GPEmu: A GPU Emulator for Faster and Cheaper Prototyping and Evaluation of Deep Learning System Research

Meng Wang
University of Chicago
wangm12@uchicago.edu

Gus Waldspurger
University of Chicago
gus@waldspurger.com

Naufal Ananda
Telkom University
naufalrezkyananda@student.telkomuniversity.ac.id

Yuyang Huang
University of Chicago
yuyangh@uchicago.edu

Kemas Wiharja
Telkom University
bagindokemas@telkomuniversity.ac.id

John Bent
LANL
johnbent@gmail.com

Swaminathan Sundararaman
IBM Research
swami@cs.wisc.edu

Vijay Chidambaram
UT Austin
vijayc@utexas.edu

Haryadi S. Gunawi
University of Chicago
haryadi@cs.uchicago.edu

ABSTRACT

Deep learning (DL) system research is often impeded by the limited availability and expensive costs of GPUs. In this paper, we introduce GPEmu, a GPU emulator for faster and cheaper prototyping and evaluation of deep learning system research without using real GPUs. GPEmu comes with four novel features: time emulation, memory emulation, distributed system support, and sharing support. We support over 30 DL models and 6 GPU models, the largest scale to date. We demonstrate the power of GPEmu by successfully reproducing the main results of nine recent publications and easily prototyping three new micro-optimizations.

PVLDB Reference Format:
Meng Wang, Gus Waldspurger, Naufal Ananda, Yuyang Huang, Kemas Wiharja, John Bent, Swaminathan Sundararaman, Vijay Chidambaram, and Haryadi S. Gunawi. **GPEmu: A GPU Emulator for Faster and Cheaper Prototyping and Evaluation of Deep Learning System Research**. PVLDB, 18(6): 1919 - 1932, 2025.
doi:10.14778/3725688.3725716

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/mengwanguc/gpemu.

1 INTRODUCTION

Deep Neural Networks (DNNs) have significantly advanced various machine learning domains, including image recognition, video classification, and natural language processing. This surge in DNN applications has led to a substantial increase in the demand for graphical processing units (GPUs), the preferred hardware for DNN training and inference due to their computational capabilities. However, the high costs of GPUs [3, 6, 18] and their limited availability in public research clouds [8, 40] often result in resource scarcity, causing delays in numerous DNN-related research projects.

For example, the Chameleon research cloud [8] offers only four servers with NVIDIA V100 GPUs, which are highly competitive and often require reservations weeks in advance. While GPU servers are more accessible on commercial clouds, their reservation costs are very high [3, 6, 18], often exceeding budgets allocated for system research. For instance, CloudBank [9, 65] provides up to $5000 per 6-month period for system-focused research, even with machine learning involved. However, this budget is insufficient for large-scale experiments. Reserving four AWS p3.16xlarge instances [3], similar to the setup used in the Synergy [60] paper, would deplete this budget in just 51 hours. Commercial clouds also face limitations, such as GPU unavailability for tens of hours during peak usage [15, 16] and delays of weeks for quota increases [4, 5, 7, 14].

This work is driven by the insight that for a number of DNN research projects, real, physical GPUs are *not required*. For example, some researchers and engineers aim to increase GPU utilization by working on the layers above the GPU in the stack, focusing on aspects such as data loading, preprocessing, job scheduling, and many others [34, 43, 54, 55, 58, 60, 61, 70, 73, 77, 80, 81, 83–85, 87, 89, 90].

For this type of research, rather than the *results* of GPU computations, what matters is the *performance* of the GPU. Thus, we argue that the research community needs a GPU emulator capable of replicating GPU behaviors without the need for physical GPUs. Such an emulator would greatly enhance research prototyping efficiency and also reduce costs, addressing the GPU resource limitations outlined above.

While there are some preliminary emulators available in the industry such as MLPerf storage [23] or ad-hoc emulators created by authors for evaluation purposes [49, 83], these emulators lack crucial components like memory emulation and data preprocessing, as explained more below. Moreover, they are capable of supporting only a small number of configurations and domains.

We present GPEmu, a GPU emulator for faster and cheaper prototyping and evaluation of research on deep learning systems. The design of GPEmu is general, supporting over 30 models and six GPUs. GPEmu is easy to use: a user can run a typical deep learning

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 18, No. 6 ISSN 2150-8097.
doi:10.14778/3725688.3725716

system stack (*e.g.*, PyTorch, TensorFlow) on top of GPEmu. GPEmu works in both single node and distributed setups (such as Kubernetes). GPEmu is focused on deep learning training workloads.

Building GPEmu involved surmounting a number of challenges. First, GPEmu had to emulate a number of different aspects: computation time, preprocessing time, data transfer time, GPU memory consumption to name a few. Focusing on only one or few of these aspects leads to inaccurate emulation results; emulation can report 20% or higher time compared to real runs. Second, GPEmu has to support a number of different deep learning models, different GPUs, and different batch sizes. Finally, GPEmu has to support distributed training, job scheduling and GPU sharing.

Our goals with GPEmu encompass four key facets: (1) We focus on DL training workloads, where GPU compute time is more predictable. (2) Extensive support for emulating a wide range of models, GPUs, and batch sizes, ensuring broad applicability. (3) Users can use GPEmu to effectively reveal DL system bottlenecks during emulations, providing valuable insights into system performance. (4) Users can use GPEmu to quickly show the benefits of new system optimizations across a comprehensive spectrum of research areas. To achieve these goals, we introduce four important features in GPEmu, as explained below.

Time emulation (Section 2.1): We employ a sleep-based method to accurately mimic the time costs associated with GPU-related steps within DL workloads. These steps include GPU-based model computations, host-to-GPU data transfers, and GPU-driven data preprocessing. Based on extensive profiling efforts, we support time emulation for a wide spectrum of DL workloads, covering 30+ DL models (e.g., AlexNet) from widely-used computer vision and speech task catogories, and six GPU models (e.g., P100), with various batch sizes. These capabilities are encapsulated within a Python library, which can be easily integrated into popular DL frameworks such as PyTorch [66].

Memory emulation (Section 2.2): We provide memory emulation for each DL workload. First, we emulate GPU memory usage following the observed patterns. By profiling various DL workloads, we focus on three key metrics: compute peak, model persistent, and preprocessing GPU memory usage. These metrics are essential for assessing model compatibility with specific GPUs and fundamental to GPU sharing emulations. Second, we emulate pinned host memory, originally managed by CUDA for DMA between host and GPU memory. To emulate it, we developed our own pinned memory manager using the `mlock` system call and integrated it into PyTorch, enhancing emulation accuracy.

Distributed system support (Section 2.3): We provide support for emulating DL workloads in distributed setups. First, we emulate multi-GPU single-node training by adjusting time and memory emulations accordingly. Second, we emulate multi-node distributed training based on existing distributed data parallel modules. Third, we support emulating multi-job (cluster) GPU scheduling by introducing an `emuGPU` (emulated GPU) resource type and implementing a custom device plugin for it in Kubernetes. This custom device plugin facilitates the detection and scheduling of `emuGPU` resources within Kubernetes, closely mirroring the scheduling of real GPUs.

GPU sharing support (Section 2.4): We provide emulation support for GPU sharing [45, 76, 81] among multiple DL workloads. To achieve this, we implement an emulation sharing manager. The manager receives emulation requests from DL workloads via RabbitMQ [27] and coordinates their emulation executions to facilitate `emuGPU` sharing. We support GPU sharing emulation for both single-node setups and Kubernetes environments.

Table 1: Features and configurations supported by prior GPU emulators and GPEmu (§2). ***The features' abbreviations read from top to bottom. Labeled by "x", supported features are compared across five categories: (1) GPU-Free (GF); (2) Time emulation: Compute (TC), Data Transfer (TT), and Preprocessing (TP); (3) Memory emulation: GPU Memory Consumption (MC) and Pinned memory (MP); (4) Distributed system support: Multi-GPU training (DG), Multi-Node (DN) training, and Multi-Job scheduling (DJ); (5) GPU Sharing Support (SS). Furthermore, we evaluate the configurations that they support, including the number of DL models (#DL), GPU models (#GPU), and batch sizes (#BS).***

Features	G TTT MM DDD S F CTP CP GNJ S	#DL	#GPU	#BS
Silod [83]	. x.. .. xxx .	8	1	N/A
DLCache [49]	x x..	2	1	5
MLPerf [23]	x x.. .. x.. .	3	2	1
GPEmu	x xxx xx xxx x	36	6	256

To demonstrate the benefits of GPEmu, we first **(a)** *reproduce the main experimental results from various papers across diverse setups and complexities* (Section 3). These experiments range from single-node DL training to multi-node distributed training, extending to 13-node clusters for GPU scheduling emulation. The reproduced results cover analyses and optimizations such as data stall analysis in DL training [61], single-node data loader optimizations [55], disaggregated data preprocessing [34, 73], caching optimizations in distributed training [61, 80], GPU cluster scheduling [54, 60], and GPU sharing with related optimizations [81, 89].

Next, we show **(b)** *how to use GPEmu to introduce and evaluate new micro-optimizations* (Section 4). First, we extend MinIO caching [61] by prioritizing small file caching, maximizing sequential reads while maintaining the same cache hit ratio. Second, we enable the concurrent submission of numerous file read requests to the OS, leveraging its I/O reordering capabilities to reduce disk seek overhead and accelerate data loading for DL training. Finally, we introduce file grouping, which trades minor sampling randomness for improved I/O performance. All optimizations are implemented and evaluated on CPU nodes using GPEmu, demonstrating the emulator's effectiveness in enabling and testing these enhancements.

We discuss the non-goals and future extensions in Section 5 and related work in Section 6.

2 DESIGN FEATURES

To prototype and evaluate without GPUs, people resort to many simulation approaches [35, 44, 47, 51, 54, 60] but impedes end-to-end/full-stack experiments. Others try "emulation" approaches as summarized in **Table 1**. Silod [83] for example, profiled model compute time on the expensive V100 GPU and emulated it on a cluster of cheaper K80 GPUs to evaluate GPU cluster scheduling. MLPerf [23] and DLCache [49] provide a host-side emulator that only performs compute time emulation. As hinted in the table,

these approaches lack vital emulation components and support only limited configurations and experiments.

To the best of our knowledge, GPEmu is the first advanced GPU emulator to offer comprehensive time emulation (compute, data transfer, preprocessing), memory emulation (GPU and pinned memory), distributed system support (multi-GPU training, multi-node training, multi-job scheduling), and GPU sharing capabilities. Additionally, GPEmu supports the largest number of DL models, GPU models, and configurations to date, as shown on the right side of Table 1. Below, we detail GPEmu's design and the emulation challenges it addresses.

2.1 Time Emulation

Time emulation is a feature where the emulator "fakes" the GPU-side operation with a simple sleep timer. While this sounds simple, providing an accurate time emulation requires understanding of the various **steps** of DL workloads: **(1)** Reading samples; **(2)** Preprocessing; **(3)** Data Transfer (from host memory to GPU memory); **(4)** Forward Propagation; **(5)** Backpropagation [42]. It's also worth noting that some frameworks, such as DALI [10], allow for the preprocessing step to be shifted from the CPU to the GPU, altering the sequence of steps 2 and 3.

To emulate DL workloads without actual GPUs, we replace GPU-related steps (steps #3–5, and Step 2 if GPU-based) with simple `sleep(T)` calls, where T represents the projected time for each step. This method emulates the wait time for data transfer between host and GPU memory, as well as GPU preprocessing and model computation. We provide these time emulation methods as a Python package, offering user-friendly APIs like `emuForward(config)` to emulate specific operations. The `config` parameter specifies workload details such as the DL model, batch size, and GPU model, which are essential for accurately predicting operation times.

We support both synchronous (`sync`) and asynchronous (`async`) modes for emulation. In `sync` mode, the sleep-based emulation blocks until the specified duration has elapsed. In `async` mode, the emulation is non-blocking, enabling the simulation of pipelining optimizations commonly used in frameworks like PyTorch [26] and TensorFlow [62]. For PyTorch, `asyncio` is used to implement non-blocking behavior, while in TensorFlow, the `sleep` is integrated into the computation graph. Maintaining operation dependencies is critical in `async` mode to prevent unexpected reordering or pruning during execution. For example, we ensure that forward propagation emulation (Step 4) begins only after the input tensor has been transferred to the GPU (Step 3).

Finally, accurately predicting the emulated sleep time for each step, tailored to the user's specific configuration (model, batch size, GPU type), is essential. The following subsections explain this.

2.1.1 Compute/Propagation Time: We first predict GPU compute time for forward and backward propagations (Steps 4 and 5), a critical aspect of DL workloads. Fortunately, these operations are predictable with minimal variability, as they involve a predefined sequence of tensor operations without conditional branches [45, 77]. As shown in **Figure 1a**, training ResNet50 on a NVIDIA V100 GPU for 1000 batches at batch size 128 exhibits consistent forward and backward durations, except for the first 1-2 batches, which take longer due to initialization.

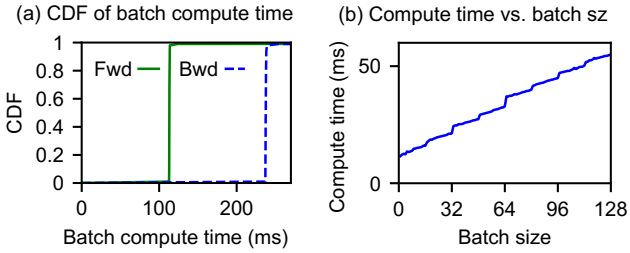

Figure 1: Compute time's pattern (§2.1.1). ***The figures show that (a) compute time is consistent within the same setup, (b) compute time doesn't always linearly correlate with batch size.***

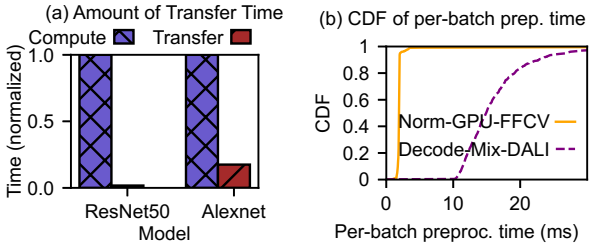

Figure 2: (a) Amount of data transfer time (§2.1.2) and (b) CDF of preprocessing time (§2.1.3).

To predict per-batch GPU compute time, we profile models, GPU types, and batch sizes. We calculate the average compute time, excluding initialization outliers, and store the results in a database. When emulating training for a specific configuration, GPEmu retrieves the profiled compute time for that setting from the database.

Some might assume GPU compute time linearly correlates with batch size, allowing projections from just two profiled batch sizes. However, this holds mainly for computation-heavy models like ResNet101. For computation-light models such as AlexNet, the relationship is more complex and less predictable. As shown in **Figure 1b**, AlexNet on a P100 GPU exhibits a piecewise linear pattern rather than a simple linear correlation. Empirically, we found that models with compute times exceeding 0.2 seconds at batch size 64 on a specific GPU typically exhibit a stable linear correlation with batch size. Profiling these models across all batch sizes is time-intensive, so we use linear projections for quick prototyping. For models below this threshold, we conduct extensive profiling over a broader range of batch sizes.

We provide a profiling tool to support custom settings (§2.6).

2.1.2 Data Transfer Time: We also predict the time required to transfer the input tensor from host to GPU memory. This aspect is often overlooked by other emulation tools [23, 49], yet its importance varies depending on the model and GPU. For instance, as indicated in **Figure 2a**, in larger models like ResNet50 where GPU compute time predominantly dictates performance, the transfer time can be relatively negligible. However, for computation-light models such as AlexNet, the input transfer time becomes more significant, reaching as much as 20% of the GPU compute time.

Like GPU compute time, host-to-GPU data transfer time shows little variability for a fixed-size tensor on a given GPU and typically

scales linearly with data volume. Therefore, we predict the input transfer time based on this linear relationship.

2.1.3 Preprocessing Time: Most DL frameworks perform data preprocessing (Step 2) on CPUs, which does not require specific emulation as preprocessing continues to run on CPUs. However, some libraries, like DALI and FFCV, support offloading preprocessing to GPUs [10, 55], and GPEmu supports this as well. The challenge is that *GPU-side preprocessing time is highly variable*. For instance, **Figure 2b** shows the CDF of two preprocessing operations: normalization and decoding from FFCV and DALI, respectively. Normalization shows smaller variability due to its consistent computational complexity across batches, whereas decoding exhibits significant variability depending on factors such as file format, compression quality, and image complexity. This variability is further complicated in DALI, where batches of random images are processed in parallel, making it computationally prohibitive to profile preprocessing times for all possible image combinations.

To emulate GPU-side preprocessing time for DALI, we take a practical approach: we first profile the time distribution of various operations across different batch sizes and store it in a database. During emulation, we sample times from this distribution instead of relying on a fixed average value. This profiling is dataset-based rather than model-based, as many models share the same preprocessing operations when training on the same dataset. While this method is limited to scenarios where GPU preprocessing is not the primary optimization focus, it is still useful when DALI is used as a baseline for evaluating data loading and CPU-based preprocessing optimizations [55, 61, 73]. For tasks requiring GPU preprocessing optimization, we recommend using real GPUs.

For operations like decoding in DALI, which involve both CPU and GPU processing, the time cost depends on the number of CPUs used. Since DALI's CPU-GPU mix is embedded in its native code and hard to decouple, we emulate both. To account for CPU consumption during decoding, we profile time distributions for various CPU counts and use busy-wait loops to replicate the time cost.

2.2 Memory Emulation

Next, we discuss the need for memory emulation. We divide this into two features: tracking GPU memory consumption and pinning host-side memory.

2.2.1 GPU memory consumption: Not all DL workloads can run on a specific GPU; a job's GPU memory requirement might exceed the GPU's memory capacity. To *precisely* emulate if a DL workload can fit into a GPU's memory capacity, it's crucial to understand three types of memory usage in DL jobs: (1) compute peak, (2) model persistent, and (3) preprocessing memory usages.

Compute peak memory usage means the maximum memory to hold all intermediate results produced by model computation during the propagation phase. Persistent memory usage pertains to storing model parameters over time and plays a role in emulating GPU sharing (§2.4). For these two types of usages, we analyze how a DL model uses memory during propagations. Fortunately, they follow a repeating pattern, as seen in **Figure 3a**. This happens because each batch goes through the same calculations and memory (de)allocations. We also found that both of these usages remain consistent across different GPUs and demonstrate a strong linear

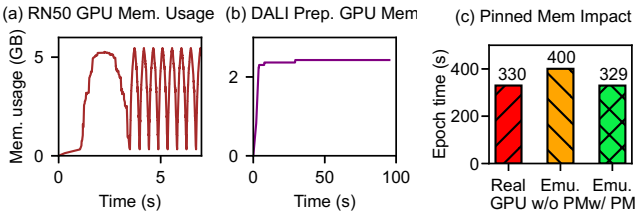

Figure 3: Memory emulation (§2.2). ***The figures show (a) the GPU memory consumption of propagation and (b) GPU-driven preprocessing, as well as (c) the impact of emulating pinned memory.***

correlation with batch size. To mimic this predictable behavior, we profile both usages after processing a few batches across different models. This data serves as the basis for predicting memory usage for configurations that we have not profiled yet.

For GPU-side preprocessing memory usage (e.g., in DALI), we analyze its behavior as shown in **Figure 3b**. Memory consumption steadily grows to a maximum after several batches and then remains constant. This happens because DALI avoids freeing and reallocating memory during training to reduce overhead. Peak memory usage typically occurs when processing a batch with high preprocessing demands. To replicate this behavior, we profile the maximum memory usage after a few epochs to ensure it stabilizes.

2.2.2 Pinned memory: To ensure accurate memory emulation, we need to emulate pinned memory at the host that is managed by CUDA. In real GPU runs, during data transfer from the host to GPU memory, CUDA first allocates a pinned (page-locked) host memory region, copies the host data into it, and then transfers the data from the pinned region to the GPU memory [19].

Failure to emulate pinned memory *can lead to inconsistent performance* compared to real GPU runs. For example, as shown in **Figure 3c**, AlexNet's epoch time differs across various PyTorch setups. Without pinned memory emulation (orange bar), epoch time is up to 20% longer than in real GPU runs (red bar). Our investigation of PyTorch internals revealed that this additional time stems from Python garbage collection (GC) in the main training loop when pageable input data is dereferenced. In real GPU runs, this overhead is avoided as GC occurs in background workers after copying pageable input data to pinned memory. CUDA retains pinned memory during training to avoid costly reallocations, preventing Python GC in the main loop.

For these reasons, we support pinned memory emulation. Since host-side pinned memory is originally managed by CUDA, which does not run on CPU nodes, we developed a custom pinned memory manager. This manager uses the `mlock()` system call to allocate genuine page-locked memory on the host. We mimic CUDA's memory management to allocate the same space and avoid deallocation. As shown by the green bar in **Figure 3c**, our pinned memory emulation effectively resolves the GC stall issue.

2.3 Distributed System Support

Next we discuss how to emulate DL workloads in distributed setups, covering multi-GPU, multi-node, and multi-job (cluster) scheduling.

2.3.1 Multiple-GPU (Single-Node) Training: When emulating multiple GPUs within the same machine, we support PyTorch's

DataParallel (**DP**) module for its simplicity and convenience. DP divides a batch of input data into smaller chunks, distributing each chunk to a different GPU for training. Our time and memory emulation mechanisms can adapt to each GPU's training workload. For example, when training with a batch size of 256 on 4 GPUs, each GPU processes a chunk of 64 samples, leveraging our profiled time and memory data for batch size 64.

An additional step in multi-GPU training is the communication of gradients. After each GPU completes its forward and backward computations, the gradients are gathered across GPUs, averaged on GPU 0, and redistributed. To account for this communication cost in emulation, we profile it for different numbers of GPUs (up to 8). The final per-GPU training time is then computed as the sum of the per-GPU compute time and the GPU communication cost.

We also support emulating Distributed DataParallel (**DDP**) training [57] for multi-GPU workloads. Unlike DP, DDP overlaps inter-GPU communication for gradient reduction with the backward pass. Accurately emulating this overlap is challenging, as gradient reduction is triggered when a bucket of gradients is ready. Precise emulation requires fine-grained profiling of backward pass operations and handling DDP's multi-process nature, adding complexity.

Fortunately, DDP introduces synchronization points at the forward and backward passes [17], simplifying batch-level GPU time emulation. Based on it, we profile total batch time for multi-GPU workloads (up to 8 GPUs), including both compute and overlapped communication costs. To emulate this, we insert two `barrier()` operations at the start and end of each batch, ensuring synchronization across GPUs. Between these barriers, we simulate the profiled batch GPU time using a sleep operation.

2.3.2 Multi-Node Training: PyTorch leverages DDP for multi-node training [57], which we support in our emulation. Similar to single-node DDP emulation, we insert barriers in each batch to enforce synchronization across all nodes. Without these barriers, each node would run the emulation independently.

PyTorch overlaps gradient reduction with the backward pass, complicating accurate emulation of inter-node communication costs. Capturing these costs requires fine-grained per-operation profiling, which we leave for future work. Instead, like single-node DDP, we profile batch-level GPU time for multi-node setups (up to 4 nodes), including compute and overlapped communication costs, and use this for emulation. For setups with more than 4 nodes, we reuse the 4-node profiled time.

We acknowledge that this approach does not precisely replicate network communication costs, especially at larger scales. However, it enables functional multi-node training emulation and is effective for evaluations less sensitive to gradient communication, as is shown in Section 3.4.

2.3.3 Multi-Job (Cluster) Scheduling: We support emulating multi-job (cluster) GPU scheduling for both custom schedulers and Kubernetes environments, each presenting unique challenges.

Custom schedulers like Synergy [60] rely on predefined cluster configuration files containing detailed resource information for the GPU cluster. To enable GPU scheduling emulation, we provide emulated GPU availability data within these configuration files. This allows the scheduler to continue using its original scheduling algorithm to allocate resources for DL jobs. Each DL job runs within a container integrated with GPEmu, which performs the emulation based on the allocated GPU configurations.

Kubernetes scheduling operates differently by discovering the availability of custom resources, such as GPUs, through the custom device plugin [11]. To accommodate this, we introduced a new resource type, "`emuGPU`" (emulated GPU), and developed a custom device plugin specifically for it. Each `emuGPU` is represented as a file on the worker node. The `emuGPU` device plugin runs on each worker node, identifies these files, and reports their availability to the Kubernetes scheduler. Based on this data, the Kubernetes scheduler allocates `emuGPUs` using the same algorithm it uses for real GPUs. Once a job is scheduled, the device plugin receives the allocation decision and mounts the corresponding `emuGPU` files to the job's container. The job then accesses the emulated GPU information from these files to emulate DL workloads.

We support multi-GPU, multi-node setups, and multi-job scheduling in the emulation. A multi-GPU training job is a GPEmu-integrated container requesting multiple GPUs for scheduling. Similarly, a multi-node training job is a Kubernetes job deploying multiple emulation container replicas.

2.4 Sharing Support

DL workloads often need to time-share the GPU, a technique commonly used in both academia [45, 76, 81] and industry [30, 31]. They typically time-slice the GPU at the granularity of a batch or an inference request. Hence, we build time-sharing support in GPEmu for both single-node setups and Kubernetes environments.

We begin with single-node configurations, which present two key challenges. First, we must coordinate the time emulation of multiple DL applications sharing the GPU to ensure only one application uses the GPU at a time. Second, we must emulate GPU memory usage during sharing. If the combined memory usage of model-persistent memory from waiting applications and compute-peak memory from the running application exceeds the GPU's memory capacity, the application will crash.

Based on these considerations, we've developed an *emulation sharing manager*. DL applications first register with their model persistent memory usage. Registered applications seeking GPU access send time emulation requests to the manager via RabbitMQ [27]. When dealing with a shared GPU, the manager handles one request at a time, sleeping for the requested duration and returning a completion response. The request is rejected if the combined memory usage exceeds the GPU memory capacity.

For Kubernetes setups, to mimic real GPU time sharing on Kubernetes [31], we create multiple replica files for each `emuGPU`. This allows our device plugin to allocate them to multiple DL job containers. Additionally, we deploy a RabbitMQ broker and a sharing manager for each node as Kubernetes daemonsets, facilitating communication and emulating GPU time-sharing management.

2.5 Applications and Use Cases

With its features, GPEmu can replace real GPUs in a wide range of scenarios. Below, we present examples of currently supported use cases. Non-goals and future work are discussed in Section 5.

Identifying bottlenecks in DL training: While DL training is compute-intensive, the true bottlenecks often lie in other layers,

such as data loading in the storage layer or preprocessing in the CPU layer [34, 43, 55, 61, 73, 85, 87]. GPEmu helps identify these bottlenecks for specific workloads and resources, enabling better resource allocation to minimize data stalls.

Evaluating Data Loader Optimizations: Many studies optimize data loaders in DL training [48, 55, 61, 79]. GPEmu enables the evaluation of these optimizations across workloads and configurations.

Evaluating Preprocessing Optimizations: Data preprocessing, a common CPU-intensive bottleneck in DL training, has been tackled with techniques such as disaggregation, offloading, and reordering [34, 43, 73, 75, 85–87]. GPEmu enables users to evaluate these techniques and analyze their impact on training performance.

Distributed Training Optimizations: Multi-node distributed training presents unique system challenges. For example, distributed caching optimizations seek to minimize data stalls during training [58, 61, 70, 80, 90]. GPEmu enables prototyping and evaluating such optimizations with its distributed training emulation capabilities.

Evaluating GPU Cluster Scheduling and Sharing: Numerous techniques have been proposed to improve GPU scheduling in clusters [54, 60, 63, 83, 84, 89]. GPEmu provides scheduling and GPU sharing support, enabling users to evaluate these policies under realistic scenarios.

2.6 Framework for Unification and Extensibility

We have unified GPEmu's diverse features into a modular framework designed to support various use cases and future extensibility. The framework consists of the following components:

Profiler: GPEmu provides a modular, user-friendly profiler to measure key metrics such as model compute time, CPU-GPU data transfer time, and GPU memory consumption under user-specified configurations. We have pre-profiled over 30 models on 6 different GPUs across various batch sizes to build a comprehensive dataset.

Time and Memory Emulation Modules: GPEmu includes modular components for various types of time and memory emulation. These modules can be easily customized by users to integrate custom time distributions or values from their profiling data.

Distributed Training Emulation Modules: GPEmu supports distributed training emulation with modular components that include synchronization mechanisms and emulation considering inter-GPU communication cost. These modules are designed to replicate realistic distributed training environments.

Container-Based Benchmarking: GPEmu offers a container that wraps its emulation modules and example training workloads. Users can run these containers to benchmark configurations and identify system bottlenecks. Leveraging PyTorch's modular data loader support, users can also evaluate custom data loading and preprocessing mechanisms for single- and multi-GPU/node setups.

Emulated GPU Scheduling Framework: Built on Kubernetes, GPEmu enables users to create emulated GPU devices on host machines and leverage our custom device plugin and sharing managers to support GPU scheduling and sharing. Users can easily launch emulation jobs on Kubernetes using provided custom containers. Since the framework integrates with Kubernetes's support for custom schedulers, users can evaluate their own GPU scheduling algorithms with minimal effort, requiring just a one-line change in the scheduled resource type (GPU → emuGPU).

Table 2: Implementation efforts (§2.5). ***The left table shows the LOCs for implementing GPEmu. The right table shows the LOCs for re-implementing work from existing papers and for implementing our new micro-optimizations.***

(a)	LOC
Emulation lib	916
K8s device plugin	488
Profiler	1482
PyTorch integration	79
TF integration	20
DALI integration	46
Total	3031

(b) Reimpl.	LOC
CoorDL	2756
LADL	96
Muri	298
Micro-Opt.	
SFF	45
Async Batch	1171
File grouping	695
Total	5061

Extensibility: GPEmu already supports many popular models and GPUs, and extending its capabilities to additional models and GPU architectures is straightforward. Using the profiler, users can gather data for new configurations and add it to GPEmu's library for future use, benefiting the broader community.

GPEmu's modular implementation also allows users to integrate their own time measurement techniques, such as compute time modeling [36, 37], or external profiling data for scenarios like PCIe link sharing. By supporting custom time distributions and profiling data, GPEmu is highly adaptable, encouraging collaboration within the community to expand its coverage and capabilities.

2.7 Implementation Efforts

Our entire effort is quantified in **Table 2a**, a total of **3031 LOC** for the GPEmu implementation, comprising an all-in-one emulation library (in Python), a Kubernetes device plugin, a Profiler, and 20-80 LOC "hooks," for integration into various platforms such as PyTorch, TensorFlow, and DALI.

Furthermore, for showing many case studies in the next two sections, we wrote another **5061 LOC** (**Table 2b**) for re-implementing existing work from scratch (either because they are not available or not fully functional) and for adding new micro-optimizations. For example, we reimplemented CoorDL (§3.4), LADL (§3.4), and Muri (§3.6). We also wrote three micro-optimizations: small-file first (SFF) caching policy (§4.1), asynchronous batch reading (§4.2), and random-class file grouping (§4.3). We have open-sourced our code on Github (https://github.com/mengwanguc/gpemu).

3 CASE STUDIES OF SUPPORTED RESEARCH

To demonstrate GPEmu's versatility and capabilities, we reproduced experiments from nine papers [34, 54, 55, 60, 61, 61, 73, 80, 81, 89], each utilizing different GPEmu features. **Table 3** summarizes the GPEmu design features used (marked with "x") in each reproduction. These experiments range in complexity from single-node training to multi-node distributed training and even a 13-node cluster for GPU scheduling emulation. The papers span various analyses and optimization techniques relevant to deep learning systems, covering the use cases described in Section 2.5.

The advantage of GPEmu over existing emulators can be seen by joining **Table 1** with **Table 3**. If an emulator in **Table 1** lacks features required by a paper in **Table 3**, it cannot evaluate that paper.

Table 3: GPEmu features for paper reproductions (§3). ***The features' short names are identical to the names in Table 1.***

	T C	T T	T P	M C	M P	D G	D N	D J	S S
§3.1 DataStall [61], VLDB '21	x	x	.	x	x	x	x	.	.
§3.2 TF-DS [34], SoCC '23	x	x	.	x	.	x	.	.	.
§3.2 FastFlow [73], VLDB '23	x	x	.	x	.	.	.	.	.
§3.3 FFCV [55], CVPR '23	x	x	x	x	x	.	.	.	.
§3.4 LADL [80], HiPC '19	x	x	.	x	x	x	x	.	.
§3.5 Synergy [60], OSDI '22	x	x	.	x	x	x	.	x	.
§3.5 Allox [54], EuroSys '20	x	x	.	x	x	.	.	x	.
§3.6 Salus [81], MLSys '20	x	x	.	x	x	.	.	.	x
§3.6 Muri [89], SIGCOMM '22	x	x	.	x	x	.	.	.	x

For instance, MLPerf [23] does not support the multi-job scheduling (DJ) feature and therefore cannot evaluate the Synergy [60] and Allox [54] papers listed in **Table 3**.

A primary use case of GPEmu is to enable faster prototyping and evaluation of systems research in deep learning, especially given the scarcity of GPU resources. Our evaluation validated GPEmu 's ability to replicate **patterns** observed in the original papers by comparing our results with their figures. We also evaluated GPEmu 's accuracy by comparing its results to actual GPU runs. Our findings show that GPEmu can replace real GPUs in many system research scenarios, with its non-goals and limitations discussed in Section 5.

In the following subsections, we illustrate how GPEmu can support different types of deep learning system research.

3.1 Data Stall Analysis

A common use case for GPEmu is analyzing system behaviors under varying DL training workloads. Here, we reproduce the **DataStall**$_{\text{VLDB22}}$ [61] paper ("DS" for short). The DS paper studied the impact of input data pipelines on training durations for popular deep learning models. *It found that data stalls—time spent waiting for data fetching and CPU preprocessing—accounted for most training time in many cases.* We show how GPEmu can reproduce this conclusion without requiring real GPUs.

The first step in the DS paper measured fetch stall percentage (time spent waiting for data fetching divided by total training time) when only 35% of data could be cached in memory. In **Figure 4a**, we reproduced these measurements for six models using GPEmu, with two key observations. First, *the pattern is consistent.* Fetch stalls are common, with varying percentages (y-axis) across models. AlexNet, with lightweight computations, spends more time waiting for data, resulting in the highest fetch stall. In contrast, Vgg11, with heavier computations, performs better as GPU computation via prefetching hides most data fetch time. Second, *emulation values differ from the original paper but align with our GPU runs.* These differences stem from setup variations, such as SSD models, page cache fluctuations during training (due to PyTorch memory use), and different data loaders (PyTorch default vs. DALI). Nonetheless, emulation results match our GPU runs, demonstrating GPEmu's accuracy.

The DS paper next assessed how cached data amounts affect training time. In **Figure 4b**, our GPEmu reproduction shows that lower cache percentages increase training times due to more fetch stalls, following a pattern similar to DS paper's [61, Fig.4a]. The

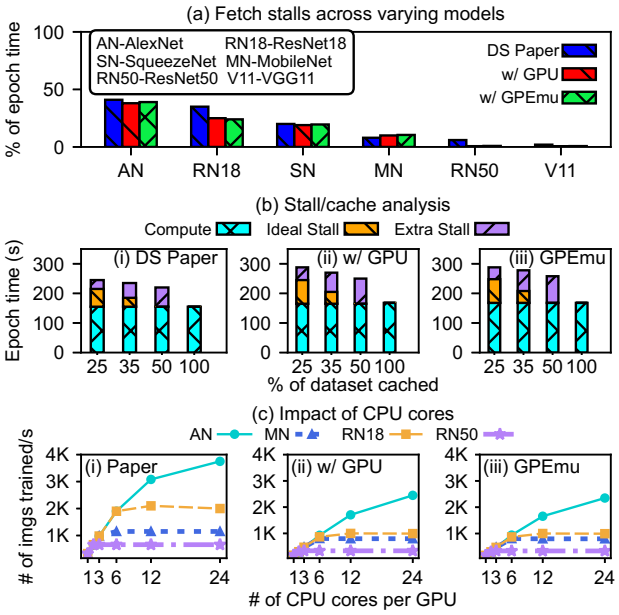

Figure 4: Data stall analysis with GPEmu (§3.1).

figure also highlights the inefficiency of the OS page cache in DL training due to thrashing. For instance, with 25% data cached, the ideal cache hit rate suggests a fetch stall of 248 seconds per epoch, yet the OS page cache can lead to up to 290 seconds. Our emulation aligns with GPU runs and shows patterns similar to the DS paper, though exact values differ due to setup differences.

Finally, the DS paper evaluated the relationship between CPUs per GPU and training speed in images per second [61, Fig. 5]. In **Figure 4c**, we reproduced this using GPEmu for four models. The pattern matches the DS paper, with each model showing distinct CPU requirements per GPU to minimize prep stalls (*i.e.*, delays from CPU data preprocessing). For example, ResNet50 (purple line), a computation-heavy model, requires 3 CPUs per GPU to eliminate prep stalls, while AlexNet (cyan line), with lighter computations, needs 24 CPUs per GPU. The emulated training speed closely matches our GPU runs but differs from the original paper due to hardware and software variations. We think these differences are acceptable as the patterns align.

3.2 Preprocessing Disaggregation

To address data stalls, lots of research has focused on optimizing the data preprocessing stage, often a bottleneck in DL training in various scenarios. A common approach is preprocessing disaggregation [34, 43, 73, 85, 87]. In this subsection, we reproduce two studies using GPEmu: **tf.data service**$_{\text{SoCC23}}$ [34] and **FastFlow**$_{\text{VLDB23}}$ [73].

The first system, tf.data service (TF-DS) [34], is a data preprocessing disaggregation framework built on TensorFlow's tf.data module. Unlike traditional methods relying on local CPUs, which often cannot keep up with GPU demands, TF-DS offloads preprocessing to remote worker CPUs, enabling scalable preprocessing by adding more workers.

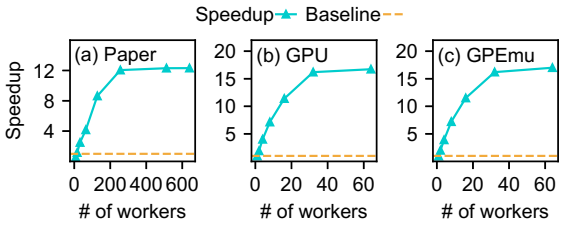

Figure 5: TF-DS [34] training speedup with GPEmu (§3.2).

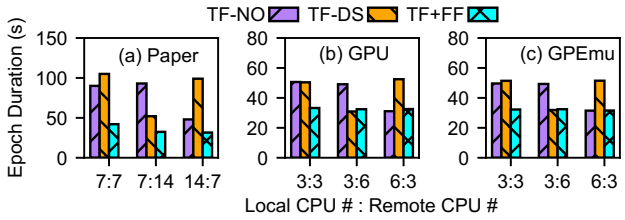

Figure 6: FastFlow's [73] benefits with GPEmu (§3.2).

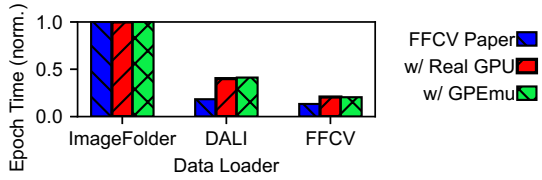

Figure 7: FFCV's [55] benefits shown with GPEmu (§3.3).

We reproduced the results from the TF-DS paper using GPEmu. Our experiments emulated GAN model training on the CUB-200-2011 dataset with a single training client (4 CPUs) and 8 V100 GPUs in TensorFlow integrated with GPEmu. The baseline used local preprocessing on the client machine. We then offloaded preprocessing to a variable number of remote workers, each with 4 CPUs. Training time speedups relative to the baseline are shown in **Figure 5**.

We observe two key points from this figure. First, the speedup increases with more workers, demonstrating TF-DS's effectiveness. Second, beyond 32 workers, adding more yields minimal training time improvements, as preprocessing becomes fast enough and the bottleneck shifts to (emulated) GPU compute. These findings align with those in the TF-DS paper [34, Fig. 9], though we use a different configuration due to the anonymization of their model.

Next, we reproduce FastFlow [73], built atop TF-DS. FastFlow improves preprocessing disaggregation by utilizing both local and remote CPUs. It dynamically splits the preprocessing pipeline between them based on performance metrics.

We obtained the FastFlow source code from their GitHub [13] and reproduced experiments for the Transformer ASR workload using GPEmu. Our setups varied local-to-remote CPU ratios (3:3, 3:6, 6:3). **Figures 6** compare epoch times across three policies: TF-NO (no preprocessing disaggregation), TF-DS, and TF+FF (FastFlow). FastFlow consistently outperforms the others in all scenarios. TF-NO struggles with limited local CPUs, while TF-DS falls behind with few remote CPUs. Our emulation results match our GPU runs and follow the patterns reported in the FastFlow paper [73, Fig. 6].

3.3 Data Loader Optimization

Another approach to reducing data stall is optimizing the data loading phase in DL training. For example, **FFCV**$_{\text{CVPR23}}$ [55] introduces techniques like efficient file storage formats, optimized process-level caching, and just-in-time compiled data preprocessing. Here we demonstrate FFCV's benefits using GPEmu.

In **Figure 7**, we show results from three setups: a real RTX-6000 GPU (red bars), the original FFCV paper (blue bars), and GPEmu (green bars). Our emulation *demonstrates FFCV's performance improvement*, cutting training time by 80% compared to PyTorch's default ImageFolder data loader (leftmost vs. rightmost green bars). The emulation closely matches real GPU results and aligns with the FFCV paper. The y-axis is normalized to the maximum training time per configuration, as we used a smaller dataset for faster research. GPEmu highlights its value for rapid prototyping.

3.4 Distributed Training Optimization

In previous subsections, we used GPEmu to reproduce single-node training analyses and optimizations. Here, we demonstrate how *GPEmu can also reproduce distributed training optimizations*, such as distributed caching, which reduces data stalls in distributed training [58, 61, 70, 80, 90]. We reproduce two papers: CoorDL from the DataStall$_{\text{VLDB21}}$ ("DS") paper [61], and the locality-aware data loading ("LADL") paper from HiPC 2019 [80].

Let's start with **CoorDL** [61], which introduces partitioned caching. In this approach, different compute nodes cache distinct parts of the dataset and coordinate to speed up data loading in distributed training. With partitioned caching, a local cache miss leads to fetching the missing data from remote caches on other nodes via network communication. CoorDL applies this technique to their MinIO caching algorithm, demonstrating significant advantages.

Now, we show using GPEmu to assess CoorDL's benefits. Initially, we tried to obtain CoorDL's source code from its GitHub repository [22], but faced challenges in compiling due to dependency issues with an older version of DALI. As a result, we *re-implemented* CoorDL's MinIO and partitioned caching from scratch in PyTorch.

To reproduce CoorDL's results, we used GPEmu for distributed training on compute nodes, each emulating 8 GPUs, with local cache percentages of 65% and 40%. **Figure 8** shows normalized training speedups: orange bars represent the baseline, blue bars show CoorDL results from the DS paper, and green bars indicate CoorDL with GPEmu. The DS paper shows CoorDL's speedups increasing with more nodes due to partitioned caching. Our emulation reproduces this trend, though values differ due to setup variations. For example, with 65% local cache and 1 node, CoorDL with GPEmu achieves a 3x speedup, primarily from MinIO caching, rising to 6x with 2 nodes as combined cache covers the full dataset.

Next, let's move to "**locality-aware data loading**" (or "**LADL**") [80]. Distributed caching techniques like CoorDL offer impressive gains with few nodes and adequate bandwidth but face scalability challenges in large-scale distributed training. These challenges are mainly due to increased network traffic, potentially causing bandwidth bottlenecks. LADL addresses this by altering the sampling algorithm to prefer locally cached samples, reducing data fetching

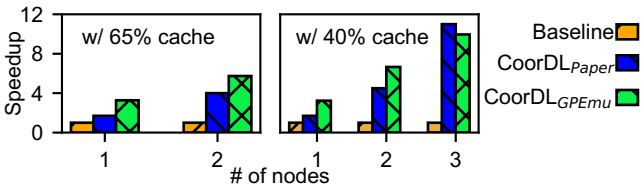

Figure 8: CoorDL's benefits with GPEmu (§3.4).

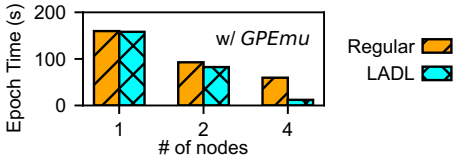

Figure 9: LADL's [80] benefits shown with GPEmu (§3.4).

from remote cache. Though this slightly reduces sampling randomness, it significantly decreases network traffic, thus enhancing scalability for distributed training.

Here we use GPEmu to showcase LADL's advantages. With the original source code unavailable and no response to our request, we *re-implemented* it in PyTorch based on our earlier CoorDL implementation. While the original paper used a 16-128 node cluster to show distributed cache scalability challenges, our resources are more limited. However, as GPEmu is designed for rapid prototyping, we emulated results on a 4-node cluster. To mimic network bandwidth bottlenecks typical in large-scale training, we limited our cluster's network bandwidth to 3Gbps.

The emulation results are shown in **Figure 9**, with orange bars representing regular distributed cache and cyan bars for LADL. The results demonstrate LADL's improvement in distributed training scalability. For example, in 4-node training with all data cached (locally or remotely), regular distributed cache requires 60 seconds per epoch due to network bandwidth limitations. In contrast, LADL reduces the epoch time to just 12 seconds by lowering network bandwidth demands. These findings align with the original paper.

3.5 GPU Scheduling

Next, we explore using GPEmu in GPU scheduling. Numerous techniques have been proposed for enhancing GPU scheduling in clusters [54, 60, 63, 78, 83, 84]. We reproduce two: **Synergy**$_{OSDI22}$ [60] and **Allox**$_{Eurosys20}$ [54]. Synergy is selected for its custom scheduler's easy compatibility with GPEmu, and we also reproduce Allox to demonstrate GPEmu's ability to support Kubernetes scheduling.

We first reproduce some **Synergy** experiments [60]. Synergy is a scheduler that optimistically profiles each job's resource needs and allocates GPUs, CPUs, and memory accordingly. This method effectively prevents data stalls and maximizes GPU utilization.

To reproduce the Synergy paper, we obtained its source code from their GitHub repository [29] and integrated it with GPEmu, using GPEmu for resource demand profiling and GPU scheduling emulation. Synergy was designed for jobs that utilize DNN-aware cache systems like MinIO [61]. We opted for our implementations of MinIO instead of the original, as it faced compilation issues due to dependency conflicts.

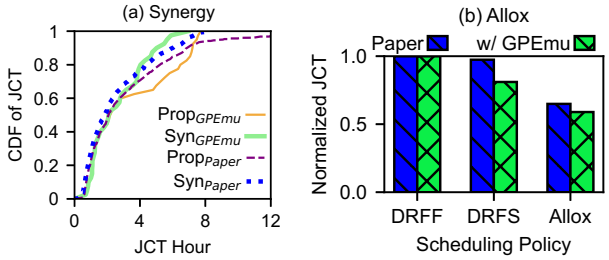

Figure 10: (a) Synergy [60] and (b) Allox's [54] JCT reduction with GPEmu.

We conducted experiments scheduling 40 DL jobs on a 4-node cluster, with each node equipped with 24 CPUs, 100GB DRAM, and 8 emulated GPUs. Our evaluation compared Synergy with GPU-proportional resource allocation using Least Attained Service (LAS) policies. **Figure 10a** shows the CDF of job completion time (JCT) for both policies, with results from the original paper and GPEmu. Consistent with the original paper, our emulation demonstrates Synergy's effectiveness in reducing JCT. For example, Synergy reduces average JCT by 22% and 95th percentile JCT by 25% (solid orange vs. green lines).

Next, we reproduce **Allox** [54] to showcase our Kubernetes scheduling support. Allox is designed for DL jobs with interchangeable resource configurations, such as CPU versus GPU. It strategically schedules jobs across various compute resources, selecting the optimal configuration for each while maintaining fairness.

We acquired the Allox source code from GitHub [2] and integrated it with GPEmu using our Kubernetes support. Our experiments used a cluster with one master node, four `emuGPU` worker nodes, and eight CPU worker nodes. We scheduled 40 PyTorch jobs across four users, with GPU vs. CPU worker speedups ranging from 1.3 to 8.2. **Figure 10b** shows the normalized average JCT for three scheduling policies: Dominant Resource Fairness with First Come First Serve (DRFF), Dominant Resource Fairness with Shortest Job First (DRFS), and Allox. Consistent with the original paper, our emulation demonstrates Allox's effectiveness, reducing average JCT by 42% compared to DRFF (leftmost vs. rightmost green bar).

3.6 GPU Sharing

Finally, we reproduce papers on GPU sharing. There's been considerable research into enabling GPU sharing among DL workloads and optimizing GPU job scheduling based on this sharing [45, 77, 78, 81, 89]. We focus on two notable studies: **Salus**$_{MLSys20}$ [81] and **Muri**$_{SIGCOMM22}$ [89]. Salus is chosen for its focus on enabling fine-grained GPU sharing in DL applications, and Muri for its insights on GPU sharing's impact on GPU job scheduling.

We begin with reproducing **Salus [81]**, which offers an efficient execution service for fine-grained GPU sharing through iteration-level computation scheduling, thereby enabling rapid job switching between different DL applications.

Using our GPU sharing emulation, *we successfully reproduced iteration-level job switching as in Salus.* We integrated GPEmu into PyTorch and emulated [81, Fig. 9] by running three ResNet50 training jobs on a single machine, fairly sharing an emulated P100 GPU.

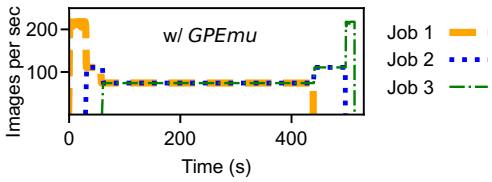

Figure 11: Salus's [81] GPU sharing with GPEmu (§3.6).

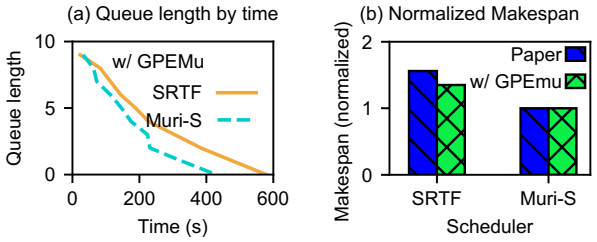

Figure 12: Muri's benefits shown using GPEmu (§3.6).

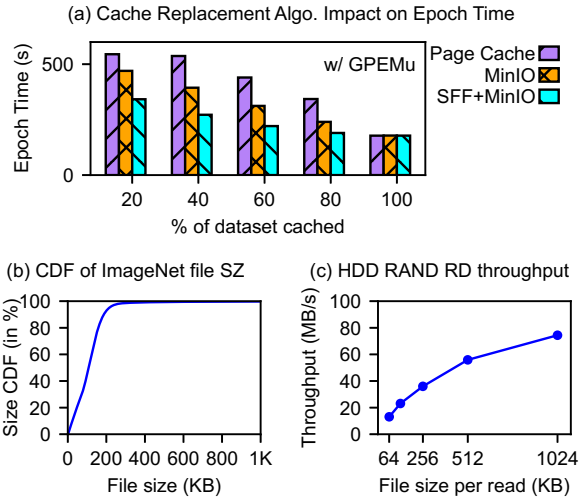

Figure 13: SFF's benefit with GPEmu (§4.1).

Figure 11 depicts the emulated training throughput. Initially, Job 1 achieves 220 images/sec, fully utilizing the GPU. At 30s, Job 2 starts, halving throughput (dashed blue line). At 60s, Job 3 starts, reducing throughput to a third. As jobs finish in reverse order, throughput gradually restores. The pattern aligns with [81, Fig. 9].

Next, we reproduce **Muri [89]**, a DL job scheduler for multi-resource clusters using GPU time sharing. Muri interleaves jobs bottlenecked by different resources (e.g., GPU, CPU, storage, network) and groups them on the same machine to optimize utilization.

To reproduce Muri, we obtained its code from GitHub [24]. However, the testbed deployment code was mostly pseudocode, and the original industry implementation was unavailable. Thus, *we developed a simplified version of Muri*, focusing on single-machine scheduling and CPU-GPU interleaving for fast reproduction. GPU interleaving was implemented via our GPU sharing emulation.

In our experiments, we used Muri with GPEmu to schedule 10 emulated training jobs, ranging from CPU-bound models like AlexNet to GPU-bound models like ResNet50. We compared two algorithms: SRTF and Muri-S (Muri with SRTF). **Figure 12a** shows job queue length changes over time for each policy. Muri outperforms SRTF by reducing queue length faster, running more jobs concurrently, and utilizing GPUs more efficiently, consistent with [89, Fig. 8]. **Figure 12b** presents normalized makespan, showing Muri's 1.35x speedup over SRTF. These results validate Muri's effectiveness and align with the original paper.

4 MICRO-OPTIMIZATIONS

Besides reproducing existing work, we showcase the power of GPEmu by prototyping and evaluating new micro-optimizations without using real GPUs. Below, we present several storage-stack optimizations that can improve DL training epoch time, such as small-file first (SFF) caching policy (§4.1), asynchronous batched data (§4.2), and random-class file grouping (§4.3).

4.1 Cache Small Files

In this subsection, we explore how tailoring caching algorithms to the characteristics of DL training datasets and HDDs impacts performance, demonstrating the benefits with GPEmu. Inspired by the "MinIO" algorithm proposed in the DataStall paper [61], we explore its design. MinIO leverages DL training's unique data access pattern: data is read in epochs, with each epoch accessing every item exactly once in random order. Thus, if X% of data is cached, the maximum attainable cache hit rate is also X%. To address this, MinIO avoids evictions once the cache is filled, mitigating thrashing caused by premature evictions in previous policies.

We implemented MinIO in PyTorch and evaluated it using GPEmu. **Figure 13a** demonstrates MinIO's benefits, with purple bars showing OS page cache performance and orange bars depicting MinIO results. At the same cache percentage, MinIO consistently outperforms the OS page cache. For instance, with a 40% cache size, MinIO achieves an epoch time of 394 seconds compared to 537 seconds for the OS page cache.

However, we found *a limitation in MinIO: it treats all data items as equal and populates the cache with random items.* Storage reads, however, often exhibit preferences for certain files within the same dataset. For instance, **Figure 13b** shows the cumulative distribution function (CDF) of file sizes in the ImageNet dataset [68], revealing significant variability in file sizes, spanning orders of magnitude. Storage systems, particularly HDDs, favor large file reads due to their sequential nature, which reduces seek and rotation overheads. **Figure 13c** illustrates that, when reading files randomly from disk, larger files achieve higher read throughput than smaller ones.

To address this limitation, we propose the **Small Files First (SFF)** caching, *prioritizing small files when populating the MinIO cache.* Before training, we read dataset metadata, sort files by size, and set a threshold (Th) such that all files smaller than Th exactly fill the cache. MinIO is then configured to cache only these files during the first epoch. This approach ensures that more small files are cached, leaving only large files to be read from storage.

We implemented SFF in MinIO and evaluated it using GPEmu. The cyan bars in **Figure 13a** show the training performance with MinIO+SFF. With the same cache size, SFF outperforms MinIO, especially at lower cache percentages. For example, with 20% cache,

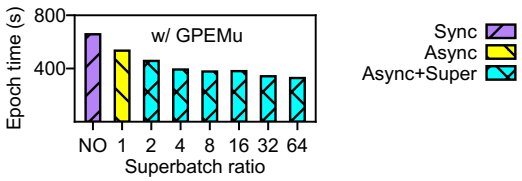

Figure 14: Async batch's benefit shown with GPEmu (§4.2).

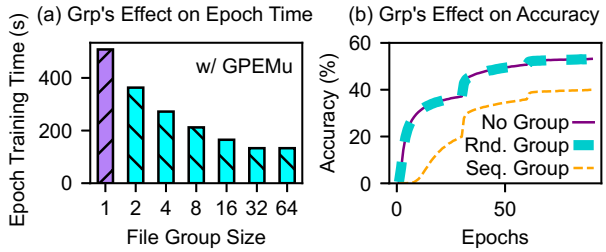

Figure 15: File grouping's effect (§4.3).

SFF reduces training time by 28% compared to MinIO. As the cache percentage increases, SFF's benefits diminish, as the files cached by MinIO increasingly align with the SFF strategy.

4.2 Async Batch

Another way to reduce the overhead of random reads is by leveraging I/O reordering, a technique employed by the Linux I/O scheduler [21]. The I/O scheduler reorders I/O requests based on their logical block addresses (LBA) in order to reduce seek time and rotational latency, hence increasing the overall throughput. The more requests the scheduler can process together, the greater the reduction is.

Unfortunately, we found that PyTorch's data loader does *not* fully benefit from I/O reordering. More specifically, PyTorch's official ImageNet training script [20] employs four workers to load data, with each worker handling one batch. Each worker reads *one* image at a time, waiting for the read to complete. Consequently, *at most* four read requests are simultaneously sent to the OS.

PyTorch's design is simple to implement; every worker can read and preprocess each image synchronously (without having to worry about concurrency issues). Overall, it also uses minimal memory; each worker only has a single batch in memory during loading. However, it is not efficient because the OS does not have enough in-flight I/Os to reorder to improve the I/O performance and the LBA gaps between the four target I/Os remain large, causing high seek and rotational overhead.

Ideally we should send more concurrent requests to the OS so that it may benefit more from reordering. To do this, we send all the requests in the entire batch. While this sounds simple, one minor complication is that we must move from the blocking/synchronous style to an asynchronous I/O design, otherwise the OS cannot see all the requests. To achieve this, we (1) utilize `io_uring` [32] asynchronous system call, (2) organize the sending and receiving of asynchronous I/Os using input and completion queues, and (3) carefully employ spinlocks and atomics to handle concurrency issues when using `io_uring`. Furthermore, since the OS is limited by its maximum queue depth of 2048 requests [1], when submitting more than this limit, we must pre-sort the requests by LBA at the application level before sending them to the OS.

Within this "*asynchronous-batch*" design, we found more room for optimization. The number of concurrent I/O requests is currently bottlenecked by the training batch size, and in some scenarios the batch size can be as small as 2 [25], limiting the benefits of I/O reordering. To overcome this limitation, we introduce "*superbatch*, " which is a multiple (*e.g.*, 2x) of the regular batch size. During data loading, we asynchronously send a superbatch of requests to the OS, maximizing the advantages of I/O reordering. It is important to note that superbatch only affects data loading and does not alter the training batch size or other training steps.

We evaluated the benefits of asynchronous (super)batch reading with GPEmu. **Figure 14** shows the epoch time for emulating AlexNet training with 4 workers and batch size of 16 using different data reading policies. Employing the asynchronous batch reading alone (yellow bar) reduces epoch time by 19% compared to the original PyTorch data loader (purple bar). After applying the superbatch approach, the reduction can be up to 50% (the cyan bars). Increasing the superbatch size too big yields minimal additional benefits, as I/O reordering is already efficiently utilized. Excessively large superbatch is also not recommended as it increases the memory pressure, which can adversely affect the epoch time.

4.3 File Grouping

The last two micro-optimizations have focused on reducing the penalty of random file reads for disk-based systems without altering the original random sampling order. In this subsection, we propose another optimization, termed "file grouping," which slightly reduces sampling randomness in exchange for enhanced I/O performance.

The file grouping approach works as follows: (a) Prior to training, we group every X random small files into one large file, storing location and labels in the metadata. For a dataset with M files, this results in M/X large files; (b) During training, instead of reading N random small files per batch, we read N/X random large files.

With GPEmu, we evaluated file grouping's impact on AlexNet training time with various group sizes, as is shown in **Figure 15a**. For example, without grouping (group size = 1), the training time is 498 seconds. With 8 images per group, the training time reduces by 2.4x. Interestingly, grouping the images more than 32 images offers no further improvement, implying that the disk seek overhead is already minimal compared to the data transfer time.

One concern about file grouping is its potential impact on model accuracy. Could the model become biased and lose accuracy? We experimented with *random-class grouping*, where each group mixes random "classes" (e.g., "cat", "dog", "snake" in the ImageNet dataset) and found no loss in accuracy (compared to no grouping) as shown in **Figure 15b**. We validated this across three different tasks (image classification, object detection, audio classification), using three datasets (ImageNet, COCO, Speech Commands) and five models (AlexNet, ResNet18, ResNet50, Faster R-CNN, M5), consistently observing the same pattern.

However, employing a naive *"sequential" grouping* (*i.e.*, grouping the unzipped ImageNet dataset images sequentially) significantly reduced accuracy, shown in the orange line in **Figure 15b**. This decline is likely because sequential groups contain images from

the same category, like all "cat"' pictures, leading to biased learning. This bias can cause catastrophic forgetting [41], where the model temporarily excels in the current category but may forget the previously learned ones.

5 DISCUSSION

We believe that we have built the major features for GPEmu. However, we acknowledge that GPEmu cannot fully replace real GPUs in all scenarios and can be extended to include more detailed features. Below, we discuss its non-goals and potential future extensions.

5.1 Non-goals

The following are GPEmu ' non-goals: (1) GPEmu cannot be used for evaluating accuracy metrics. (2) GPEmu cannot be used for optimizations that rely on (e.g. are tuned based on) dynamic values that evolve during training runtime, such as gradients, model parameter values, or popularity of embedding entries [33, 38, 52, 67].

5.2 Future extensions

Layer-level profiling: Currently, our profiling operates at the model level. A valuable enhancement would involve collecting runtime statistics at the layer level (e.g., linear layers and convolutional layers), allowing for the computation time of various models to be derived from a combination of these layers.

Comprehensive support for distributed training: GPEmu can be extended to emulate the network communication of gradient synchronization during Distributed Data-Parallel Training. Additionally, we aim to support other distributed training techniques, such as Fully Sharded Data Parallel (FSDP) [88] and Model Parallelism (MP) [69], in the future.

Support for DL Inference: While our work focuses on DL training, we believe GPEmu can be extended to emulate DL inference. Inference workloads are generally easier to emulate compared to training workloads, as they do not involve the complexities of backpropagation or gradient updates. Profiling for inference would primarily focus on forward pass computations and memory usage, which are relatively consistent for a given model and batch size.

This extension would enable evaluating and reproducing system optimizations for DL inference, as explored in works like [45, 64, 82].

GPU spatial sharing: The sharing support in GPEmu currently focuses on time sharing. A valuable extension would be to support spatial sharing, which enables multiple DL jobs to run concurrently on a GPU, considering the interference between DL jobs [71, 74, 77].

Transfer time variability: Our current data transfer time emulation assumes a dedicated PCIe link and does not explicitly account for scenarios involving PCIe link sharing or dynamically changing workloads. These scenarios can introduce significant variability in data transfer times. Future work may explore profiling techniques to better capture these complexities.

Support Large Language Models: GPEmu has primarily focused on traditional DL workloads. Given the rapid evolution and popularity of large language models (LLMs) recently, it would be valuable to extend GPEmu to emulate LLMs and foundation models.

6 RELATED WORK

Emulation of Other Resource Types: An emulator replicates a special piece of a system to enable it to run on a different platform. There have been a great number of emulation tools for emulating system pieces other than GPU, such as FEMU [56] and RAMSSD [28] for SSD emulation, FAME [59] and HME [39] for memory emulation, and Emulab [46] and CloudLab [12, 40] for network emulation.

Simulation of GPU: A simulator attempts to model the behaviors of a real-world system for analysis. The examples are GPU performance simulators like MacSim [53], GPGPU-Sim [35], MGPU-Sim [72], and Accel-Sim [51], as well as GPU cluster scheduling simulators used in prior papers [44, 47, 54, 60].

GPU simulators like GPGPU-Sim can simulate the execution of models on GPUs and report GPU-related performance statistics such as GPU memory usage and clock cycle count. However, they do not support end-to-end/full-stack DL system experiments which include other layers like storage, CPU, host memory, and network. In contrast, GPEmu can evaluate these experiments.

Emulation of GPU: There are a limited number of existing GPU emulators for DL workloads. Silod [83] profiled model compute time on the expensive V100 GPU and emulated it on a cluster of cheaper K80 GPUs to evaluate GPU cluster scheduling, but still requires real GPUs. MLPerf Storage [23] and DLCache [49] provide a simple host-side emulator that only performs compute time emulation. They all lack vital emulation components and support only a limited range of configurations and experiments.

Modeling DNN latency: Several studies have proposed methods to model the latency of DNNs [36, 37], where latency is estimated based on individual operations in the network, allowing generalization across different models. We recognize that these approaches are more scalable, and we believe their latency models could be integrated into GPEmu to support emulation of new models. On the other hand, our brute-force profiling offers real-world latency results, which are more reliable for capturing unpredictable interactions between GPU types and the components of AI models.

7 CONCLUSION

We present GPEmu, a comprehensive GPU emulator tailored for DL workloads, equipped with a wide array of emulation features and extensive support for diverse DL configurations. We hope GPEmu will follow prior emulators' successes and bring a strong impact to the community by enabling faster and cheaper prototyping and evaluation of deep learning system research.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their tremendous feedback and comments. This material was supported by funding from NSF (grant Nos. CCF-2119184, CNS-2402327, CNS-2027170, and CNS-2431425) as well as generous donations from Seagate. The experiments in this paper were performed on Chameleon [8, 50] and also used Google Cloud thru the CloudBank project [65] supported by NSF grant #1925001. Any opinions, findings, and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF or other institutions.

REFERENCES

[1] Accessed in November 2024. 8.4. Configuration Tools Red Hat Enterprise Linux 7 | Red Hat Customer Portal. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/performance_tuning_guide.

[2] Accessed in November 2024. Allox GitHub. https://github.com/lenhattan86/allox.

[3] Accessed in November 2024. Amazon EC2 P3 Instances. https://aws.amazon.com/ec2/instance-types/p3.

[4] Accessed in November 2024. AWS service quotas. https://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html.

[5] Accessed in November 2024. Azure and AWS's 'GPU general availability' lies. https://www.fast.ai/posts/2016-12-19-gpu-lies.html.

[6] Accessed in November 2024. Azure Machine Learning pricing. https://azure.microsoft.com/en-us/pricing/details/machine-learning/.

[7] Accessed in November 2024. Cannot Extend GPU Quota on Google Cloud. https://stackoverflow.com/questions/48362544/cannot-extend-gpu-quota-on-google-cloud.

[8] Accessed in November 2024. Chameleon - A configurable experimental environment for large-scale cloud research. https://www.chameleoncloud.org.

[9] Accessed in November 2024. Cloudbank Website. https://www.cloudbank.org.

[10] Accessed in November 2024. DALI. https://developer.nvidia.com/dali.

[11] Accessed in November 2024. Device Plugins | Kubernetes. https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/.

[12] Accessed in November 2024. Emulating multipath wireless links on CloudLab and FABRIC. https://witestlab.poly.edu/blog/emulating-multipath-wireless/.

[13] Accessed in November 2024. FastFlow GitHub. https://github.com/SamsungLabs/FastFlow.

[14] Accessed in November 2024. GCE Discussion: GPU Quota. https://groups.google.com/g/gce-discussion/c/mtHV1NlKKBo/m/i9uyk-PeAgAJ.

[15] Accessed in November 2024. GCE Discussion: No P100 GPUs in any us zone. https://groups.google.com/g/gce-discussion/c/34zBBmTV8Tg.

[16] Accessed in November 2024. GCE Discussion: Not enough resources to fulfill for the past 14 hours. https://groups.google.com/g/gce-discussion/c/8vCwUKaGs2o.

[17] Accessed in November 2024. Getting Started with Distributed Data Parallel. https://pytorch.org/tutorials/intermediate/ddp_tutorial.html.

[18] Accessed in November 2024. Google Cloud GPU Pricing. https://cloud.google.com/compute/gpus-pricing.

[19] Accessed in November 2024. How to Optimize Data Transfers in CUDA C/C++. https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/.

[20] Accessed in November 2024. ImageNet training in PyTorch. https://github.com/pytorch/examples/tree/main/imagenet.

[21] Accessed in November 2024. Linux I/O schedulers. https://wiki.ubuntu.com/Kernel/Reference/IOSchedulers.

[22] Accessed in November 2024. MinIO GitHub. https://github.com/msr-fiddle/CoorDL.

[23] Accessed in November 2024. MLPerf Storage Benchmark Suite GitHub. https://github.com/mlcommons/storage.

[24] Accessed in November 2024. Muri GitHub. https://github.com/pkusys/Muri.

[25] Accessed in November 2024. Object detection reference training scripts. https://github.com/pytorch/vision/tree/main/references/detection.

[26] Accessed in November 2024. PyTorch CUDA Asynchronous Execution. https://pytorch.org/docs/master/notes/cuda.html#asynchronous-execution.

[27] Accessed in November 2024. RabbitMQ: easy to use, flexible messaging and streaming — RabbitMQ. https://www.rabbitmq.com.

[28] Accessed in November 2024. RAMSSD GitHub. https://github.com/thustorage/ramssd.

[29] Accessed in November 2024. Synergy GitHub. https://github.com/msr-fiddle/synergy.

[30] Accessed in November 2024. Time-sharing GPUs on GKE | Google Kubernetes Engine (GKE) | Google Cloud. https://cloud.google.com/kubernetes-engine/docs/concepts/timesharing-gpus.

[31] Accessed in November 2024. Time-Slicing GPUs in Kubernetes — NVIDIA GPU Operator. https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/gpu-sharing.html.

[32] Accessed in November 2024. What is io-uring? https://unixism.net/loti/what_is_io_uring.html.

[33] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J Nair. 2021. Accelerating Recommendation System Training by Leveraging Popular Choices. *Proceedings of the VLDB Endowment (PVLDB)* 15, 1 (2021), 127–140.

[34] Andrew Audibert, Yang Chen, Dan Graur, Ana Klimovic, Jiří Šimša, and Chandramohan A Thekkath. 2023. tf. data service: A case for disaggregating ML input data processing. In *Proceedings of the 2023 ACM Symposium on Cloud Computing (SoCC)*. 358–375.

[35] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 163–174.

[36] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. 2019. Once-for-all: Train one network and specialize it for efficient deployment. *arXiv preprint arXiv:1908.09791* (2019).

[37] Han Cai, Ligeng Zhu, and Song Han. 2018. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332* (2018).

[38] Weijian Chen, Shuibing He, Yaowen Xu, Xuechen Zhang, Siling Yang, Shuang Hu, Xian-He Sun, and Gang Chen. 2023. icache: An importance-sampling-informed cache for accelerating i/o-bound dnn model training. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 220–232.

[39] Zhuohui Duan, Haikun Liu, Xiaofei Liao, and Hai Jin. 2018. HME: A lightweight emulator for hybrid memory. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1375–1380.

[40] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. 2019. The Design and Operation of CloudLab. In *2019 USENIX annual technical conference (USENIX ATC)*. 1–14.

[41] Robert M French. 1999. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences* 3, 4 (1999), 128–135.

[42] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press.

[43] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A Thekkath, and Ana Klimovic. 2022. Cachew: Machine learning input data processing as a service. In *2022 USENIX Annual Technical Conference (USENIX ATC)*. 689–706.

[44] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. 2019. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 485–500.

[45] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 443–462.

[46] Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb, and Jay Lepreau. 2008. Large-scale virtualization in the emulab network testbed. In *2008 USENIX Annual Technical Conference (USENIX ATC)*.

[47] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R Ganger. 2023. Sia: Heterogeneity-aware, goodput-optimized ML-cluster scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*. 642–657.

[48] Aarati Kakaraparthy, Abhay Venkatesh, Amar Phanishayee, and Shivaram Venkataraman. 2019. The case for unifying data loading in machine learning clusters. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*.

[49] Zhuangwei Kang, Ziran Min, Shuang Zhou, Yogesh D Barve, and Aniruddha Gokhale. 2023. Dataset Placement and Data Loading Optimizations for Cloud-Native Deep Learning Workloads. In *2023 IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 107–116.

[50] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S Gunawi, Cody Hammock, et al. 2020. Lessons learned from the chameleon testbed. In *2020 USENIX annual technical conference (USENIX ATC)*. 219–233.

[51] Mahmoud Khairy, Zhesheng Shen, Tor M Aamodt, and Timothy G Rogers. 2020. Accel-Sim: An extensible simulation framework for validated GPU modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 473–486.

[52] Redwan Ibne Seraj Khan, Ahmad Hossein Yazdani, Yuqi Fu, Arnab K Paul, Bo Ji, Xun Jian, Yue Cheng, and Ali R Butt. 2023. SHADE: Enable Fundamental Cacheability for Distributed Deep Learning Training. In *21st USENIX Conference on File and Storage Technologies (FAST)*. 135–152.

[53] Hyesoon Kim, Jaekyu Lee, Nagesh B Lakshminarayana, Jaewoong Sim, Jieun Lim, and Tri Pho. 2012. Macsim: A cpu-gpu heterogeneous simulation framework user guide. *Georgia Institute of Technology* (2012), 1–57.

[54] Tan N Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. 2020. Allox: compute allocation in hybrid clusters. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*. 1–16.

[55] Guillaume Leclerc, Andrew Ilyas, Logan Engstrom, Sung Min Park, Hadi Salman, and Aleksander Mądry. 2023. FFCV: Accelerating training by removing data bottlenecks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 12011–12020.

[56] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S Gunawi. 2018. The CASE of FEMU: Cheap, accurate, scalable and extensible flash emulator. In *16th USENIX Conference on File and Storage Technologies (FAST)*. 83–90.

[57] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. 2020. PyTorch distributed: experiences on accelerating data parallel training. *Proceedings of the VLDB Endowment (PVLDB)* 13, 12 (2020), 3005–3018.

[58] Jie Liu, Bogdan Nicolae, and Dong Li. 2022. Lobster: Load balance-aware I/O for distributed DNN training. In *Proceedings of the 51st International Conference on Parallel Processing (ICPP)*. 1–11.

[59] Krishna T Malladi, Mu-Tien Chang, John Ping, and Hongzhong Zheng. 2015. FAME: A fast and accurate memory emulator for new memory system architecture exploration. In *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 43–46.

[60] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. 2022. Looking Beyond GPUs for DNN Scheduling on Multi-Tenant Clusters. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 579–596.

[61] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. 2021. Analyzing and mitigating data stalls in DNN training. *Proceedings of the VLDB Endowment (PVLDB)* 14, 5 (2021), 771–784.

[62] Derek G Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. 2021. tf. data: a machine learning data processing framework. *Proceedings of the VLDB Endowment (PVLDB)* 14, 12 (2021), 2945–2958.

[63] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 481–498.

[64] Kelvin KW Ng, Henri Maxime Demoulin, and Vincent Liu. 2023. Paella: Low-latency model serving with software-defined gpu scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*. 595–610.

[65] Michael Norman, Vince Kellen, Shava Smallen, Brian DeMeulle, Shawn Strande, Ed Lazowska, Naomi Alterman, Rob Fatland, Sarah Stone, Amanda Tan, et al. 2021. Cloudbank: Managed services to simplify cloud access for computer science research and education. In *Practice and Experience in Advanced Research Computing (PEARC)*. 1–4.

[66] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems (NIPS)* 32 (2019).

[67] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. 2021. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[68] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. Imagenet large scale visual recognition challenge. *International journal of computer vision (IJCV)* 115 (2015), 211–252.

[69] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[70] Dheeraj Sreedhar, Vaibhav Saxena, Yogish Sabharwal, Ashish Verma, and Sameer Kumar. 2018. Efficient training of convolutional neural nets on large distributed systems. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 392–401.

[71] Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys)*. 1075–1092.

[72] Yifan Sun, Trinayan Baruah, Saiful A Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, Carter McCardwell, Vincent Zhao, et al. 2019. MGPUSim: Enabling multi-GPU performance modeling and optimization. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*. 197–209.

[73] Taegeon Um, Byungsoo Oh, Byeongchan Seo, Minhyeok Kweun, Goeun Kim, and Woo-Yeon Lee. 2023. Fastflow: Accelerating deep learning model training with smart offloading of input data pipeline. *Proceedings of the VLDB Endowment (PVLDB)* 16, 5 (2023), 1086–1099.

[74] Guanhua Wang, Kehan Wang, Kenan Jiang, Xiangjun Li, and Ion Stoica. 2021. Wavelet: Efficient DNN training with tick-tock scheduling. *Proceedings of Machine Learning and Systems (MLSys)* 3 (2021), 696–710.

[75] Meng Wang, Gus Waldspurger, and Swaminathan Sundararaman. 2024. A Selective Preprocessing Offloading Framework for Reducing Data Traffic in DL Training. In *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage)*. 63–70.

[76] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. 2023. Transparent GPU Sharing in Container Clouds for Deep Learning Workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 69–85.

[77] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. 2018. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 595–610.

[78] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 533–548.

[79] Lijie Xu, Shuang Qiu, Binhang Yuan, Jiawei Jiang, Cedric Renggli, Shaoduo Gan, Kaan Kara, Guoliang Li, Ji Liu, Wentao Wu, et al. 2022. In-database machine learning with corgipile: Stochastic gradient descent without full data shuffle. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD)*. 1286–1300.

[80] Chih-Chieh Yang and Guojing Cong. 2019. Accelerating data loading in deep neural network training. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 235–245.

[81] Peifeng Yu and Mosharaf Chowdhury. 2020. Fine-Grained GPU Sharing Primitives for Deep Learning Applications. In *Proceedings of Machine Learning and Systems (MLSys)*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 98–111.

[82] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. 2023. SHEPHERD: Serving DNNs in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 787–808.

[83] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Mingxia Li, Fan Yang, Qianxi Zhang, Binyang Li, Yuqing Yang, Lili Qiu, et al. 2023. Silod: A co-design of caching and scheduling for deep learning clusters. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys)*. 883–898.

[84] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis CM Lau, Yuqi Wang, Yifan Xiong, et al. 2020. HiveD: Sharing a GPU Cluster for Deep Learning with Guarantees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 515–532.

[85] Hanyu Zhao, Zhi Yang, Yu Cheng, Chao Tian, Shiru Ren, Wencong Xiao, Man Yuan, Langshi Chen, Kaibo Liu, Yang Zhang, et al. 2023. Goldminer: Elastic scaling of training data pre-processing pipelines for deep learning. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[86] Mark Zhao, Emanuel Adamiak, and Christos Kozyrakis. 2024. cedar: Composable and Optimized Machine Learning Input Data Pipelines. *arXiv preprint arXiv:2401.08895* (2024).

[87] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, et al. 2022. Understanding data storage and ingestion for large-scale deep recommendation model training: Industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*. 1042–1057.

[88] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. 2023. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277* (2023).

[89] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. 2022. Multi-resource interleaving for deep learning training. In *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM)*. 428–440.

[90] Yue Zhu, Fahim Chowdhury, Huansong Fu, Adam Moody, Kathryn Mohror, Kento Sato, and Weikuan Yu. 2018. Entropy-aware I/O pipelining for large-scale deep learning on HPC systems. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 145–156.