nvshare: Practical GPU Sharing without Memory Size Constraints

Georgios Alexopoulos
University of Athens, and
National Infrastructures for Research and Technology
Greece
grgalex@ba.uoa.gr

Dimitris Mitropoulos
University of Athens, and
National Infrastructures for Research and Technology
Greece
dimitro@ba.uoa.gr

ABSTRACT

GPUs are essential for accelerating Machine Learning (ML) workloads. A common practice is deploying ML jobs as containers managed by an orchestrator such as Kubernetes. Kubernetes schedules GPU workloads by exclusively assigning a device to a single job, which leads to massive GPU underutilization, especially for interactive development jobs with significant idle periods. Current GPU sharing approaches assign a fraction of GPU memory to each co-located job to avoid memory contention and out-of-memory errors. However, this is impractical, as it requires a priori knowledge of memory usage and does not fully address GPU underutilization. We propose `nvshare`, which transparently enables page faults (i.e., exceptions that are raised when an entity attempts to access a resource) to allow *virtual* GPU memory oversubscription. In this way we permit each application to utilize the entire *physical* GPU memory (Video RAM). To prevent *thrashing* (a situation in which page faults dominate execution time) in a reliable manner, `nvshare` serializes overlapping GPU bursts from different applications. We compared `nvshare` with `KubeShare`, a state-of-the-art GPU sharing solution. Our results indicate that both perform equally well in convential sharing cases where total GPU memory usage fits into VRAM. For memory oversubscription scenarios, which `KubeShare` does not support, `nvshare` outperforms the sequential execution baseline by up to 1.35x. A video of `nvshare` is available at https://www.youtube.com/watch?v=9n-5sc5AICY

CCS CONCEPTS

• **Computer systems organization** → *Cloud computing*; • **Computing methodologies** → **Graphics processors**; • **Software and its engineering** → **Multiprocessing / multiprogramming / multitasking**.

KEYWORDS

Graphics Processing Unit, Resource Sharing, Machine Learning

ACM Reference Format:
Georgios Alexopoulos and Dimitris Mitropoulos. 2023. nvshare: Practical GPU Sharing without Memory Size Constraints. In *Proceedings of 46th International Conference on Software Engineering (ICSE 2024).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE 2024, April 2024, Lisbon, Portugal
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Graphics Processing Units (GPUs) are essential for accelerating Machine Learning (ML) workloads [18, 21] which include throughput-intensive model training, latency-sensitive inference, and interactive development (e.g., as done in Jupyter Notebooks [11]).

GPUs currently offer no way to limit the per-process VRAM consumption, which can lead to Out-of-Memory (OOM) errors when co-locating applications. To avoid this inter-job interference, container orchestrators such as Kubernetes[5] and Slurm [13] opt to schedule GPU workloads by assigning a whole device to a single job in an exclusive fashion. This one-to-one relationship leads to GPU underutilization, especially for workloads with significant idle periods (e.g. during code refactoring in interactive development tasks) and bursts of heavy GPU usage.

There are several GPU sharing solutions that attempt to address the aforementioned challenge. A number of proposed methods ignore the memory contention issue [8, 14–16], warning users of potential fatal OOM errors. Other methods assign a fixed subset of GPU memory to each job [4, 7, 29], requiring prior knowledge of peak memory usage. However, such approaches suffer from GPU underutilization. Finally, there are solutions that permit each job to use the entire VRAM through paging [27]. Nevertheless, such cases apply to specific ML training jobs and suffer from *thrashing*, i.e., excessive page faults.

We present `nvshare`, a GPU sharing mechanism that transparently enables GPU page faults to allow *virtual* GPU memory oversubscription. As a result, every co-located application can utilize the entire *physical* GPU memory (VRAM). In addition, our mechanism serializes overlapping bursts of computations on the GPU to prevent thrashing. Under `nvshare`, each application runs in its own context, with guaranteed fault isolation and security. Finally, it is application-agnostic as it works at the GPU API layer.

To evaluate `nvshare`, we integrate it with Kubernetes and compare it against `KubeShare` [29], a state-of-the-art approach with positive results. Furthermore, we show that even for GPU-intensive workloads whose combined GPU memory allocations exceed VRAM capacity (and which `KubeShare` cannot handle), `nvshare` offers a 1.35x speedup when compared to the standard sequential execution.

We have released our mechanism as open-source software, available at https://github.com/grgalex/nvshare. Notably, it has attracted interest from both research [1, 10] and industrial environments [9].

2 BACKGROUND AND MOTIVATION

CUDA. Compute Unified Device Architecture (CUDA) [2] is a well-established platform and API that enables general-purpose computing on NVIDIA GPUs. CUDA is designed to work with different programming languages such as Python and C++. Through its API, CUDA provides access to the GPU's virtual instruction

Pre-print

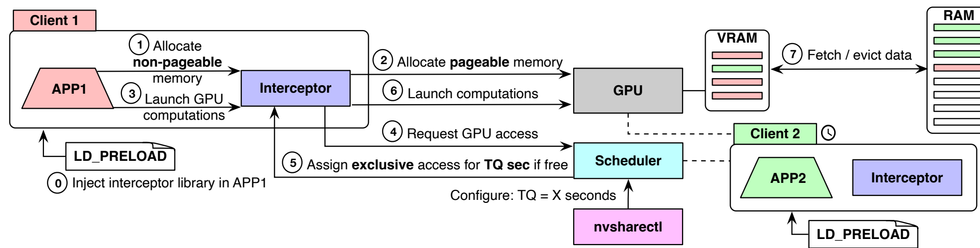

Figure 1: The nvshare architecture. In this snapshot, client 1 is about to evict a page from client 2 who is waiting to use the GPU.

set and computational elements. A CUDA program consists of a mixture of host code, which runs on the CPU and device code, which in turn runs on the GPU.

CUDA Context. A CUDA context is the GPU equivalent of a CPU process. To utilize a GPU, an application creates a context and issues corresponding commands (e.g., memory allocations and computations). Each context has its own set of page tables (virtual address space) and cannot access another context's memory. While multiple contexts exist concurrently on the same GPU, only one can execute work at a given moment. The black-box CUDA driver time-slices contexts running on the same GPU in an undisclosed manner. This time slice is in the order of a few milliseconds.

The standard way of allocating GPU memory is through a method named `cuMemAlloc`. Every byte of virtual memory allocated in this manner, must be backed by a byte of physical memory. An implication of this concept is that the sum of GPU memory allocations across all contexts can at most be equal to the physical GPU memory size. For example, in the case of a 16GB GPU memory, if one process allocates 10GB, then another one can at most allocate 6GB of GPU memory without getting a fatal OOM error. CUDA offers no way to limit the size of per-context memory allocations. Note that it is not possible to know the peak GPU memory use of every application a priori. As a result, orchestrators such as Kubernetes [5] and Slurm [13] address the threat of OOM errors by enforcing a one-to-one job-to-GPU allocation policy and disallow sharing.

Unified Memory. The Unified Memory (UM) [17] hardware / software technology addresses the aforementioned challenge, as it enables page faults for the GPU, using the system's RAM as swap space. To allocate pageable memory, CUDA programs use the `cuMemAllocManaged` method. When a page fault occurs, the UM subsystem (kernel module) fetches the missing page to GPU memory and chooses a victim page from any context to evict to host RAM, employing a Least Recently Used (LRU) replacement policy. UM thus allows GPU memory to be oversubscribed, i.e., the sum of memory allocations across all contexts can exceed the GPU physical VRAM size.

Thrashing. Thrashing is a situation in which time spent handling page faults overwhelms the time spent doing useful computations. When developers oversubscribe GPU memory, they have to make sure to avoid thrashing when the working sets of co-located apps (i.e., the data they are actively using; a subset of their allocations) do not fit in the corresponding GPU VRAM. The constant millisecond-scale context-switching of the black-box CUDA scheduler exacerbates this risk.

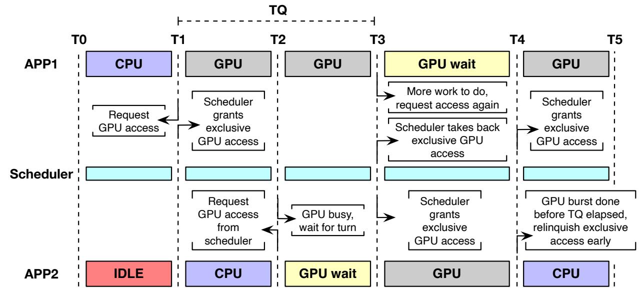

Figure 2: An example execution timeline of applications running under nvshare.

3 APPROACH

3.1 Overview

To allow each co-located application to utilize the entire GPU VRAM, `nvshare` converts every non-pageable `cuMemAlloc` call to its pageable `cuMemAllocManaged` counterpart. To avoid thrashing due to excessive swapping, `nvshare` artificially stretches the exclusive CUDA context time slice from a few milliseconds to tens of seconds, which we call a *Time Quantum* (*TQ*). To this end, we introduce the concept of a per-device *GPU lock*, which we schedule among applications that want to run a computational burst on the system's GPU. Only the application holding the lock can launch work on the GPU, while the CPU parts of co-located applications can overlap freely. When an application obtains the lock, it gradually pages in its data into GPU physical memory and evicts the data of the other co-located applications. In this way, each application can utilize the entire GPU VRAM while also avoiding thrashing.

3.2 System Architecture

Figure 1 depicts the overall architecture of our system which involves three basic components: (1) the *interceptor*, which hooks CUDA calls, (2) the *client*, which requests / releases the GPU lock on behalf of the application and, (3) the *scheduler*, which manages the GPU lock among the co-located applications.

Interceptor. The interceptor library re-exports the subset of functions from the CUDA API that we need to hook. The `LD_PRELOAD` environment variable points to the location of our interceptor library in the execution environment (⓪). This variable instructs the dynamic linker to link our library with the corresponding application before any other library. As a result, when the application makes a CUDA call, it calls our version of the function instead. The interceptor hooks application calls to `cuMemAlloc` (①) and internally invokes `cuMemAllocManaged`(②) instead, forcing the application to use UM. The interceptor also hooks CUDA API

functions (③) which can cause page faults and blocks them until the application obtains the GPU lock (⑥).

Client. The interceptor spawns the client (a set of threads) when the user application makes its first CUDA call. The client communicates with the scheduler to request or release the GPU lock (④). Before releasing the lock it makes sure all previously submitted GPU work is complete. In this manner, it avoids scenarios in which the GPU computations of the previous lock owner overlap with those of the new lock owner and cause unnecessary page faults. Additionally, it releases the GPU lock back to the scheduler early if the application has not been doing any GPU work in the last few seconds. This optimization improves GPU utilization, especially when the TQ is set to a large value, as it ensures that clients hold the GPU lock only as long as they are actively using the GPU.

Scheduler. The scheduler's job is to manage the GPU lock. It has two modes, on and off. When it is enabled (on), it assigns the GPU lock to a single client at a time (⑤) for a configurable time quantum (TQ), handling requests from clients in an first come first served manner. When it is disabled (off), it notifies all clients that they can freely launch GPU work in parallel. Users can configure the scheduler's TQ and toggle it on / off using the `nvsharectl` command-line utility. Note that we have not yet implemented automatic thrashing detection, therefore we must toggle the scheduler on and off manually. The safest approach is to always leave it on, unless we know the workloads fit into VRAM.

3.3 GPU Sharing Scenario

Figure 2 presents an execution timeline scenario that involves two applications running in parallel on the same GPU with `nvshare` enabled. As an example, consider two interactive ML development tasks (Jupyter Notebooks) that also include idle periods, during which the developer refactors their code.

From $T0$ until $T1$, `APP1` preprocesses its data on the CPU. In the meantime, `APP2` is idle, as developers are refactoring their code. At $T1$, `APP1` starts running a notebook cell with GPU computations. Thus, the application requests the GPU lock from the scheduler, and since the GPU is vacant, the scheduler provides it. At $T2$, `APP2` also wants to run GPU computations, but has to wait for the GPU to be free, as `APP1` is using it. At $T3$, TQ elapses. The scheduler retrieves the GPU lock from `APP1` and provides it to `APP2`. Since `APP1` still wants to use the GPU it requests the lock again. At $T4$, `APP2` releases the GPU lock early, because it does not need the whole TQ to finish its GPU burst.

3.4 Implementation

We have implemented `nvshare` in the C programming language (~3000 lines of code). The client and scheduler communicate with a persistent connection over UNIX sockets. In this way we eliminate the need for heartbeat messages. In case of a client crash, the scheduler can recover the GPU lock immediately.

4 EVALUATION

We have integrated `nvshare` into Kubernetes [5]. In this context, we have compared our approach against another GPU sharing mechanism, `KubeShare` [29].

Variant	Batch Size	GPU Memory use	GPU/CPU ratio
`light_balanced`	16	7.2 GB	50/50
`light_intense`	16	7.2 GB	90/10
`heavy_balanced`	64	15.3 GB	50/50
`heavy_intense`	64	15.3 GB	90/10

Table 1: Our evaluation involves four variations of a synthetic ML application, which train Tensorflow's `ResNet152v2` [22] model with different parameters.

Interactive applications, including ML development on Jupyter Notebooks, do not have a finite execution time and users can change their code dynamically. As a result, there is no straightforward way to define corresponding quantitative measurements. We chose to evaluate `nvshare` on non-interactive, ML training workloads with no idle periods, which represent the most computationally dense scenario. In this manner, we can estimate the lower bound of the performance increase that `nvshare` offers.

4.1 Setup

Kubernetes Integration. Integrating `nvshare` into an orchestrator is straightforward. To highlight this feature, we have integrated `nvshare` with Kubernetes [5]. To do so, we implemented a device plugin [3] which advertizes the "*nvshare.com/gpu*" resource type, in line with the official NVIDIA device plugin [6], which in turn advertizes "*nvidia.com/gpu*".

Comparison. `KubeShare` [29] is a promising and well-established GPU sharing mechanism for Kubernetes (167 stars on GitHub). `KubeShare` addresses the GPU memory contention problem by assigning a fixed slice of GPU memory to every co-located application. Note that `KubeShare` does not allow memory oversubscription, thus we can only compare it with `nvshare` on a subset of experiments, where the total memory usage is smaller than the VRAM size. For memory oversubscription scenarios we compare `nvshare` with the only choice that users currently have, which is running the workloads sequentially.

Workloads. Our evaluation suite comprises 4 variants of a synthetic ML application that trains a dog breed image classifier using a `ResNet152v2` [22] model, written in Tensorflow. Table 1 presents the variants and their corresponding characteristics. Our suite contains a mix of CPU and GPU computational parts. The CPU part trains the model for a few steps, while the GPU part trains the model for 5 Epochs. Note that training on the CPU is an order of magnitude slower than on the GPU. Since the total volume of computations is fixed, we can use the total completion time as the measure of performance. The variants differ on (1) GPU memory usage and (2) GPU / CPU compute ratios. The `light` variants use 7 GB of GPU memory, so we can co-locate two instances on the same GPU with KubeShare and compare it against our mechanism. The `heavy` variants use 15 GB of GPU memory, which means that only `nvshare` can run two of them in parallel, and our baseline involves sequential execution. We also vary the GPU / CPU part ratio to examine different GPU contention scenarios. When GPU computations dominate the execution time, the benefit of sharing diminishes, as a single application can saturate the GPU on its own.

Execution Environment. We conducted our experiments on a server machine with a 16-core CPU (Intel Xeon 2.3 GHz), 104 GB

Pre-print

Scenario	light_balanced (s)	light_intense (s)	heavy_balanced (s)	heavy_intense (s)
Baseline (sequential)	**2636**	**1384**	**2766**	**1438**
KubeShare [29]	1724	1078	OOM Error	OOM Error
nvshare (scheduler off)	1772	1128	*11757 (thrashing)*	*11434 (thrashing)*
nvshare (1000)	2053	1361	2043	1380
nvshare (100)	2010	1351	2100	1435
nvshare (60)	2020	1348	2122	1468
nvshare (30)	2017	1366	2170	1521

Table 2: Total Completion Time (TCT) measurements for two copies of each workload. For `nvshare`-related rows, we show Time Quantum (TQ) values in parentheses.

of host RAM and an NVIDIA Tesla P100 GPU with 16 GB of device memory. We used Tensorflow 2.3.0 and Kubernetes 1.25.

4.2 Results

Performance measurements. Table 2 presents our results. The baseline for our measurements is the sequential execution of two copies of each variant. For the `light` variants, `nvshare` with the scheduler disabled offers a comparable speedup to that of `KubeShare`. In `KubeShare`'s case, we must declare the peak memory usage of the workloads before launching them, which is not always the case with real workloads. `nvshare` with the scheduler enabled lags behind, but still offers an up to 1.28x total throughput increase. This is expected, as in these `light` applications there is no thrashing and therefore no need to enforce exclusive GPU access. Note that the default black-box NVIDIA context switcher, which is used in `KubeShare` and `nvshare` without scheduling, handles the GPU workloads in a more efficient manner. Specifically, it switches between the workloads every few milliseconds and better fills in the utilization gaps in the computational units.

`KubeShare` cannot support two copies of `heavy` variants running in parallel, as the combined GPU memory usage of 30 GB far exceeds the VRAM capacity. When running two workloads with `nvshare`'s scheduler disabled the system thrashes, which is reflected by the huge execution times and illustrates the necessity of our scheduler. However, with the anti-thrashing scheduler enabled, we offer an up to 1.35x speedup in the 50/50 case, and a 1.04x speedup even in the 90/10 case, which represents the absolute worst case in terms of GPU computational intensity.

Choosing a TQ for the scheduler. Figure 3 presents the Total Completion Time (TCT) for two `heavy_balanced` instances running in parallel under `nvshare`. The horizontal line shows the TCT for the sequential execution of the workloads and acts as a baseline. For interactive jobs, a smaller TQ is ideal. A large TQ value minimizes TCT, as the GPU changes hands less often leading to a smaller page fault overhead. When the TQ becomes too small (<10 sec), TCT begins to sharply rise, as execution time is dominated by page faults. This is because each application does not hold the GPU long enough to perform meaningful computations after having fetched its data to the GPU. For batch jobs, such as ML training, where our only goal is to minimize TCT, we recommend setting TQ to a large value (e.g., 1000 sec), to let each workload perform its GPU bursts unhindered and minimize page fault overhead.

5 RELATED WORK

For the sake of brevity, we will only discuss GPU sharing solutions that are transparent, i.e., require no source code changes to the application [23, 26], framework [20, 25, 28], or the OS [24].

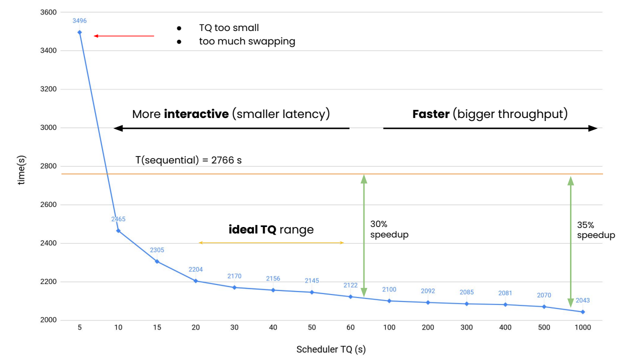

Figure 3: Total completion time of two `heavy_balanced` workloads for varying Time Quantum (TQ) values.

Memory-agnostic. This category ignores memory contention and circumvents the one-to-one job-to-GPU allocation. NVIDIA *Kubernetes time-slicing* [16] uses MPS [8], which funnels multiple applications into the same CUDA context. Thus, they are treated as a single application by CUDA. MPS does not allow GPU memory oversubscription and does not provide any fault isolation. *Slurm Sharding* [14] and *GKE* (Google Kubernetes Engine) *time-sharing* [15] allow multiple jobs to run on the same GPU but suffer from OOM errors, as they do not restrict the per-job memory consumption.

Memory-aware. NVIDIA's *Multi-instance GPU* (*MIG*) [7] is a hardware mechanism available only on some GPU models. It enables presenting a GPU as smaller independent devices, each with its own memory and compute units. The *GPU Sharing Scheduler Extender* [4] in Kubernetes utilizes pre-declared GPU memory requirements to bin pack jobs. Note that it does not enforce these limits at runtime. KubeShare [29] (see Section 4.1), additionally enforces these requirements at runtime, meaning that if a job tries to allocate more memory than its slice, it gets an OOM error. In all the above approaches, the limiting factor is the VRAM size. Users need to be aware of the peak memory usage for their applications beforehand. This requirement, combined with the fact that ML Frameworks [12, 19] overallocate GPU memory, can be impractical. *TGS* [27], is similar to `nvshare`, as it also utilizes UM to enable GPU memory oversubscription. However, it is designed exclusively with deep learning training applications in mind. Specifically, it assumes a steady work submission rate and monitors deviations from that rate to detect thrashing. However, it does not prevent it in a deterministic manner (as `nvshare` does). Thus, in cases of non-steady work submission rates such as interactive ML development it can suffer from performance degradation and system crashes.

6 CONCLUSION AND FUTURE WORK

We introduced `nvshare`, a practical GPU sharing solution which enables each co-located application to utilize the entire GPU memory. Our evaluation indicated that `nvshare` performs equally well to `KubeShare` [29] in conventional sharing cases where total GPU memory usage fits into VRAM. In GPU memory oversubscription cases, which `KubeShare` does not support, `nvshare` significantly outperforms the sequential execution baseline. We plan to extend `nvshare` with a thrashing detection mechanism, which will enable or disable our scheduler automatically.

Pre-print

REFERENCES

[1] CATIE: Centre Aquitain des Technologies de l'Information et Electroniques. https://catie.fr. [Online; accessed 21-October-2023].

[2] CUDA zone. https://developer.nvidia.com/cuda-zone. [Online; accessed 21-October-2023].

[3] Device Plugins | Kubernetes. https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/. [Online; accessed 21-October-2023].

[4] GPU Sharing Scheduler Extender in Kubernetes. https://github.com/AliyunContainerService/gpushare-scheduler-extender. [Online; accessed 22-October-2023].

[5] Kubernetes. https://kubernetes.io/. [Online; accessed 21-October-2023].

[6] NVIDIA device plugin for Kubernetes. https://github.com/NVIDIA/k8s-device-plugin. [Online; accessed 21-October-2023].

[7] Nvidia multi-instance GPU (MIG). https://www.nvidia.com/en-us/technologies/multi-instance-gpu/https://www.nvidia.com/en-us/technologies/multi-instance-gpu. [Online; accessed 22-October-2023].

[8] NVIDIA Multi-process service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf. [Online; accessed 21-October-2023].

[9] nvshare | [Q & A] Intercepting cudaMallocAsync API may also be suitable to this approach? https://github.com/grgalex/nvshare/issues/4#issuecomment-1650245938. [Online; accessed 21-October-2023].

[10] nvshare | Question: usage / configuration per gpu. https://github.com/grgalex/nvshare/issues/10#issue-1906313585. [Online; accessed 21-October-2023].

[11] Project Jupyter. https://jupyter.org. [Online; accessed 21-October-2023].

[12] Pytorch. https://pytorch.org/. [Online; accessed 22-October-2023].

[13] Slurm workload manager. https://slurm.schedmd.com/documentation.html. [Online; accessed 22-October-2023].

[14] Slurm Workload Manager - Generic Resource (GRES) Scheduling. https://slurm.schedmd.com/gres.html#Sharding. [Online; accessed 21-October-2023].

[15] Time-sharing GPUs on GKE. https://cloud.google.com/kubernetes-engine/docs/concepts/timesharing-gpus. [Online; accessed 21-October-2023].

[16] Time-Slicing GPUs in Kubernetes. https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/gpu-sharing.html. [Online; accessed 21-October-2023].

[17] Unified Memory Programming. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd. [Online; accessed 21-October-2023].

[18] Why GPUs are essential for AI and high-performance computing. https://developers.redhat.com/articles/2022/11/21/why-gpus-are-essential-computing. [Online; accessed 21-October-2023].

[19] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[20] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. PipeSwitch: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 499–514, 2020.

[21] Ebubekir BUBER and Banu DIRI. Performance analysis and cpu vs gpu comparison for deep learning. In *2018 6th International Conference on Control Engineering & Information Technology (CEIT)*, pages 1–6, 2018.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[23] Gangmuk Lim, Jeongseob Ahn, Wencong Xiao, Youngjin Kwon, and Myeongjae Jeon. Zico: Efficient GPU memory sharing for concurrent DNN training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 161–175, 2021.

[24] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU concurrency with elastic kernels. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, page 407–418, 2013.

[25] Mosharaf Chowdhury Peifeng Yu. Salus: Fine-grained GPU sharing primitives for deep learning applications. In *Conference on Machine Learning and Systems*, 2020.

[26] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based GPU memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, page 891–905, 2020.

[27] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. Transparent GPU sharing in container clouds for deep learning workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 69–85, 2023.

[28] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548, 2020.

[29] Ting-An Yeh, Hung-Hsin Chen, and Jerry Chou. KubeShare: A framework to manage GPUs as first-class and shared resources in container cloud. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, page 173–184, 2020.

Pre-print