Design and Operation of Elastic GPU-pooling on Campus

Kaicheng Guo[1][0000−0003−0493−1147], Jingyi Chen[1], Yun Wang[1][0009−0009−7969−6891], Semakin Anton[2], Tovmachenko Dmitry[2], Jiajie Sheng[3], Jianwen Wei[3], James Lin[3], Zhengwei Qi[1][0000−0003−2730−2319]✉, and Haibing Guan[1]

[1] Shanghai Jiao Tong University, Shanghai, China
guokaicheng@sjtu.edu.cn 2744234012@qq.com
{yunwang94, qizhwei, hbguan}@sjtu.edu.cn
[2] Huawei Technologies Co. Ltd.
{semakin.anton, tovmachenko.dmitry}@huawei.com
[3] Center for High Performance Computing, Shanghai Jiao Tong University, Shanghai, China
{jiajie.sheng, weijianwen, james}@sjtu.edu.cn

Abstract. With the rapid advancement of Artificial Intelligence (AI) and High-Performance Computing (HPC), GPU-centric clusters have become pivotal in driving research across diverse disciplines. Consequently, campus data centers are increasingly aggregating substantial computing resources. However, akin to commercial GPU and AI clusters, campus GPU clusters frequently encounter challenges related to GPU underutilization.

While pooling technologies offer a viable solution to this issue, existing GPU pooling approaches fall short in effectively supporting environments where multiple applications coexist within campus data centers.

This paper presents gPooling, a novel pooling scheme that leverages device driver hijacking to optimize GPU resource allocation. We designed a benchmark based on real-world traces from a campus data center and deployed gPooling within a GPU cluster environment. Experimental results from both benchmarking and actual deployment demonstrate that gPooling significantly enhances GPU utilization and reduces user waiting times, thereby improving the overall efficiency of campus GPU clusters.

Keywords: GPU Pooling, Resource Management, GPU virtualization

1 Introduction

In recent years, GPUs have become a core computing resource of supercomputing centers due to their excellent acceleration of multiple computational tasks [5,16,1]. GPUs are more costly than other resources, but suffer underutilization in multiple data centers and various application scenarios [17,13,8,20]. On the other hand, GPU-accelerated computing - particularly in AI and HPC technologies - has become foundational to advancing research across multiple

scientific disciplines. Consequently, the demand for GPU resources within the research community has surged dramatically [7,1,10]. However, academic data centers face more acute GPU shortages compared to commercial counterparts operated by profit-driven organizations [9][4].

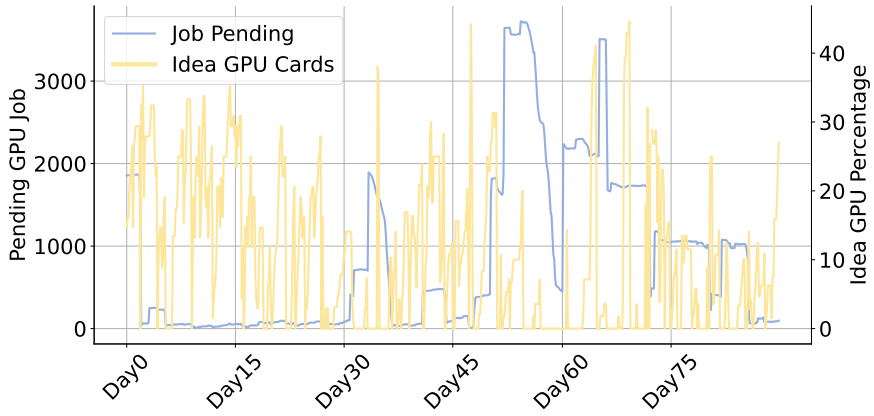

Fig. 1: The GPU utilization and the number of pending jobs trace: GPU underutilization and GPU shortage coexist.

We observe a paradoxical coexistence of chronic GPU underutilization and severe resource scarcity in campus data centers, a phenomenon uniquely shaped by their distinctive workload patterns. Figure 1 illustrates the GPU utilization and the number of pending jobs for 8 nodes in a campus data center over a 12-hour period. As shown in Figure 1, the GPU utilization of the 8 nodes is just below 25%, but at the same time, the number of pending jobs is over 200. Based on the statistics of the number of submitted jobs, the load in this data center is, at the moment, mainly small model machine learning models, large model machine learning, and scientific computing. Most of the tasks do not use the full GPU capacity but monopolize a GPU card. This causes significant resource waste, leading to GPU underutilization and long user queues.

Pooling based on GPU sharing is an effective solution to this problem [14,13,19]. Various technical approaches have been proposed in recent years to achieve effective and stable GPU sharing. Hardware vendors have proposed solutions including MPS[5], MIG[6], and vGPU. However, the software solutions among them face reliability issues and limited usage scenarios. Hardware solutions can only provide limited elasticity. The academic community's solutions [18,12], on the other hand, generally utilize API intercept forwarding at the CUDA layer or the application layer for optimizations that are deeply coupled with the application.

[4] https://www.nature.com/articles/d41586-024-03792-6
[5] https://docs.nvidia.com/deploy/mps/index.html
[6] https://docs.nvidia.com/cuda/mig/index.html

In this paper, we develop gPooling, a kernel-hijacking GPU-pooling system. By intercepting GPU driver APIs, gPooling achieves fine-grained arithmetic, storage control and state management of GPUs. gPooling achieves accurate arithmetic slicing through time-slice control and accurate memory control through interception of memory APIs. Its design is orthogonal to the aforementioned scheme. As such gPooling provides both flexibility and generality at a low-performance overhead.

We have operated gPooling in a real-world GPU cluster. We equip gPooling with GPU nodes in a campus data center. The cluster manages the scheduling of submitted tasks through a Slurm system. We combine gPooling and Slurm by registering vGPUs virtualized through gPooling in Slurm and scheduling them through the Slurm system.

This paper makes the following contributions.

- This paper introduces gPooling, a novel GPU pooling scheme that leverages driver API hijacking to achieve fine-grained resource allocation with low performance overhead.
- gPooling is specifically designed to address the unique challenges of GPU underutilization and long job queues in campus data centers, significantly enhancing GPU utilization and reducing user waiting times.
- gPooling integrates seamlessly with resource management systems like Slurm without modifying their source code or user job scripts, maintaining system stability while enabling efficient vGPU registration and scheduling.
- gPooling employs time-slice multiplexing and real-time utilization feedback to achieve precise computational slicing and performance isolation between vGPUs, ensuring low overhead and dynamic resource adaptation.

2 Background and Motivation

2.1 Characterization of GPU datacenter on campus

In this section, we perform a thorough analysis of our job traces. Some prior works [20,6,8,11] analyzed the traditional big data traces from real-world datacenters. In contrast, less work has been done to reveal how campus data centers [15] have been characterized in recent years.

Job Queuing We analyzed the operational data of a campus data center for a month. This data center employs Slurm for task management and resource allocation across multiple clusters.

We analyzed the job queuing situation within the data center. We examined the number of queued jobs in the clusters. We calculated the ratio of queued jobs to running jobs as an indicator of the severity of queuing issues within the clusters.

Figure 2 illustrates the number of pending and running jobs across different clusters. Figure 3 shows the ratio of pending jobs to running jobs for each cluster.

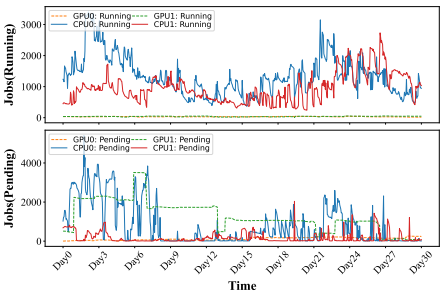

Fig. 2: Pending and Running Job Counts Across Clusters

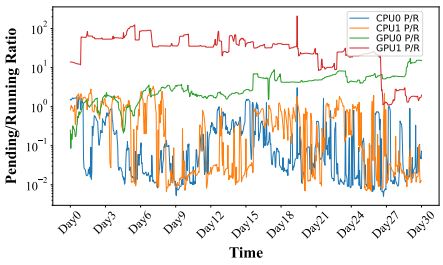

Fig. 3: Ratio of Pending to Running Jobs Across Clusters

As observed, the job queuing in GPU clusters is significantly higher than in CPU clusters. The average queue-to-execution ratio for the two CPU clusters is 0.23 and 0.53, while for the two GPU clusters, the ratios are 3.97 and 36.07, respectively.

The severe queuing issue in the GPU clusters is primarily due to the lack of flexibility in the current GPU scheduling system. At present, the GPU clusters use Slurm-based scheduling. Each job submits the number of GPUs it requires to the scheduling system, which allocates resources at the GPU card level. GPU jobs in the cluster typically require multiple GPU cards. As a result, we also observed *resource fragmentation* at the GPU card level. That is, while multiple tasks are waiting, there is still a certain number of GPU cards that remain idle.

Table 1: Workload categorization statistic.

Application	GPU-hours	Categorization	Percent
PyTorch	310885	ML	33.17%
Visual platform	255763	Mixed	27.23%
Python	241909	Mixed	25.73%
AlphaFold [7]	21896	AI for science	2.34%
TensorFlow	18700	ML	2.00%
GPUMD	16309	HPC	1.74%
Amber	5840	HPC	0.62%
RELION	4500	HPC	0.48%
PLUMED	2991	HPC	0.32%
ColabFold	2451	HPC	0.26%
GROMACS	1068	HPC	0.11%

GPU Workload Statistic Table 1 presents the categorization of workloads in terms of GPU-hours for various applications. The table includes the total GPU

hours utilized by each application, as well as the corresponding categorization, which ranges from machine learning to high-performance computing (HPC) and AI for science. The final column shows the percentage of total GPU hours occupied by each application.

From the table, it is evident that *PyTorch*, *Visualization platform*, and *Python* account for the majority of GPU usage, with *PyTorch* alone consuming over 33% of the total GPU hours. Other applications, such as *AlphaFold* and *TensorFlow*, contribute a smaller portion, highlighting the diverse range of workloads in the GPU clusters. The distribution of GPU-hours also reflects the varying demands across different fields, with machine learning applications occupying a significant share of the total GPU resources.

3 gPooling

To address the problems of GPU underutilization and long queuing times for users (§2), we propose gPooling, a driver-intercept-based GPU virtualization framework that provides low overhead, and highly elastic virtual GPUs (vGPUs) on demand. The core concept behind gPooling is to establish a cooperative and interactive relationship between the application and accelerator to proactively intercept and identify the application's computational power, memory, and management operations on the device and make subsequent optimizations based on this.

3.1 gPooling architecture

The architecture of gPooling is presented in Figure 4. The gPooling consists of several modules that operate both in the operating system kernel and user mode. gPooling exposes virtual GPUs to the user and provides the same kernel driver interface to the user. The computation will be performed in the kernel driver module of gPooling by calling the driver API of the physical device.

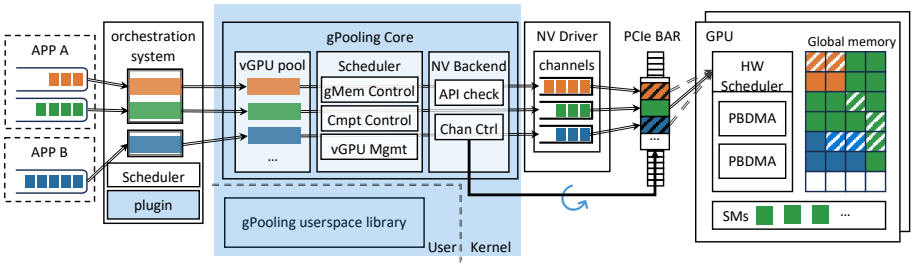

Fig. 4: gPooling architecture

gPooling components

User space library. This module wraps system calls to the gPooling kernel module into library functions called by the container management system. User applications, libraries, and schedule framework invoke functions from this library to manage gPooling virtual devices including but not limited to creating, deleting, and retrieving information. Table 2 lists parts of the API of gPooling's userspace library.

Table 2: gPooling's declarative API

API	Parameters
`create_vgpu`	p_vgpu_config, p_vgpu_address, vendor_id
`delete_vgpu`	p_vgpu_address
`get_vgpu_status`	p_vgpu_address
`set_vgpu_config`	p_vgpu_address, p_vgpu_config

Kernel module. The kernel module is the core of the gPooling system. It maintains information about the virtual gPooling device pool and physical device drivers. It contains modules for gPooling scheduler, gPooling device control, and gPooling device management. The kernel module is registered as an operating system kernel module on host and provides a unified call interface to the gPooling userspace library through system calls.

Device adapter. gPooling enables hardware devices from different vendors to support the unified gPooling core interface through device adapters. The gPooling device adapters implement the functions in the gPooling core module according to the implementation details and driver interfaces of devices from different vendors. For the widely used CUDA ecosystem, we implement the gPooling-NV adapter to intercept CUDA calls. The NV-adapter analyzes all intercepted packet data and examines key packets with specific information, such as memory allocation and deallocation. It also manages the switching tasks that are executed on the GPU. This allows setting limits on memory allocation and GPU computing power utilization and controlling their usage.

Framework adapter. gPooling has good compatibility with mainstream resource management or job scheduling systems, such as K8S and Slurm. gPooling's user mode library provides APIs for vGPU creation, registration and deletion. With these APIs, we can quickly implement plug-ins adapted to upper-layer frameworks to realize the integration of gPooling and mainstream ecosystems.

gPooling Workflow During the initialization phase of gPooling, the userspace library creates a pool of vGPU containing multiple vGPUs and registers these vGPUs in the management system (k8s or). During registration, the gPooling user library calls the GPU memory management interface in the gPooling kernel module to bind physical GPUs and allocate memory for each vGPU. In scenarios

where the management framework is adapted, the scheduler in the management framework assigns vGPUs to tasks or resource units, on demand. At runtime, gPooling scheduler achieves computational power control and performance isolation between vGPUs by managing time slices (§3.2).

Timeslicer gPooling currently implements GPU virtualization through time-slice multiplexing. Therefore the implementation of the time slice mechanism is the core of the virtualization part of gPooling. The low overhead of the time slice component ensures the low-performance loss of gPooling. The granularity of time slices also determines the granularity of GPU arithmetic slicing by gPooling. Previous work [2] has done time-slice rotation at the framework layer, in the user state. However, these solutions cannot meet gPooling's design goals in terms of time slice granularity and overhead. Therefore, gPooling chooses to intercept the device driver calls and decrypt the GPU details based on the field position, and realizes a low overhead and fine-grained time slice rotation mechanism with the help of TSG.

3.2 gPooling scheduler

As shown in Figure 4, the gPooling scheduler contains a series of interfaces for arithmetic control and virtual device management, with three main modules: video memory control, arithmetic control, and device management. The scheduler module provides a unified scheduling interface to realize the operation of specific physical devices by corresponding to the accelerator of different vendors. For Nvidia GPUs, gPooling implements these functions based on the interception of the CUDA/Driver API. Specifically, the NV compactor examines memory-related API calls to enable partitioning and control of vGPU graphics memory. For arithmetic control, gPooling realizes accurate arithmetic slicing based on time-division multiplexing by controlling channels in NV Driver.

vGPU schedule in NV backend For Nvidia GPUs, gPooling's **NV backend** implements Scheduler's features based on the interception of the CUDA/Driver API. Specifically, the **API check** in the NV backend checks memory-related API calls to enable partitioning and control of vGPU graphics memory.

For computational power control, based on previous GPU decryption work [4,3] and Nvidia's open-source GPUs[7][8], we realized accurate computational power slicing based on time-division multiplexing by controlling the channels in the NV Driver. The NV GPU's hardware scheduler will only pull commands from the PushBuffer of a valid channel [4]. Therefore, gPooling binds a corresponding channel to each vGPU. The channel control module ensures that only one channel, i.e., vGPU, is active at any given time. This design, together with a reasonable time rotation algorithm, realizes the precise control of GPUs.

[7] git://nv-tegra. nvidia.com/linux-nvgpu.git
[8] https://github.com/NVIDIA/open-gpu-doc

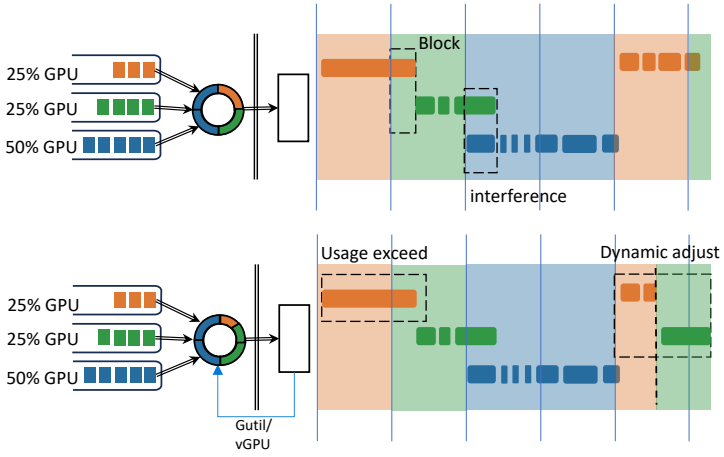

Fig. 5: An example of timeline in gPooling Scheduler

Computational Control Based on Utilization Feedback Adjustment Previous work on GPU arithmetic control through a similar mechanism, such as stream APIs at the application layer or in the user state, exists. However, these works can only achieve time-slice-based slicing, and cannot achieve accurate arithmetic control. Figure 5 illustrates an example comparing static slicing and dynamically regulated time-slice control. In this scenario, the GPU is divided into three vGPUs, each with 25%, 25%, and 50% of the arithmetic power.

In even timeslice, for kernels with time granularity larger than time slice granularity, kernel blocks or inter-vGPU kernel executions may interfere with each other. This will lead to the inability to realize the preset GPU arithmetic slicing. On the other hand, gPooling's utilization feedback mechanism reads the GPU utilization in real time. It reduces the length of the next time window after observing that a vGPU exceeds the preset utilization, which realizes accurate arithmetic power slicing.

3.3 Integrating the gPooling into Slurm

Slurm is one of the most widely used workload management systems, especially in various university clusters. Provides users with rich and direct usage interfaces that simultaneously meet the needs of different scenarios. Therefore, to maintain the stability and maintainability of the cluster, when designing the integration of gPooling and Slurm, we need to fulfill: 1. no modification of Slurm's source code, 2. not modify the user's task scripts, 3. vGPU Adaptation for Slurm's Generic Resource (GRES) Scheduling.

To fulfill these requirements, we propose the gPooling-Slurm compute node. Facing the typical control node-compute node architecture of , we only modify the compute node, while using the native control node and login node. The

gPooling-service and gPooling-SPANK plugin are integrated on the node loaded with the gPooling virtual device.

gPooling-Slurm workflow. In initialization, gPooling-service first calls the gPooling user library to register a number of vGPUs equipped with fixed resources (computational power and GPU memory) according to the configuration. After that, gPooling-SPANK plugin registers a Linux namespace for each vGPU to isolate and restrict access to them in the user mode. The gPooling-SPANK compute node is registered with Slurm by starting slurmd. The Slurm Control Node recognizes the vGPUs of the gPooling-Slurm compute node and schedules them together with the physical devices on the general physical node.

4 Implementation

gPooling core is implemented as a kernel module within the system. It introduces a miscellaneous device that facilitates the management of virtual GPUs (vGPUs) through IOCTL (Input/Output Control) commands. The gPooling schedule implements a set of methods for controlling and managing virtual devices. These methods are built upon the foundational methods defined within the gPooling core. In the gPooling core, these foundational methods are declared as virtual functions, enabling different backends to provide their specific implementations. Various backends, such as the NV backend, register themselves with the gPooling core and provide the interface functions.

gPooling user library is a user-mode shared object library that serves as an interface to the gPooling core. It facilitates communication with the gPooling core by implementing IOCTL calls to the gPoolingcore kernel module. Designed to simplify the management of gPooling virtual devices, the library enables management systems and users to interact with GPU pooling functionalities efficiently.

This library encompasses a comprehensive set of functions that allow client applications to execute precise IOCTL operations on the gPoolingcore. These functions are engineered to ensure accurate and reliable invocation of kernel-level commands, incorporating essential validation checks on incoming requests. This modular design not only improves the robustness of GPU resource management but also provides a user-friendly and adaptable interface for diverse application requirements.

NV backend implements the virtual device management methods for Nvidia GPUs. When it receives the request to create a virtual GPU, it creates one char device file corresponding physical GPU (*/dev/nvidia0*) and one control char device file corresponding to the native control char device (*/dev/nvidiactl*). GPU call from user mode will use the vgpu device files instead of original device files.

The NV backend module has UNLOCKED_IOCTL, OPEN, CLOSE, MMAP file operation functions. These functions are set for the created vgpu device files. Commands received by the module from CUDA are processed by the file operation functions in the NV backend and then transmitted to the NVidia driver by calling the original file operation functions. The NV backend creates a schedul-

ing thread for each physical GPU to monitor memory-related APIs and manage GPU utilization.

5 Evaluations

5.1 Environment Setup

Testbed. In this section, we describe the environment setup used for the experiments. The experiments were conducted on a large-scale, high-performance computing cluster designed to support both CPU and GPU-intensive workloads. The cluster hardware platform is built on dual Intel Xeon Platinum 8358 processors, with each processor featuring 32 cores and a base clock speed of 2.6 GHz. In total, the cluster comprises 936 compute nodes, providing ample processing power for large-scale computations.

For GPU-based tasks, the cluster is equipped with NVIDIA HGX A100 systems. Each of these systems features 4 GPUs, delivering state-of-the-art performance for GPU-accelerated workloads such as deep learning, scientific simulations, and high-performance data analytics. The GPU resources are distributed across 23 compute nodes, ensuring balanced and efficient handling of tasks requiring parallel computation.

To facilitate high-speed communication between compute nodes, the cluster employs Mellanox 100 Gbps InfiniBand.

Workloads. Based on the workload characteristics of the data center (§2), we selected four representative tasks to construct our workloads: (A) GROMACS [1], representing HPC tasks, (B) ResNet-50 [5] training on ImageNet, representing deep learning training tasks, (C) Baichuan-2 [16], representing LLM tasks. During the evaluation, we submitted a certain number of single or mixed tasks to Slurm, which were then allocated to GPUs for execution.

Metrics. In the evaluation, we used the job completion time as the primary performance metric. Additionally, we also examined GPU utilization as a secondary metric.

5.2 System Performance

We evaluated gPooling in single-GPU and three-GPU cluster configurations, testing GPU virtualization into 2, 4, and 7 vGPUs with non-virtualized GPUs as controls. In both environments, we tested three independent task types and one mixed-task workload comprising all three tasks. For independent tasks, we submitted a batch of 21 identical task instances to the Slurm system in a single submission. For the mixed-task scenario, we simultaneously submitted 21 instances of each of the three distinct task types to Slurm. For the Baichuan-7B task, we only evaluated two configurations: non-virtualized physical GPUs (w/o virtualization) and 2 vGPUs per GPU. This limitation occurred because our GPU virtualization strategy evenly partitioned the GPU memory across vGPUs. Under this design, deploying 4 or 7 Baichuan-7B instances on a single

physical GPU would trigger out-of-memory (OOM) errors. Similarly, in mixed-task experiments, the 4 vGPU and 7 vGPU tests excluded Baichuan-7B and were only tested with the other two task types.

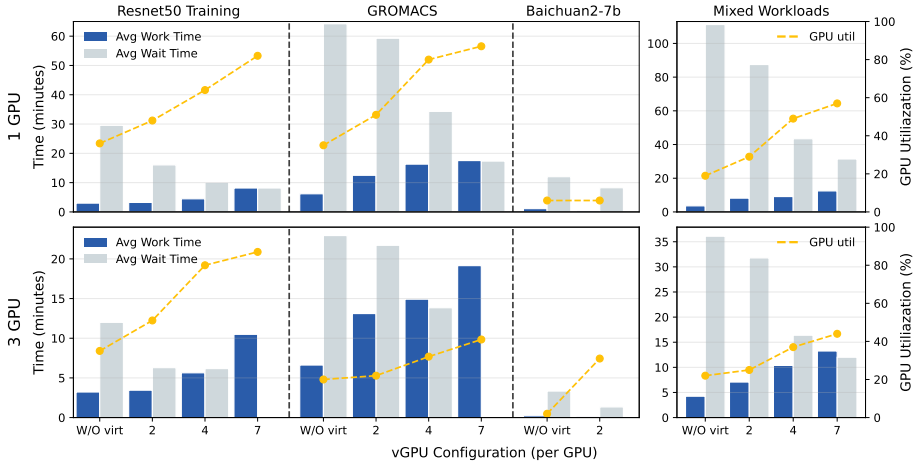

Fig. 6: System Performance

Figure 6 demonstrates the experimental results, comparing the average queuing time, average completion time, and average GPU utilization across different configurations.

Mixed Workloads Under mixed workloads, compared to the non-virtualized baseline, for single-GPU configurations, the average queuing time decreased by 21.33%, 60.94%, and 71.75% in 2 vGPU, 4 vGPU, and 7 vGPU setups respectively. For 3-GPU clusters, the reductions reached 11.97%, 54.60%, and 66.73% under equivalent vGPU configurations. The performance gains primarily stem from significant improvements in average GPU utilization. Under the 7 vGPU configuration, we observed a twofold increase in GPU utilization for single-GPU tests and an equivalent improvement in the 3-GPU experiments. While transitioning from physical GPUs to vGPUs with constrained compute and memory resources inherently increases per-task execution time, our mixed workload experiments demonstrated a 0.66× to 2.47× rise in average task duration across six test groups. However, when holistically evaluating job completion times (i.e., from submission to finish):

Under single-GPU configurations, the 2 vGPU, 4 vGPU, and 7 vGPU setups reduced average job completion times by 16.68%, 54.22%, and 61.78% respectively. In 3-GPU clusters, equivalent vGPU configurations achieved reductions of 3.72%, 33.66%, and 37.40%. A key observation is that gPooling achieves more pronounced improvements in average job completion time under single-GPU

configurations. This disparity stems from the heavier per-GPU task loads in single-node environments, where virtualized GPUs (vGPUs) substantially reduce queuing delays. Notably, while task execution times increase due to resource partitioning, this overhead remains comparable between single-GPU and 3-GPU clusters—the performance divergence primarily originates from queuing efficiency gains under high-load single-GPU scenarios.

Single-Type Workloads In the single-type workload results, we observe that the average queuing time drops to zero in 3-GPU clusters configured with 7 vGPUs per GPU. This phenomenon occurs because the number of submitted jobs (21) exactly matches the total available vGPUs (3 GPUs $\times$ 7 vGPUs = 21), eliminating resource contention and enabling immediate job execution.

Furthermore, when comparing the average task duration between ResNet-50 training and GROMACS under 2 vGPU configurations against physical GPUs, ResNet-50 training exhibits negligible changes in average task duration, whereas GROMACS shows a significant increase. This discrepancy arises because gPooling employs time-slicing-based multiplexing with dynamic adjustments for vGPU compute isolation. Deep learning workloads like ResNet-50 training inherently align with such virtualization due to their fluctuating utilization patterns, which naturally accommodate time-slicing intervals [2]. In contrast, HPC tasks like GROMACS lack natural synchronization points, leading to frequent context-switching overheads and exacerbated performance degradation.

In the Baichuan2-7B evaluation under low token input/output scenarios, we observed reduced GPU utilization and shorter task completion times. Notably, in both single-GPU and 3-GPU configurations, virtual GPUs (vGPUs) allocated 50% memory and 50% compute capacity unexpectedly outperformed physical GPUs. This anomaly originated from memory constraints: the 50% memory limitation compelled the framework to enforce aggressive optimizations, such as kv-cache retention policies and memory reuse strategies, which paradoxically improved performance in low-token workloads.

5.3 Compute Partitioning Efficiency

We then evaluate the compute partitioning capability of gPooling's virtualization component. We use a matrix multiplication from NVIDIA CUDA samples [9], configuring large matrix dimensions to fully saturate the GPU. By observing the GPU utilization of this task under vGPUs with varying compute constraints, we assess whether gPooling can precisely partition compute resources. For this test, we selected a mainstream industry solution[10] as the baseline.

Figure 7 presents the results. We tested multiple vGPU configuration modes (denoted as m%$\times$n, where n slices are created with each allocated m% of the GPU's compute capacity).

[9] https://github.com/NVIDIA/cuda-samples
[10] https://virtaitech.com/

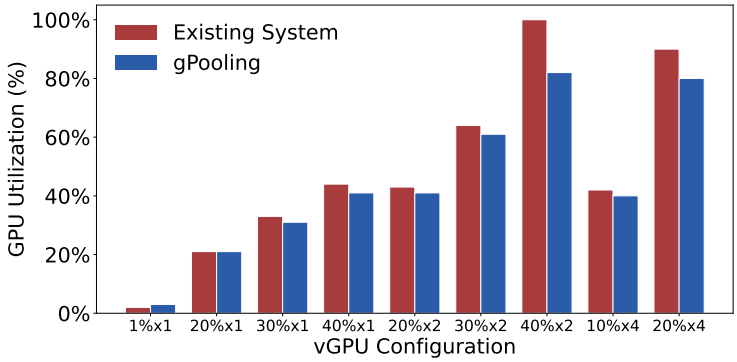

Fig. 7: Compute Partitioning Efficiency: gPooling vs. Exist System

Key observations include: gPooling outperforms existing solutions in compute partitioning stability and precision under 1vGPU, 2vGPU, and 4vGPU configurations. Notably, its compute control capability becomes more robust when the total allocated compute exceeds 80%. In extreme scenarios with 1% compute constraints, gPooling achieves comparable performance to existing solutions.

6 Conclusion

In this paper, we introduce gPooling, a versatile GPU pooling framework designed to optimize GPU resource utilization through low-overhead, high-performance time-division multiplexing based on GPU driver hijacking. By analyzing application characteristics from a university data center, we highlight the distinct differences between these environments and commercial data centers, underscoring the unique challenges faced in academic settings. Our real-world deployment and experimental evaluations demonstrate that gPooling significantly enhances GPU utilization and improves the overall user experience within GPU clusters.

Acknowledgments. This work was supported in part by the National NSF of China (NO. 62141218, 62232012), and Shanghai Key Laboratory of Scalable Computing and Systems. The experiments in this paper were run on the Siyuan-1 supercomputer supported by the HPC Center at Shanghai Jiao Tong University.

References

1. Abraham, M.J., et al.: Gromacs: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. SoftwareX **1-2**, 19–25 (2015). `https://doi.org/10.1016/j.softx.2015.06.001`
2. Bai, Z., et al.: Pipeswitch: Fast pipelined context switching for deep learning applications. In: 14th USENIX OSDI (2020)

3. Bakita, J., Anderson, J.H.: Hardware compute partitioning on NVIDIA gpus. In: 29th IEEE RTAS (2023). https://doi.org/10.1109/RTAS58335.2023.00012
4. Bakita, J., Anderson, J.H.: Demystifying NVIDIA GPU internals to enable reliable GPU management. In: 30th IEEE RTAS (2024). https://doi.org/10.1109/RTAS61025.2024.00031
5. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: IEEE CVPR (2016). https://doi.org/10.1109/CVPR.2016.90
6. Hu, Q., et al.: Characterization and prediction of deep learning workloads in large-scale GPU datacenters. In: ACM SC (2021). https://doi.org/10.1145/3458817.3476223
7. J, J., et al.: Highly accurate protein structure prediction with alphafold. nature (2021), https://www.nature.com/articles/s41586-021-03819-2
8. Jiang, Z., et al.: Megascale: Scaling large language model training to more than 10, 000 gpus. In: 21st USENIX NSDI (2024), https://www.usenix.org/conference/nsdi24/presentation/jiang-ziheng
9. Khandelwal, A., Yun, T., Nayak, N.V., Merullo, J., Bach, S.H., Sun, C., Pavlick, E.: $100k or 100 days: Trade-offs when pre-training with academic resources (2024)
10. Park, H., et al.: A generative artificial intelligence framework based on a molecular diffusion model for the design of metal-organic frameworks for carbon capture. Communications Chemistry (2024). https://doi.org/10.1038/s42004-023-01090-2
11. Stojkovic, J., et al.: Dynamollm: Designing LLM inference clusters for performance and energy efficiency. In: IEEE HPCA (2025). https://doi.org/10.1109/HPCA61900.2025.00102
12. Strati, F., et al.: Orion: Interference-aware, fine-grained GPU sharing for ML applications. In: ACM EuroSys (2024). https://doi.org/10.1145/3627703.3629578
13. Weng, Q., et. al: Beware of fragmentation: Scheduling gpu-sharing workloads with fragmentation gradient descent. In: USENIX ATC (2023), https://www.usenix.org/conference/atc23/presentation/weng
14. Xiao, W., et al.: Antman: Dynamic scaling on GPU clusters for deep learning. In: 14th USENIX OSDI 2020 (2020), https://www.usenix.org/conference/osdi20/presentation/xiao
15. Xu, K., et al.: Design and operation of shared machine learning clusters on campus. In: 30th ACM ASPLOS (2025). https://doi.org/10.1145/3669940.3707266
16. Yang, A., et al.: Baichuan 2: Open large-scale language models. CoRR **abs/2309.10305** (2023). https://doi.org/10.48550/ARXIV.2309.10305
17. Ye, Z., et al.: Deep learning workload scheduling in GPU datacenters: A survey. ACM Comput. Surv. (2024). https://doi.org/10.1145/3638757
18. Yu, P., Chowdhury, M.: Salus: Fine-grained GPU sharing primitives for deep learning applications. CoRR **abs/1902.04610** (2019), http://arxiv.org/abs/1902.04610
19. Zhang, S., et al.: Improving GPU sharing performance through adaptive bubbleless spatial-temporal sharing. In: ACM EuroSys (2025). https://doi.org/10.1145/3689031.3696070
20. Zhang, Y., et al.: Workload consolidation in alibaba clusters: the good, the bad, and the ugly. In: 13th ACM SoCC (2022). https://doi.org/10.1145/3542929.3563465