ahn, W. Xiao, Y. Kwon, and M. Jeon, "Zico: Efficient GPU memory sharing for concurrent DNN training," in *USENIX ATC*, 2021.

[42] J. Kehne, J. Metter, and F. Bellosa, "GPUswap: Enabling oversubscription of gpu memory through transparent swapping," in *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2015.

[43] C. Tan, Z. Li, J. Zhang, Y. Cao, S. Qi, Z. Liu, Y. Zhu, and C. Guo, "Serving dnn models with multi-instance gpus: A case of the reconfigurable machine scheduling problem," *arXiv preprint arXiv:2109.11067*, 2021.

[44] J. Kosaian, K. V. Rashmi, and S. Venkataraman, "Parity models: Erasure-coded resilience for prediction serving systems," in *ACM SOSP*, 2019.

[45] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram, "Nexus: A GPU cluster engine for accelerating DNN-based video analysis," in *ACM SOSP*, 2019.

[46] "AWS containers." https://aws.amazon.com/containers/.

[47] "Microsoft azure containers." https://azure.microsoft.com/en-us/product-categories/containers/.

[48] "Google cloud containers." https://cloud.google.com/containers.

[49] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *EuroSys*, 2015.

[50] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *EuroSys*, 2013.

[51] D. Kim, T. Yu, H. H. Liu, Y. Zhu, J. Padhye, S. Raindel, C. Guo, V. Sekar, and S. Seshan, "Freeflow: Software-based virtual RDMA networking for containerized clouds," in *USENIX NSDI*, 2019.

[52] D. Zhuo, K. Zhang, Y. Zhu, H. H. Liu, M. Rockett, A. Krishnamurthy, and T. Anderson, "Slim: OS kernel support for a low-overhead container overlay network," in *USENIX NSDI*, 2019.

[53] B. Li, T. Cui, Z. Wang, W. Bai, and L. Zhang, "Socksdirect: Datacenter sockets can be fast and compatible," in *ACM SIGCOMM*, 2019.

[54] Z. He, D. Wang, B. Fu, K. Tan, B. Hua, Z.-L. Zhang, and K. Zheng, "Masq: RDMA for virtual private cloud," in *ACM SIGCOMM*, 2020.

[55] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. Olson, M. Ryan, E. Rubow, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat, "Snap: a microkernel approach to host networking," in *ACM SOSP*, 2019.

[56] A. Narayan, A. Panda, M. Alizadeh, H. Balakrishnan, A. Krishnamurthy, and S. Shenker, "Bertha: Tunneling through the network API," in *ACM SIGCOMM HotNets Workshop*, 2020.

[57] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, "A GPGPU transparent virtualization component for high performance computing clouds," in *European Conference on Parallel Processing*, 2010.

[58] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan, "GViM: GPU-accelerated virtual machines," in *ACM Workshop on System-level Virtualization for High Performance Computing*, 2009.

[59] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Ortí, "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," in *International Conference on High Performance Computing & Simulation*, 2010.

[60] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar, "Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework," in *IEEE HPDC*, 2011.

[61] L. Shi, H. Chen, J. Sun, and K. Li, "vCUDA: GPU-accelerated high-performance computing in virtual machines," *IEEE Transactions on Computers*, vol. 61, 2011.

[62] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU concurrency with elastic kernels," in *ACM ASPLOS*, 2013.

[63] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative preemption for multitasking on a shared GPU," in *ACM ASPLOS*, 2015.

[64] T. T. Yeh, A. Sabne, P. Sakdhnagool, R. Eigenmann, and T. G. Rogers, "Pagoda: Fine-grained GPU resource virtualization for narrow tasks," in *ACM PPoPP*, 2017.

[65] K. Zhang, B. He, J. Hu, Z. Wang, B. Hua, J. Meng, and L. Yang, "G-NET: Effective GPU sharing in NFV systems," in *USENIX NSDI*, 2018.

[66] S. Ghodrati, B. H. Ahn, J. Kyung Kim, S. Kinzer, B. R. Yatham, N. Alla, H. Sharma, M. Alian, E. Ebrahimi, N. S. Kim, C. Young, and H. Esmaeilzadeh, "Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks," in *IEEE/ACM MICRO*, 2020.

A Convergence of Adaptive Rate Control Algorithm

We assume each GPU has an unknown constant throughput limit B. The TGS's goal is to maximize throughput of the opportunistic job without affecting the production job very much. We assume throughput of the production job is relatively stable. Therefore, the adaptive rate control algorithm can accurately measure the throughput of the production job, i.e. α_{in}. When throughput of the production job is unstable beyond a manual tuned threshold, TGS re-estimates α_{in}. In this context, we define that a *cycle* is a phase starting after TGS detects contention and ending when TGS detects contention again. A *step* is defined as an invocation of the rate control component to adjust rate limit of the opportunistic job, such as an additive increase or a multiplicative decrease. Hence, a *cycle* consists of one multiplicative decrease *step* and multiple continous additive increase *step*s. Let the initial value of β_{out} be β_0 ($\beta_0 \leq B$). The simplified convergence of the rate adaptive control algorithm is shown as follow:

Opportunistic Job	production Job
β_0	$\min(R, B-\beta_0)$
$\beta_1 = \beta_0 + \delta_{AI}$	$\min(R, B-\beta_1)$
$\beta_2 = \beta_0 + \delta_{AI} + \delta_{AI}$	$\min(R, B-\beta_2)$
$\vdots$	$\vdots$
$\beta_k = \beta_0 + \underbrace{\delta_{AI} + \cdots + \delta_{AI}}_{k}$	$\min(R, B-\beta_k)$
Detect Contention:	$R + \beta_0 + k\delta_{AI} \geq B$
Action:	Multiplicative Decrease
$\beta_{k+1} = \frac{\beta_0 + k\delta_{AI}}{\delta_{MD}}$	$\min(R, B-\beta_{k+1})$
$\beta_{k+2} = \frac{\beta_0}{\delta_{MD}} + \frac{k\delta_{AI}}{\delta_{MD}} + \delta_{AI}$	$\min(R, B-\beta_{k+2})$
$\beta_{k+3} = \frac{\beta_0}{\delta_{MD}} + \frac{k\delta_{AI}}{\delta_{MD}} + \delta_{AI} + \delta_{AI}$	$\min(R, B-\beta_{k+3})$
$\vdots$	$\vdots$
$\beta_{k+l+1} = \frac{\beta_0}{\delta_{MD}} + \frac{k\delta_{AI}}{\delta_{MD}} + \underbrace{\delta_{AI} + \cdots + \delta_{AI}}_{l}$	$\min(R, B-\beta_{k+l+1})$
Detect Contention:	$R + \frac{\beta_0}{\delta_{MD}} + \frac{k\delta_{AI}}{\delta_{MD}} + l\delta_{AI} \geq B$
Action:	Multiplicative Decrease
$\beta_{k+l+2} = \frac{\beta_0}{\delta_{MD}^2} + \frac{k\delta_{AI}}{\delta_{MD}^2} + \frac{l\delta_{AI}}{\delta_{MD}}$	$\min(R, B-\beta_{k+l+2})$
$\vdots$	$\vdots$
$\beta^* = \frac{\beta_0}{\delta_{MD}^{\log\beta_0}} + \frac{k\delta_{AI}}{\delta_{MD}^{\log\beta_0}} + \frac{l\delta_{AI}}{\delta_{MD}^{\log\frac{\beta_0}{2}}} + \cdots + m$	$\min(R, B-\beta^*)$

We assume the unit of bandwith is indivisible. As shown above, the adaptive rate control algorithm converge in $O(\log\beta_0)$ *cycle*s, because the unknown term β_0 decreases to zero in $O(1+\log\beta_0)$ *cycle*s, i.e. $O(B\log B)$ *step*s. Therefore, the complexity of the adaptive rate control algorithm is $O(B\log B)$.