An Empirical Study on Low GPU Utilization of Deep Learning Jobs

Yanjie Gao
Microsoft Research
Beijing, China
yanjga@microsoft.com

Yichen He*
Microsoft Research
Beijing, China
v-yichenhe@microsoft.com

Xinze Li*
Peking University
Beijing, China
panzer_marching@163.com

Bo Zhao*
Microsoft Research
Beijing, China
v-bozha@microsoft.com

Haoxiang Lin†
Microsoft Research
Beijing, China
haoxlin@microsoft.com

Yoyo Liang
Microsoft
Beijing, China
yoliang@microsoft.com

Jing Zhong
Microsoft
Beijing, China
jinzhong@microsoft.com

Hongyu Zhang
Chongqing University
Chongqing, China
hyzhang@cqu.edu.cn

Jingzhou Wang*
Tsinghua University
Beijing, China
jz-wang20@mails.tsinghua.edu.cn

Yonghua Zeng
Microsoft
Beijing, China
yozen@microsoft.com

Keli Gui
Microsoft
Beijing, China
keligui@microsoft.com

Jie Tong
Microsoft
Beijing, China
jietong@microsoft.com

Mao Yang
Microsoft Research
Beijing, China
maoyang@microsoft.com

ABSTRACT

Deep learning plays a critical role in numerous intelligent software applications. Enterprise developers submit and run deep learning jobs on shared, multi-tenant platforms to efficiently train and test models. These platforms are typically equipped with a large number of graphics processing units (GPUs) to expedite deep learning computations. However, certain jobs exhibit rather low utilization of the allocated GPUs, resulting in substantial resource waste and reduced development productivity. This paper presents a comprehensive empirical study on low GPU utilization of deep learning jobs, based on 400 real jobs (with an average GPU utilization of 50% or less) collected from Microsoft's internal deep learning platform. We discover 706 low-GPU-utilization issues through meticulous examination of job metadata, execution logs, runtime metrics, scripts, and programs. Furthermore, we identify the common root causes and propose corresponding fixes. Our main findings include: (1) Low GPU utilization of deep learning jobs stems from insufficient GPU computations and interruptions caused by non-GPU tasks; (2) Approximately half (46.03%) of the issues are attributed to data operations; (3) 45.18% of the issues are related to deep learning models and manifest during both model training and evaluation stages; (4) Most (84.99%) low-GPU-utilization issues could be fixed with a small number of code/script modifications. Based on the study results, we propose potential research directions that could help developers utilize GPUs better in cloud-based platforms.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**.

KEYWORDS

deep learning jobs, GPU utilization, empirical study

ACM Reference Format:
Yanjie Gao, Yichen He, Xinze Li, Bo Zhao, Haoxiang Lin, Yoyo Liang, Jing Zhong, Hongyu Zhang, Jingzhou Wang, Yonghua Zeng, Keli Gui, Jie Tong, and Mao Yang. 2024. An Empirical Study on Low GPU Utilization of Deep Learning Jobs. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639232

1 INTRODUCTION

Deep learning (DL) has made remarkable achievements in various areas, including natural language processing, gaming, and image recognition. It is now playing a critical role in numerous intelligent software applications. To facilitate efficient DL model training and testing, enterprises have established shared, multi-tenant platforms such as Microsoft Azure Machine Learning [47], Amazon SageMaker [2], and Google Vertex AI [20] for developers to submit and

*The work was performed during the internship at Microsoft Research.
†Haoxiang Lin is the corresponding author.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04
https://doi.org/10.1145/3597503.3639232

run their DL jobs. These platforms are equipped with a large number of hardware accelerators dedicated to DL computation, among which graphics processing units (GPUs) are typically used. Given the computational intensity inherent in deep learning [72], *GPU utilization*—a measurement of the actual GPU computing time as a percentage of the total GPU service (i.e., allocation) time—serves as a crucial indicator for both the runtime performance of DL jobs and the service-level agreement compliance within DL platforms.

Microsoft is a multinational IT company offering cloud computing and intelligent services. Internally, Microsoft houses Platform-X, a deep learning platform constructed with commodity computing hardware (e.g., GPUs and RDMA networks) and widely used open-source software (e.g., Kubernetes [6] and Docker [44]). Every day, hundreds of developers from various research and product teams train and test their DL models on Platform-X for tasks such as object detection, machine translation, and advertising. These DL jobs exclusively occupy GPUs due to the lack of mature and efficient hardware/software support for fine-grained GPU sharing [21, 66]. Contrary to the expectation that DL jobs should fully utilize computing resources, we observe that some of them exhibit rather low utilization of the allocated GPUs. For example, a job with eight NVIDIA Tesla V100 GPUs used a too small batch size, causing its GPU utilization to fluctuate primarily between 10% and 40%. Another scenario involved a job frequently saving model checkpoints to a distributed data store for fault tolerance. The large model size and synchronous remote upload led to recurrent idle GPU periods, severely hindering the training process. Low GPU utilization of DL jobs not only results in a substantial waste of precious platform resources (including GPUs, CPUs, main memory, network bandwidth, and storage) but also reduces development productivity. Such adverse effects may be exacerbated in the widely adopted practice of automated machine learning (AutoML), where many trial jobs of the same experiment could share analogous computation processes and GPU utilization since their programs, neural architectures, and hyperparameter values are highly similar. As deep learning becomes a fundamental component of modern software applications, and cloud-based platforms serve as the predominant infrastructure for training and deploying models, it is particularly relevant and essential for software engineering researchers and practitioners to comprehend the reasons behind low GPU utilization in DL jobs and seek solutions. Addressing these issues helps unveil unique software engineering challenges in the deep learning domain [41] and develop high-quality, cost-efficient software solutions.

Considerable research has been conducted on CPU utilization [5, 22, 27, 50, 74]. However, previous work could not sufficiently support developers in enhancing GPU utilization due to fundamental differences between CPUs and GPUs. These differences encompass various aspects, including the number of computing cores, cache/memory size, interconnect bandwidth, nature of scheduling, and application programming interface (API). Recently, researchers have performed quite a few empirical studies on deep learning [7, 25, 26, 28, 29, 69, 82–84], many of which concentrate on the failures and defects of deep learning programs, jobs, frameworks, and compilers. For example, Zhang et al. [84] analyzed 175 TensorFlow [1] program bugs from GitHub and Stack Overflow. Investigating low GPU utilization in large-scale, multi-tenant GPU clusters dedicated to deep learning training, Jeon et al. [28] identified resource locality, gang scheduling [55], and job failures as the root causes from a platform perspective. Their research primarily focuses on the GPU utilization of clusters instead of individual jobs.

This paper conducts the first comprehensive empirical study on low GPU utilization of industrial deep learning jobs. We randomly selected 400 real jobs from Platform-X as study subjects. The average GPU utilization of each job was less than or equal to 50%—a utilization threshold determined in collaboration with the Platform-X team. Through meticulous examination of job metadata, execution logs, runtime metrics, scripts, and programs, we discovered 706 low-GPU-utilization issues across the sampled jobs. These issues were attributed to the code logic of scripts and programs. We further identified the common root causes, classified them into four high-level dimensions and fifteen categories, and proposed fixes.

We obtain many findings and list the main ones as follows:

(1) Low GPU utilization of deep learning jobs stems from insufficient GPU computations (e.g., using a small batch size or running a non-DL, CPU-centric job) and interruptions caused by non-GPU tasks (e.g., saving a model checkpoint synchronously to the distributed data store).
(2) Approximately half (46.03%) of the issues are attributed to data operations, including inefficient data transfer between main memory and GPU memory (27.90%) and continual data exchange among GPUs in distributed training (7.08%).
(3) 45.18% of the issues are related to deep learning models and manifest during both model training and evaluation stages, such as using improper batch sizes (25.64%) and performing less efficient model checkpointing (16.43%).
(4) Most (84.99%) low-GPU-utilization issues could be fixed with a small number of code/script modifications. However, the remainder will necessitate enhancements in deep learning platforms, frameworks, and toolchains. Experimental results on the BERT [11] and Swin Transformer [38] jobs demonstrate substantial speedups of up to 7.52X and 3.95X, respectively.

Based on the study results, we provide guidelines to help deep learning developers utilize GPUs more efficiently. Moreover, we propose potential future research directions.

In summary, this paper makes the following contributions:

(1) We identify a critical, challenging, and timely research problem in real-world enterprise AI development practices: low GPU utilization of deep learning jobs.
(2) Through the first comprehensive empirical study on 400 real industrial jobs, we discover 706 low-GPU-utilization issues, identify their root causes, and propose corresponding fixes.
(3) We present the findings of our empirical study and recommend enhancements for both developers and platforms to improve GPU utilization.

2 BACKGROUND: DEEP LEARNING JOBS ON PLATFORM-X

Platform-X, Microsoft's internal deep learning platform, operates across multiple physical GPU clusters, supporting hundreds of developers from various research and product teams. Every day, thousands of DL jobs for tasks like object detection, machine translation, advertising, and gaming are submitted and executed on Platform-X.

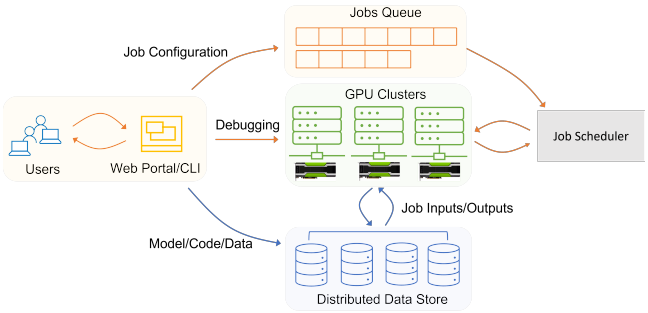

Figure 1: Workflow of Platform-X.

Platform-X adopts comparable types of computing hardware, widely used open-source software, and a standard DL programming paradigm. Therefore, the job submission and execution of Platform-X closely resemble those of public platforms such as Microsoft Azure Machine Learning [47], Amazon SageMaker [2], and Google Vertex AI [20]. Figure 1 illustrates a concise overview of Platform-X's workflow. Firstly, developers upload their input data, scripts, programs, and model checkpoints (for continuous training) to a distributed data store via the web portal or a command-line tool. The training code is typically written in Python, utilizing DL frameworks like PyTorch [56] and TensorFlow [1]. Next, developers specify the job configuration, including resource quota (e.g., preferred GPU type and quantity), input/output paths, Docker [44] image, startup Shell script, main program file, etc. Platform-X offers a standard deep learning toolchain (e.g., official PyTorch Docker images) to establish a hermetic environment for job execution. The submitted jobs are initially queued for scheduling. Once a job is selected, Platform-X allocates all requested GPUs and resources at once (using gang scheduling [55]) and instantiates Docker containers to execute the startup script and main program file. The subsequent model training takes place on one or more GPU machines provided by Platform-X. If the job experiences an unexpected slowdown or a sudden crash during execution, the developer can directly connect to those worker machines for remote debugging.

From a developer's perspective, the lifecycle of a DL job is typically divided into the following four consecutive stages:

(1) Initialization. The job customizes the execution environment on Platform-X to align with the developer's local setup due to inherent environmental differences [82]. For example, many dependent libraries might be missing, particularly when the developer specifies an official DL Docker image. Accordingly, the startup Shell script must install them by either invoking the `pip` tool or cloning and building the source code.
(2) Data preprocessing. This stage primarily involves cleaning and augmenting [18] input data. As the job continually utilizes remote data over extended periods, a common practice is to initially download input data to all worker machines.
(3) Model training. The deep learning model—essentially a layered data representation [18]—is constructed using mathematical operations termed *operators*, such as `Conv2d` (2D convolution) and `ReLU` (rectified linear unit). Subsequently, the training process iteratively updates these operators' learnable parameters (biases and weights) until the model achieves the desired learning performance (e.g., predictive accuracy). Periodically, the job assesses its current learning performance to prompt adjustments to hyperparameter values or initiate early termination if necessary.
(4) Model evaluation (i.e., testing). After training, the job quantifies the final learning performance and saves the trained model and evaluation results to the distributed data store.

Platform-X employs Prometheus [61] to monitor system status and gather real-time metrics for each DL job at regular intervals. GPU utilization data is retrieved from the NVIDIA Data Center GPU Manager (DCGM)[1]. Additional metrics include GPU memory footprint, CPU utilization, main memory footprint, network sent/received bytes, disk read/write bytes, etc.

3 STUDY METHODOLOGY

3.1 Study Design

Our goal is to understand and resolve low GPU utilization of deep learning jobs. In this paper, we limit our research scope to the real jobs on Platform-X. These jobs target a wide range of application areas, and Platform-X shares broad similarities with other deep learning platforms. We aim to answer the following two questions:

(1) What are the root causes of low-GPU-utilization issues?
(2) What are the current solutions to fix these issues?

The first question aims at revealing the essential reasons behind the emergence of low-GPU-utilization issues, and the findings can facilitate the development of effective approaches for issue localization. The second question tries to devise practical, non-intrusive, and easily validated fixes. Therefore, developers can promptly apply them to mitigate low-GPU-utilization issues without breaking their programs and jobs. Indeed, we have motivated and discussed more advanced fixes in Section 5.2. These include GPU utilization estimation/prediction under various hyperparameter values, heterogeneous pipelining, sophisticated model checkpointing, and many others. However, implementing such advanced fixes is considerably complicated, necessitating major code modifications to the jobs, along with indispensable platform and tool support.

In contrast to many prior related studies [7, 26, 29, 69, 83, 84], there is a lack of available human labels, discussions, conclusions, and fixes for reference in our research, such as those commonly found in Stack Overflow questions and GitHub issues. Consequently, we perform manual analysis from scratch using domain expertise and experience to answer the above two questions. Reference materials include the related work [7, 26, 28, 84] and best practices of Platform-X and other platforms [3, 19, 46]. To minimize subjectivity, each job is independently analyzed by two of the authors, and we calculate Cohen's kappa statistic [10] to assess the inter-rater reliability of root cause labeling. When the analysis is over, we involve group discussion to review the results, resolve disagreements, finalize the classification of root causes, and devise fixes for each issue category. We further organize the issue categories into high-level dimensions. Experiments are conducted to confirm the issues and validate the fixes. Regarding complicated cases, we contact the job submitters directly for clarification.

[1] https://developer.nvidia.com/dcgm

3.2 Data Collection

As mentioned before, Platform-X collects the GPU utilization of jobs periodically. Suppose that a job requested N GPUs, with t and $\hat{t}$ as its start and end times, respectively. Let t_j be K time points ($1 \leq j \leq K$, and $t = t_0 < t_1 < \cdots < t_K = \hat{t}$) at which $u_{i,j}$—the current GPU utilization of the i-th GPU ($1 \leq i \leq N$)—was collected. Therefore, we calculated the average GPU utilization of the i-th GPU (denoted by U_i) and the job (denoted by U) as follows:

$$U_{i\in[1,N]} = \frac{\sum_{j=1}^{K} u_{i,j} \times (t_j - t_{(j-1)})}{\hat{t} - t}, \quad U = \frac{\sum_{i=1}^{N} U_i}{N}.$$

We started by crawling all jobs submitted within ten days in August 2021 by developers from both product and research teams. Next, we filtered out the running, failed, and terminated jobs because their lifecycles were essentially incomplete, and thus, certain low-GPU-utilization issues may not be exposed. The system jobs from dedicated user accounts for testing purposes only were also excluded. Then, we retained only those jobs with an average GPU utilization of 50% or less—a utilization threshold determined in collaboration with the Platform-X team, considering the current overall GPU utilization of Platform-X as well as domain expertise and experience. Lastly, we randomly selected 400 jobs across all job submitters, teams, clusters, and application areas to ensure greater diversity. These jobs involved training representative models, such as BERT [11], Swin Transformer [38], NeRF [48], ResNet50, Inception-V3, and Graph Neural Networks. They targeted a wide range of application areas, including but not limited to natural language processing, computer vision, bioinformatics, speech and audio processing, search engines, advertising, and gaming. For each selected job, we also collected relevant information for subsequent investigation, including, for example, job metadata, execution logs, various runtime metrics, scripts, and programs.

3.3 Identification of Low-GPU-Utilization Issues

We analyzed all 400 jobs manually from scratch. For each candidate job, we first thoroughly inspected the user-written Shell scripts and Python programs to understand the job's purpose and which devices (such as GPU, CPU, or network) it would operate. If a code statement invoked third-party library APIs, we also referred to the library documentation and looked through the library source code when possible.

Secondly, we correlated Shell and Python code with runtime metrics, such as the utilization of GPUs, CPUs, and main memory, for further investigation. Essentially, we located log messages to infer a code snippet's start and end timestamps. For example, developers often print messages before and after model checkpointing statements. As another example, package installation statements using the `pip` tool also output details about the installation process. Additionally, DL jobs usually write logs (e.g., printing the iteration number) and perform non-GPU tasks (e.g., accuracy calculation) at the beginning and end of a training or evaluation iteration. This causes the GPU utilization to drop suddenly and then recover, allowing us to infer the execution period of each iteration. With the start and end timestamps, we extracted the runtime metrics from the time range and correlated them with the code.

Thirdly, we re-evaluated the execution of Shell scripts and Python programs by cross-referencing them with correlated runtime data. The objective was to ascertain whether the allocated GPUs had been in full use during their execution. Low-GPU-utilization issues could potentially arise if there was room for improving one or more GPUs' utilization of GPU-related code (e.g., trying a larger batch size), or if there existed opportunities to minimize non-GPU-related operations through reductions, eliminations, or overlaps with other operations (e.g., transitioning from synchronous to asynchronous model checkpointing). If uncertain about specific details, we contacted the job owner directly for clarification.

In the end, our investigation discovered a total of 706 low-GPU-utilization issues attributed to the code logic of the jobs' scripts and programs. These issues were classified into fifteen categories by root causes, employing a general classification schema established in prior studies [7, 26, 82, 84] (such as "Improper Batch Size", "Data Preprocessing", and "API Misuse"). We also supplemented some new categories (such as "GPU Oversubscription", "Unreleased Job", and "External Data Usage"). To present a comprehensive overview, we further organized the categories into four high-level dimensions: Job, Model, Data, and Library, each associated with the execution of jobs. We devised fixes based on domain expertise and experience, related work [7, 26, 28, 84], and best practices of Platform-X and other platforms [3, 19, 46]. Moreover, experiments on real jobs (such as BERT [11], Swin Transformer [38], NeRF [48], Fairseq [54], and many others) were conducted to confirm the issues and validate the fixes.

3.4 Threats to Validity

Threats to Internal Validity. Due to the unavailability of reference labeling data, we manually analyzed all jobs using domain expertise and experience. Therefore, inherent subjectivity arises in analyzing issues, identifying root causes, and proposing fixes, owing to the complexity of deep learning and the extensive manual effort involved. To mitigate this threat, each job was independently analyzed by two of the authors. The inter-rater reliability (Cohen's kappa statistic [10]) of root cause labeling is 94.51%, indicating a strong level of agreement. Moreover, experiments were conducted on Platform-X to confirm the issues and validate the fixes. In instances of disagreement, we strived to reach a group consensus before finalizing decisions. Regarding complicated jobs, we contacted their submitters directly for clarification.

Threats to External Validity. We conducted our study on DL jobs that were all collected from Platform-X. These jobs train representative models and target a wide range of application areas. Additionally, Platform-X shares broad similarities with other DL platforms in platform architecture, software stack, job management, and toolchain. However, it is possible that certain findings might pertain exclusively to Microsoft and might not be applicable to other companies and platforms. To mitigate this threat, we strived to avoid drawing conclusions specific to Platform-X and Microsoft within the study. In Section 5.1, we discuss the generality of our findings in detail. Another potential threat might be the timeliness of certain fixes dependent on specific hyperparameters or API behaviors (e.g., setting a particular `pin_memory` parameter to fix Inefficient Host-GPU Data Transfer issues in Section 4.3). These

Table 1: Classification of the 706 low-GPU-utilization issues discovered across 400 deep learning jobs.

Dimension	Category	Description	No.	Ratio
Job	Interactive Job	The execution of a job entails regular interaction with its owner.	15	2.12%
	GPU Oversubscription	A job requests more GPUs than it actually utilizes.	6	0.85%
	Unreleased Job	A job does not terminate promptly after completing its computation.	9	1.27%
	Non-DL Job	A job is unrelated to deep learning and does not utilize GPUs at all. For example, the job performs data analysis using CPUs solely.	4	0.57%
	Subtotal		34	4.82%
Model	Improper Batch Size	Improper values of the batch size are used, which decrease the GPU computation of deep learning operators.	181	25.64%
	Insufficient GPU Memory	The GPU memory is not sufficient to support more GPU computation.	22	3.12%
	Model Checkpointing	A job saves model checkpoints synchronously to the distributed data store.	116	16.43%
	Subtotal		319	45.18%
Data	Inefficient Host-GPU Data Transfer	The data transfer between main memory and GPU memory is not efficient.	197	27.90%
	Data Preprocessing	Raw input data is preprocessed using CPUs before model training.	28	3.97%
	Remote Data Read	A job opens and reads input data directly from the distributed data store.	18	2.55%
	External Data Usage	A job accesses input data or model files directly from external sites.	18	2.55%
	Intermediate Result Upload	A job saves intermediate training results synchronously to the distributed data store.	14	1.98%
	Data Exchange	The GPUs of a distributed job continually exchange data, such as gradients and output tensors, among one another.	50	7.08%
	Subtotal		325	46.03%
Library	Long Library Installation	The installation of dependent libraries takes too much time (at least 10 minutes).	12	1.70%
	API Misuse	API usage violates assumptions.	16	2.27%
	Subtotal		28	3.97%
Total			706	100%

fixes may become outdated due to future software updates. In practice, platform engineers could update such fixes more promptly to mitigate the threat.

4 STUDY RESULTS

This section presents the categorization of issues and their respective fixes in detail. Table 1 illustrates the 706 low-GPU-utilization issues discovered across 400 deep learning jobs, classified into fifteen categories by root causes. We further organize these categories into four high-level dimensions: Job, Model, Data, and Library.

4.1 Dimension 1: Job

This dimension consists of four categories and encompasses 34 (4.82%) low-GPU-utilization issues, attributed to inappropriate job types or configurations. These issues typically lead to prolonged idle times or the non-use of one or more GPUs.

Interactive Job (15; 2.12%) means that a DL job entails regular developer interaction during its execution. For example, a job utilizing four NVIDIA V100 GPUs ran for approximately 251.8 minutes. However, the average GPU utilization of this job was less than 1% because it regularly utilized CPUs for interaction, relegating GPUs to a brief usage window. In most cases, we observed that developers connected remotely to Platform-X using SSH. They may either develop and test their models on Platform-X as they would on local development machines or debug and optimize their jobs on the platform. Alternatively, developers launched Jupyter [31] notebooks to execute programs line by line. Due to the non-deterministic nature of interaction, it is impossible to predict when and for how long the GPUs will remain idle in advance. A common fix to these issues is to prevent the submission of interactive jobs. Moreover, platforms could employ preemptible resources to establish a dedicated service (e.g., JupyterHub) for interactive computing.

GPU Oversubscription (6; 0.85%): Occasionally, developers request more GPUs than they actually utilize. When dealing with distributed jobs, it is crucial to accurately determine the total number of GPUs for even workload distribution. Typically, developers specify this quantity as a parameter. However, errors may occur when the specified value is mistakenly set lower than the actual number of GPUs. For example, in a PyTorch job, while half of the eight GPUs exhibited good utilization at 83.24%, the remaining GPUs showed zero utilization. Our investigation into the scripts and programs revealed an incorrect setting for the `nproc_per_node` parameter[2], which was erroneously configured to 4 instead of 8 (all GPUs originating from a single machine). As a result, four GPUs remained idle throughout the job's lifecycle. To fix this type of issue, developers must meticulously assess how many GPUs they really need and ensure that their jobs receive the correct number of GPUs. Moreover, platforms could facilitate resolution by offering a standardized interface that enables easier access to essential job configurations, such as through environment variables.

Unreleased Job (9; 1.27%) means that a DL job does not terminate promptly after completing its computation. We observed a job being forced to sleep for an extra ten minutes after training. Subsequently, we contacted the developer for assistance. He intended to upload the final result to the distributed data store. Assuming that the remote upload would operate asynchronously, the developer added *enough* minutes of sleep to accommodate the upload's completion. Contrary to his assumption, accessing the distributed data store was indeed synchronous. Therefore, this sleeping operation was

[2] https://pytorch.org/docs/1.12/distributed.html

unnecessary and resulted in a conspicuous waste of eight NVIDIA V100 GPUs. The fix to this issue is to remove the extra-sleeping code. In other instances, developers appeared to manage the lifecycles of their jobs manually; nevertheless, they failed to precisely track the completion of deep learning computations, causing delays in GPU release. In general, developers should utilize platforms to manage job lifecycles instead of relying on their own methods.

Platform-X is a dedicated platform designed for deep learning. However, we observed that certain data analysis jobs and traditional machine learning jobs (e.g., classification using scikit-learn [57]) were submitted. Despite developers requesting and being allocated GPUs, these jobs utilized CPUs solely, resulting in *Non-DL Job* issues (4; 0.57%) and causing a severe waste of GPUs. The fix is to submit non-DL jobs to platforms specifically designed for such tasks, rather than using Platform-X. For example, an Apache Spark [81] job should be directed to a big-data analytics platform.

4.2 Dimension 2: Model

There are 319 (45.18%) low-GPU-utilization issues in this dimension, which are related to deep learning models and manifest during the model training and evaluation stages.

The first of the three categories, labeled *Improper Batch Size*, comprises 181 (25.64%) issues. During the training of a model, various hyperparameters govern the training process, including the learning rate and sequence length. Among these, the *batch size*—representing the number of input data items in a single training iteration—exerts a significant influence on model training. Improper batch sizes noticeably decrease the GPU computation of deep learning operators (expressed as the total number of floating-point operations or FLOPs), thereby reducing overall GPU utilization. Another minor issue arises from developers usually using the same batch size for both model training and evaluation, unaware that the amount of GPU computation required for each stage differs obviously. Because model evaluation is less computation-intensive (lacking backpropagation and weight updates), this uniformity could lead to reduced GPU utilization during the evaluation stage. Figure 2 illustrates an example with a small batch size. Identifying an Improper Batch Size issue necessitates that both the average GPU utilization and the peak GPU memory utilization during the model training or evaluation stage are less than or equal to 50%. We introduce this extra condition on GPU memory utilization to eliminate cases where the limited capacity of GPU memory causes low GPU utilization: Developers opt for a small batch size to prevent out-of-GPU-memory exceptions, as larger batch sizes also increase GPU memory consumption (refer to the second paragraph below). This category experiences numerous low-utilization issues, given the challenge for developers to determine the optimal batch size that fully utilizes target GPUs before job submission. This challenge motivates tool development to estimate or predict the GPU utilization of deep learning models [13, 43, 79].

A simple yet common fix is to use reasonably large batch sizes [3, 42, 80] that do not lead to out-of-GPU-memory exceptions during both model training and evaluation. Developers should strive to determine optimal batch sizes for three reasons: (1) Deep learning tasks demand more powerful and abundant GPUs than those utilized in local development. Consequently, developers have the obligation and willingness to fine-tune the batch size, maximizing the use of platform resources; (2) Considering the rapid growth of training data and model size, employing larger batch sizes becomes a pragmatic approach to accomplishing large-scale model training within a reasonable timeframe; (3) Recent research [42, 68, 80] suggests that there is "no evidence that larger batch sizes degrade out-of-sample performance." [68] Note that we are not advocating for an arbitrary increase in batch size without considering the impact on training instability. Developers can leverage familiar techniques such as data shuffling, proper weight initialization, and learning rate scheduling [3] to stabilize training and enhance model learning performance. Additionally, tools such as Microsoft DeepSpeed [65] and NNI [45] offer convenient batch-size autotuning.

```
1 from torch.utils.data import DataLoader
2
3 train_batch_size = 32 384
4 eval_batch_size  = 32 448
5 train_loader = DataLoader(dataset = train_data, batch_size =
  ↪  train_batch_size, shuffle = True)
6 eval_loader  = DataLoader(dataset = eval_data, batch_size =
  ↪  eval_batch_size, shuffle = True)
```

Figure 2: A simplified example of Improper Batch Size issues. "train_batch_size" and "eval_batch_size" are specified arguments for model training and evaluation, respectively. The fix is to increase their values independently (lines 3–4).

The second category is *Insufficient GPU Memory* (22; 3.12%), indicating a limitation in GPU memory capacity that constrains GPU utilization. Compared to main memory, the physical memory of a GPU is notably smaller. When a job consumes excessive GPU memory (e.g., training a large language model), it becomes challenging to augment deep learning computation and GPU utilization (e.g., by enlarging the batch size) due to the potential occurrence of out-of-GPU-memory exceptions. To illustrate, a GPT-2 [62] job with a single NVIDIA V100 GPU exhibited an average GPU utilization of only 40%. However, attempts to increase the batch size were hindered by the imminent depletion of GPU memory, with peak utilization reaching 83%. Detection of an Insufficient GPU Memory issue involves two conditions: (1) Average GPU utilization during model training or evaluation must be less than or equal to 50%; (2) Meanwhile, peak GPU memory utilization should exceed 80%. Developers address this type of issue by either requesting additional GPUs or optimizing GPU memory usage through careful data placement and timely cold data swap-out. Another approach involves leveraging automatic GPU memory management provided by specific DL frameworks and optimization libraries. For example, Microsoft DeepSpeed incorporates the novel Zero Redundancy Optimizer (ZeRO) [64], significantly conserving GPU memory.

Model Checkpointing (116; 16.43%) is the third and final category. Model checkpointing preserves the state of a model (including both the learnable parameters and optimizer condition) to reliable storage, which is crucial for recovering from job failures. However, checkpointing is time-consuming due to the substantial data transfer required from GPU memory to main memory, followed by writing to the distributed data store over the network. As the model size increases, this procedure leads to extended periods of GPU inactivity. For example, a job with a single NVIDIA V100

```
1 import torch
2+import threading
3+import shutil
4+import os
5
6 def main():
7   ...
8   for epoch in range(start_epoch, num_epochs):
9     if step > num_training_steps:
10      break
11    ...
12    for i, batch in enumerate(tqdm(train_loader)):
13      ...
14      logits = model(input_data)
15      ...
16      optimizer.zero_grad()
17      loss.backward()
18      optimizer.step()
19      ...
20      f1, pred = evaluator.evaluate(val_loader, model, step)
21      ...
22      if f1 > max_f1:
23        max_f1 = f1
24        ~~torch.save(model, remote_path)~~
25      + local_path = make_tmp_path(epoch, i)
26      + torch.save(model, local_path)
27
28      + def save_file(local_path, remote_path):
29      +   shutil.copyfile(local_path, remote_path)
30      +   os.remove(local_path)
31
32      + threading.Thread(target = save_file, args = [local_path,
        ↪  remote_path]).start()
33
34 if __name__ == '__main__':
35   main()
```

Figure 3: A simplified example of Model Checkpointing issues. The fix is to save the checkpoint to a local temporary file (lines 25–26), followed by an asynchronous copy to the remote data store in a separate thread (lines 28–32).

GPU saved a new checkpoint at the end of each *epoch* (i.e., full training on the entire input data). Nevertheless, the checkpointing task took an average of 96 seconds, surpassing more than half of the total epoch duration (168 seconds on average). As a result, the average GPU utilization of this job dropped significantly to 39.57%. Moreover, due to the unpredictable nature of failures, DL jobs necessitate frequent checkpointing to minimize recovery costs, which continually disrupts normal GPU computations and slows down model training. A simple yet common fix is to use asynchronous model checkpointing, enabling both GPUs and I/O devices to work in parallel [49, 51]. Figure 3 provides an illustrative example. More advanced checkpointing techniques are discussed in Section 5.2.

4.3 Dimension 3: Data

This is the largest dimension among the four, encompassing 325 (46.03%) low-GPU-utilization issues attributed to diverse data operations. The first five categories pertain to the manipulation of input, output, and model data, while the sixth category relates to data exchange among multiple GPUs in distributed training.

Training a model involves loading input data into GPU memory from its initial residence in main memory. PyTorch developers commonly employ the `torch.utils.data.DataLoader` class [59] to handle input data. Subsequently, they utilize the `torch.Tensor.to`

```
1 import torch
2 from torch.utils.data import DataLoader
3
4 train_loader = DataLoader(train_set,..., num_workers=8,
  ↪  pin_memory=True)
5 eval_loader = DataLoader(eval_set, ..., num_workers=8,
  ↪  pin_memory=True)
6
7 for epoch in range(num_epochs):
8   ...
9   for _, data in enumerate(train_loader, 0):
10    # get the inputs
11    inputs, labels = data
12    inputs = inputs.to(device, non_blocking=True)
13    labels = labels.to(device, non_blocking=True)
```

Figure 4: A simplified example of Inefficient Host-GPU Data Transfer issues. The fix is to enable automatic memory pinning by setting the `pin_memory` parameter to `True` (lines 3–4). We further set the `non_blocking` parameter to `True` (lines 11–12), which tries asynchronous data transfer if possible.

function[3] to transfer input tensors from main memory to GPU memory. However, the default usage of `DataLoader` disables *automatic memory pinning* [59], resulting in increased data transfer latency, continual fluctuations in GPU utilization, and the emergence of *Inefficient Host-GPU Data Transfer* issues. This category comprises 197 (27.90%) issues, accounting for the largest among all the fifteen categories. Note that these issues are currently confined to PyTorch jobs, as TensorFlow manages host-GPU data transfer transparently, leaving developers with no direct control over it. Figure 4 provides an illustrative example. An effective fix is to set the `pin_memory` parameter of `DataLoader` to `True` (lines 3–4), which allows input data to be loaded into locked main memory pages, thereby expediting subsequent host-GPU data transfer [70]. Developers can also consider setting the `non_blocking` parameter of `torch.Tensor.to` to `True` (lines 11–12), in which PyTorch "tries to transfer asynchronously with respect to the host if possible." [59] Moreover, bulk transfer (i.e., transferring multiple input tensors at once) and tensor prefetching can further enhance transfer efficiency.

Developers usually engage in preprocessing raw input data, such as removing erroneous or corrupted data points. Another essential aspect is dataset augmentation [18]: Developers flip, crop, or rotate input images in computer vision tasks to expand the training dataset, thereby enhancing model learning performance. Despite its significance, data preprocessing predominantly relies on CPUs rather than GPUs, leading to *Data Preprocessing* issues (28; 3.97%). A common fix is to preprocess raw input data in advance through a dedicated data analysis job. Another approach for specific data types, such as images, audio, or videos, involves leveraging the NVIDIA Data Loading Library (DALI)[4], which allows developers to expedite preprocessing with GPUs. Figure 5 demonstrates the utilization of DALI to address a low-utilization issue.

Remote Data Read (18; 2.55%) denotes that a DL job accesses and retrieves input data directly from the distributed data store. Due to the relatively sluggish network transfers, GPUs intermittently pause while waiting for input. Even worse, these remote files will be repeatedly accessed if the job cannot accommodate the entire

[3] https://pytorch.org/docs/1.12/generated/torch.Tensor.to.html
[4] https://developer.nvidia.com/dali

```
+from nvidia.dali import *
+@pipeline_def(batch_size=..., num_threads=..., device_id=...)
+def data_pipeline(path):
+ jpegs, labels = fn.readers.file(file_root=path)
+ images = fn.decoders.image(jpegs, ...)
+ flipped = fn.flip(images, ...)
+ return flipped, labels

for image_path in files:
  ...
  feat_extract.main(image_path = image_path, ...)
+ pipe = data_pipeline(image_path)
+ pipe.build()
+ pipe.run()
  ...
```

Figure 5: A simplified example of Data Preprocessing issues. The fix is to leverage the NVIDIA Data Loading Library, utilizing GPUs instead of CPUs for faster data preprocessing.

```
1 # CUDA_VISIBILE_DEVICES=0,1 python -u train.py ...
2 import torch
3 # Main trainer:
4 class Trainer:
5   def __init__(self, model, train_dataset, test_dataset, config):
6     self.model = model
7     self.train_dataset = train_dataset
8     self.eval_dataset = eval_dataset
9     self.config = config
10
11    # take over whatever gpus are on the system
12    self.device = 'cpu'
13    if torch.cuda.is_available():
14      self.device = torch.cuda.current_device()
15 +    if torch.cuda.device_count() > 1:
16 +      self.model = torch.nn.DataParallel(self.model)
17      self.model = self.model.to(self.device)
```

Figure 6: A simplified example of API Misuse issues. The data-parallel job employs just one of the two assigned GPUs. The fix is to encapsulate the model within the `DataParallel` class (lines 17–18) to fully utilize both GPUs.

input data in main memory because of excessive data volume or limited memory capacity. A common practice developers adopt is to download remote input files to the local storage of worker machines before initiating model training. A more efficient approach, as discussed in Section 5.2, involves pipelining the sequence of data copying and model training, which enables training to commence as soon as partial input data becomes available.

DL jobs can rely on external data sources for their routine work. For example, when utilizing the Hugging Face Transformers library [75] for continuous training, developers may require datasets and model files housed within Hugging Face's own repository. However, because accessing external sites tends to be less reliable and slower than the internal storage, these jobs could significantly slow down, resulting in *External Data Usage* issues (18; 2.55%). The fix is to pre-upload the external data into the internal distributed data store before job submission.

Intermediate Result Upload (14; 1.98%) indicates the frequent uploading of intermediate training results in a DL job to the distributed data store, including regular validation outputs or trial statistics from an AutoML experiment. All these issues involve synchronous remote file uploads and share similarities with those found in the Model Checkpointing category. Consequently, the fix involves a similar approach: Developers create an asynchronous task to upload intermediate results.

In a distributed job, GPUs consistently share data such as gradients and output tensors. This data exchange between GPUs, whether through the network or PCI Express bus, frequently interrupts their designated tasks and causes a sudden drop in GPU utilization to zero. Within the *Data Exchange* category, there are 50 (7.08%) issues. A common fix is to enhance communication efficiency by minimizing the frequency of data exchange (e.g., using large batch sizes) and enabling compressed communication [34, 39]. For users of Horovod [67], enlarging the `backward_passes_per_step` parameter[5] can help accumulate more local gradient updates and transmit them simultaneously. Developers can also leverage Microsoft DeepSpeed's 3D (data, model, and pipeline) parallelism, whose 1-bit Adam and 0/1 Adam [39] optimizers demonstrate significant reductions in communication volume.

[5] https://horovod.readthedocs.io/en/stable/api.html

4.4 Dimension 4: Library

This dimension encompasses 28 (3.97%) low-GPU-utilization issues attributed to dependent libraries and DL frameworks. These issues typically lead to prolonged inactive periods or even non-use of one or more GPUs. There are two categories within this dimension.

Long Library Installation (12; 1.70%) refers to issues where a DL job expends considerable time, at least 10 minutes, installing necessary libraries during its initialization phase. To illustrate, a job took approximately 16 minutes simply to transfer an extensive folder of library code from the distributed data store to the worker machine. Throughout this period, the assigned NVIDIA V100 GPU remained inactive, without any workload. Platform-X has indeed offered official DL Docker images to establish a hermetic environment for job execution. Moreover, it supports custom Docker images that come pre-installed with all required libraries. Despite these provisions, we observed that a majority of jobs continue to handle their libraries manually by invoking "`pip install`" for standard Python packages, cloning library code from the Internet, or directly copying code folders from the distributed storage. This tendency might be attributed to the rapid evolution of deep learning toolchains, making the maintenance of custom Docker images quite challenging. We propose that developers proactively construct custom Docker images encompassing all requisite libraries to reduce waiting time and prevent library installation failures [82].

API Misuse issues (16; 2.27%) arise from the incorrect usage of intricate framework/library APIs. We observed several instances where jobs set erroneous values to `CUDA_VISIBLE_DEVICES`, an environment variable responsible for managing GPU visibility in CUDA applications. For example, a job configured it with an empty string. As a result, no allocated GPUs were detected, compelling the job to execute on CPUs for around 481 minutes. These developers might duplicate startup Shell scripts from other sources, such as their local development setups, while neglecting to update `CUDA_VISIBLE_DEVICES` before job submission. Figure 6 illustrates another API Misuse issue, where a data-parallel job employs just one of the two assigned GPUs. The code in line 17 invokes the `torch.nn.Module.to` function to load the model into GPU memory. However, when multiple GPUs are available, the correct

Table 2: Experiments on large batch sizes.

Model	BZ	Average GPU Utilization	Peak GPU Mem Utilization	Speedup
BERT (large-uncased)	32	55.21% (64.40%)	19.46%	1.00
	64	55.52% (70.43%)	22.03%	1.96
	128	54.91% (79.40%)	27.92%	3.10
	256	62.38% (85.59%)	39.15%	4.16
	512	57.32% (88.48%)	61.53%	4.67
	768	59.06% (90.78%)	84.87%	4.92
	1024	N/A: OOM	N/A: OOM	N/A: OOM
Swin Transformer	8	53.40% (57.79%)	13.95%	1.00
	16	55.56% (63.80%)	16.49%	1.79
	32	61.00% (73.90%)	21.14%	2.51
	64	61.65% (79.36%)	29.97%	2.94
	128	62.32% (82.46%)	48.12%	3.32
	256	61.81% (85.05%)	84.86%	3.41
	384	N/A: OOM	N/A: OOM	N/A: OOM

Note: "BZ" denotes batch size, while "OOM" indicates that the job ran out of GPU memory and crashed. The numbers inside the parentheses represent the average GPU utilization during the model training stage.

approach involves initially utilizing the `torch.nn.DataParallel` class[6] (lines 15–16). This class encapsulates the model and automatically parallelizes the training on all GPUs by "splitting the input across the specified devices by chunking in the batch dimension."[6]

4.5 Evaluation of Proposed Fixes

We conducted many experiments on real deep learning jobs to validate the fixes. Upon applying these fixes, we observed that the overall GPU utilization of all experimental jobs improved. In this section, we demonstrate the effectiveness of addressing Improper Batch Size, Model Checkpointing, and Inefficient Host-GPU Data Transfer issues through large batch sizes, asynchronous model checkpointing, and automatic memory pinning, respectively. The selection of these three fixes is based on two primary reasons:

(1) They effectively resolve the top three categories of low-GPU-utilization issues (refer to Table 1).
(2) The particular issues are fairly common in deep learning development and impact the entire model training stage.

In our experiments on Platform-X, we chose two representative real-world jobs. The first one trained BERT [11], a transformer-based model for natural language processing, with a batch size of 32. The second job trained Swin Transformer [38], a general-purpose, transformer-based model for computer vision "whose representation is computed with shifted windows." [38] It used a batch size of 8. In both cases, a new model checkpoint was synchronously saved to the distributed storage per epoch. The BERT job initially had automatic memory pinning enabled by default; however, we deactivated this feature to establish a baseline for comparison.

We re-ran the two jobs with their original settings (e.g., both utilizing eight NVIDIA V100 32GB GPUs). Subsequently, we measured the average GPU utilization during the entire lifecycle, average GPU utilization specifically during model training, peak GPU memory usage, and job execution speedup. To determine the speedup, we computed $\frac{T_{orig}}{T_{fixed}}$, where T_{orig} and T_{fixed} represented the total execution time of the original and fixed jobs, respectively.

[6] https://pytorch.org/docs/1.12/generated/torch.nn.DataParallel.html

Table 3: Experiments on asynchronous model checkpointing.

Model	BZ	Ckpt Mode	Average GPU Utilization	Speedup
BERT (large-uncased)	32	Sync	55.21% (64.40%)	1.00
		Async	60.96% (63.65%)	1.20
	768	Sync	59.06% (90.78%)	4.92
		Async	83.74% (90.69%)	7.48
Swin Transformer	8	Sync	53.40% (57.79%)	1.00
		Async	53.45% (57.81%)	1.02
	256	Sync	61.81% (85.05%)	3.41
		Async	56.66% (82.17%)	3.82

Note: "Ckpt" denotes checkpointing.

Table 4: Experiments on automatic memory pinning.

Model	BZ	Memory Pinning	Average GPU Utilization	Speedup
BERT (large-uncased)	32	False	55.21% (64.40%)	1.00
		True	56.25% (63.89%)	1.15
	768	False	59.06% (90.78%)	4.92
		True	55.69% (92.06%)	5.01
Swin Transformer	8	False	53.40% (57.79%)	1.00
		True	54.82% (59.30%)	1.03
	256	False	61.81% (85.05%)	3.41
		True	62.64% (86.50%)	3.50

The experimental results in Table 2 demonstrate the positive impact of larger batch sizes on GPU utilization and job execution speed. As we incremented batch sizes, both jobs eventually maxed out GPU memory and crashed. Employing previously mentioned techniques, such as proper weight initialization and learning rate scheduling [3], helped stabilize training without compromising model learning performance (e.g., F1 score [9]). For the BERT job, average GPU utilization during the model training stage surged from 64.40% to 90.78%, resulting in a noteworthy speedup of up to 4.92X. Similarly, the Swin Transformer job experienced a speedup of up to 3.41X, with average GPU utilization climbing from 57.79% to 85.05% during model training. Due to significantly shortened training and evaluation stages, GPU operations became less and less dominant in the total execution time. Interestingly, the overall average GPU utilization of both jobs did not appear to grow steadily; instead, it even decreased slightly. This suggests the potential for optimization by enhancing non-GPU operations and stages.

Table 3 illustrates the advantages of asynchronous model checkpointing in enhancing GPU utilization and runtime performance. The BERT job experienced a significant boost due to its large model size and extended checkpoint upload. Moreover, this enhancement became more noticeable when employing a larger batch size, as the training time per epoch was considerably reduced, amplifying the proportion of model checkpointing. Table 4 demonstrates that both jobs also benefit from automatic memory pinning. Even with a larger batch size, the improvement is still evident due to a considerable reduction in training time.

By integrating these three fixes, the overall improvement becomes substantially more notable. Table 5 indicates that the BERT job achieves a remarkable final speedup of 7.52X, while the Swin Transformer job attains a final speedup of 3.95X.

Table 5: Experiments integrating large batch sizes, asynchronous model checkpointing, and automatic memory pinning.

Model	BZ	Ckpt Mode	Memory Pinning	Average GPU Utilization	Speedup
BERT (large-uncased)	32	Sync	False	55.21% (64.40%)	1.00
	768	Async	True	85.78% (91.90%)	7.52
Swin Transformer	8	Sync	False	53.40% (57.79%)	1.00
	256	Async	True	58.61% (87.50%)	3.95

5 DISCUSSION

5.1 Generality of Our Study

While our research is conducted exclusively within Microsoft, we believe that the findings on low GPU utilization are universal and applicable to other deep learning platforms like Microsoft Azure Machine Learning [47], Amazon SageMaker [2], and Google Vertex AI [20]. This perspective is grounded in the broad similarities between Platform-X and alternative platforms, primarily in two key aspects. Firstly, the programs of our jobs follow a standard deep learning programming paradigm, utilizing the identical Python language, DL algorithms, frameworks, and associated libraries. Moreover, they target typical DL applications, such as natural language processing and image recognition. Secondly, the system architecture, computing hardware, and software stack of Platform-X are widely embraced [4, 28, 77]; our job management mechanism, including submission, scheduling, and execution, aligns with practices employed elsewhere. Consequently, DL jobs across diverse platforms could experience the same low-GPU-utilization issues.

As a case in point, certain issues arise from the characteristics of GPUs and deep learning frameworks, making them applicable across various platforms. GPU memory, notably smaller than main memory, imposes limitations on the amount of data it can accommodate. Consequently, the current lack of a transparent and effective mechanism within GPUs and frameworks for data placement and swapping can result in Inefficient Host-GPU Data Transfer and Insufficient GPU Memory issues. Additionally, the intricacies of deep learning computations are concealed within frameworks and proprietary NVIDIA CUDA/cuBLAS/cuDNN APIs. This introduces quite a challenge in determining optimal batch sizes, other hyperparameter values, and neural architectures that align well with GPU computing capacities before job submission.

As another case, specific issues stem from the environmental differences between local development machines and platforms [82]. Typically, developers execute their programs interactively for local development. However, running interactive jobs becomes unaffordable due to the preciousness of platform resources. These environmental discrepancies can also cause library-related issues. For example, developers may need to install dependent libraries not included in the standard deep learning Docker images. Meanwhile, the allocated GPUs have to remain completely idle.

We observe that Google's performance and cost optimization guide for machine learning [19] corroborates a number of low-GPU-utilization issues and proposes similar fixes, affirming the generality of our study. For example, user-managed notebooks (a type of interactive job) "should be switched off or deleted unless they are really running experiments." [19] Additionally, the guide recommends that developers "preprocess the input data once and save it as a TFRecord file," [19] effectively resolving Data Preprocessing issues. The TFRecord format stores processed data in binary form and improves runtime performance. Developers can also leverage TensorFlow Transform [71] to establish flexible pipelines, enabling model training to proceed without waiting for data preprocessing to complete.

5.2 Future Research Directions

This section proposes potential future research directions based on our empirical study. In conjunction with the study findings, these suggestions aim to assist developers and platforms in optimizing GPU utilization and addressing unique software engineering challenges in the deep learning domain.

Tool Support.

Estimation and Prediction of GPU Utilization. To comprehend how GPUs are utilized, developers currently rely on their domain expertise or submit numerous jobs with diverse neural architectures and hyperparameter values. This process is both technically demanding and resource- and time-consuming. An alternative approach involves developing an estimator that infers GPU utilization and other runtime metrics (e.g., GPU memory consumption [15] and execution time [60]) for DL models. This can be achieved by constructing a comprehensive analytic cost model derived from the actual deep learning computations and GPU computing capacity. Additionally, we can train predictive models [13, 43, 79] using historical GPU utilization data. For example, DNNPerf [13] adopts the Graph Neural Network (GNN) to predict the execution time and GPU memory consumption of training jobs; this methodology can be adapted for GPU utilization. Estimation and prediction tools play a crucial role in assisting developers in selecting optimal hyperparameter values and neural architectures that fully exploit the capabilities of target GPUs across various model configurations. Furthermore, when combined with satisfiability modulo theories (SMT) solvers [14, 17], such tools enhance practicability by efficiently handling many more candidate configurations within the real constraints of computational resources.

Code Advisor. Program analysis proves to be a valuable approach for software defect detection by statically analyzing source code without execution. Code advisors that perform program analysis on deep learning scripts, programs, and configuration files can proactively uncover many types of low-GPU-utilization issues before job submission, including categories such as GPU Oversubscription, Inefficient Host-GPU Data Transfer, Remote Data Read, and Model Checkpointing. Our findings indicate that most (84.99%) issues could be fixed with a small number of code/script modifications. Consequently, code advisors have the potential to offer fix suggestions and automated program repairs to developers.

Efficient Model Checkpointing. DL jobs necessitate model checkpointing to recover from both hardware and software failures. Nevertheless, frequent checkpointing and synchronous uploading to a distributed data store, particularly for large language models, can lead to prolonged and recurrent GPU idle periods. To enhance the efficiency of model checkpointing, we can employ various techniques such as asynchrony [49, 51], frequency adaptivity [33, 49], incremental checkpointing [12], data compression [32, 78], high-performance serialization [63, 73], and distributed caching [35].

Platform Improvement.

Heterogeneous Pipelining. Currently, on platforms, GPU allocation occurs all at once at the beginning [55] and remains unchanged until job completion. Our observations indicate that many low-GPU-utilization issues stem from time-consuming synchronous data processes, such as download, preprocessing, and upload. Additionally, fewer GPUs are required during model evaluation compared to training. The static allocation of GPUs hinders efficient resource reclamation and causes prolonged periods of GPU inactivity, leading to substantial GPU waste and noticeable reductions in GPU utilization. A more effective approach involves restructuring a single job as a pipeline of heterogeneous tasks and assigning GPUs solely and separately to model training and evaluation. As a result, data preprocessing can be managed by CPU-centric big-data tasks (e.g., via Apache Spark), with the platform transferring processed data in either batch or streaming mode to subsequent GPU-based tasks. This adaptable GPU allocation method significantly enhances overall GPU utilization on deep learning platforms. Furthermore, heterogeneous pipelining simplifies data sharing across multiple jobs; in AutoML training, one data-preprocessing task can distribute outputs to numerous trial tasks simultaneously, ensuring that the same input data is processed just once.

GPU Sharing. The NVIDIA Multi-Instance GPU (MIG)[7] technique partitions a GPU (e.g., A30, A100, and H100) into a number of isolated instances, each equipped with its "own high-bandwidth memory, cache, and compute cores."[7] Platforms can integrate MIG into their job schedulers to enhance overall GPU utilization by dynamically packing a number of low-GPU-utilization jobs across various instances within a group of shared GPUs.

Distributed Data Caching. Distributed training for large models has recently gained popularity. However, transferring input data from distributed storage to each worker machine might not be affordable or even feasible due to the vast amount of data. To address this challenge, employing a distributed data cache dedicated to DL workloads like SiloD [85] can be beneficial in expediting training and reducing resource waste. This becomes particularly significant in AutoML scenarios, where multiple trial jobs of an experiment utilize the same input data. Moreover, similar to established big-data platforms such as Apache Spark [81] and Microsoft Cosmos [58], DL platforms should align with frameworks to comprehend how jobs interact with data, thereby optimizing cache performance.

6 RELATED WORK

Prior research has primarily concentrated on modeling, estimating, and predicting CPU utilization [5, 22, 27, 50, 74]. However, these efforts could not sufficiently support developers in enhancing GPU utilization due to various fundamental differences between CPUs and GPUs. Additionally, a number of researchers target performance issues [23, 24, 30, 36, 37, 52, 53]. Jin et al. [30] conducted a comprehensive study on 110 real-world performance bugs randomly selected from five software suites. They derived efficiency principles from patches to enhance performance bug detection. Nistor et al. [53] and Nistor [52] investigated how developers identified, reported, and resolved performance bugs. PCatch [36] accurately predicted performance cascading bugs in representative distributed systems, based on executions with small-scale workloads. While the previous studies may not have directly addressed deep learning, they do provide valuable insights and guidelines for enhancing the overall runtime performance of DL jobs.

In recent empirical studies on deep learning [7, 16, 25, 26, 28, 29, 69, 82–84], many researchers have focused on investigating the failures and defects in DL programs, jobs, compilers, and frameworks. Zhang et al. [82] examined the program failures of 4,960 DL jobs from Microsoft's internal platform, while Shen et al. [69] analyzed 603 bugs across three popular DL compilers and developed a fuzzer for testing Apache TVM [8]. Meanwhile, Jeon et al. [28] investigated low GPU utilization in large-scale, multi-tenant GPU clusters dedicated to deep learning training. They identified resource locality, gang scheduling, and job failures as the root causes from a platform perspective and proposed enhancements for future cluster schedulers. Their research primarily concentrated on the GPU utilization of clusters instead of individual jobs. Cao et al. [7] characterized 224 performance problems in TensorFlow and Keras programs based on 210 Stack Overflow posts. This work revealed similarities to certain issues encountered in our study, such as those stemming from inappropriate batch sizes. While the investigation of performance issues in DL programs is becoming a domain of active exploration, our study delves into the topic of low GPU utilization in industrial DL jobs. These jobs are distinct from prior study subjects (typically reported in online posts) in various dimensions, including execution mode, DL algorithm, job scale, and hardware specification. Consequently, we unveil a range of unique low-GPU-utilization issues and fresh insights (such as those related to job submission and hardware specification) arising in a cloud-based deep learning platform, yet unexplored by previous research.

Several techniques have been proposed to enhance the GPU utilization and runtime performance of DL programs and jobs. Nicolae et al. [51] introduced asynchronous model checkpointing to conceal serialization and I/O overhead by distributing the workload among all processes. Wu et al. [76] implemented EDL to enable elastic model training through dynamic adjustment of GPU numbers for optimal efficiency. Rammer [40] presented a novel DL compiler design on the automatic optimization of job execution by "exploiting parallelism through inter- and intra- operator co-scheduling." [40] Microsoft DeepSpeed [65] is an optimization library built atop PyTorch, achieving extreme-scale and efficient distributed training. While these techniques address specific low-GPU-utilization issues, our study indicates the need for a comprehensive solution to optimize overall GPU utilization in deep learning jobs.

7 CONCLUSION

This paper presents a comprehensive empirical study on low GPU utilization of deep learning jobs. We randomly selected 400 real jobs from Platform-X and discovered 706 low-GPU-utilization issues attributed to the code logic of the jobs' scripts and programs. Furthermore, we manually identified the common root causes and suggested corresponding fixes. Based on the findings, we proposed potential research topics to enhance the GPU utilization of jobs. We believe that our work provides valuable guidelines for the future development of deep learning programs and platforms.

[7] https://www.nvidia.com/en-us/technologies/multi-instance-gpu

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283.

[2] Amazon. 2023. Amazon SageMaker. https://aws.amazon.com/sagemaker.

[3] Weights & Biases. 2023. Current Best Practices for Training LLMs from Scratch. https://wandb.ai/site/llm-whitepaper.

[4] Scott Boag, Parijat Dube, Benjamin Herta, Waldemar Hummer, Vatche Ishakian, K JAYARAM, Michael Kalantar, Vinod Muthusamy, Priya NAG-PURKAR, and Florian Rosenberg. 2017. Scalable multi-framework multi-tenant lifecycle management of deep learning training jobs. In *Workshop on ML Systems, NIPS*.

[5] Samira Briongos, Pedro Malagón, José L. Risco, and José M. Moya. 2017. Building Accurate Models to Determine the Current CPU Utilization of a Host within a Virtual Machine Allocated on It. In *Proceedings of the Summer Simulation Multi-Conference* (Bellevue, Washington) *(SummerSim '17)*. Society for Computer Simulation International, San Diego, CA, USA, Article 33, 12 pages.

[6] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Commun. ACM* 59, 5 (apr 2016), 50–57.

[7] Junming Cao, Bihuan Chen, Chao Sun, Longjie Hu, Shuaihong Wu, and Xin Peng. 2022. Understanding Performance Problems in Deep Learning Systems. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 357–369.

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, Carlsbad, CA, 578–594.

[9] Nancy Chinchor. 1992. MUC-4 Evaluation Metrics. In *Proceedings of the 4th Conference on Message Understanding* (McLean, Virginia) *(MUC4 '92)*. Association for Computational Linguistics, USA, 22–29.

[10] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20, 1 (1960), 37.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186.

[12] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. 2022. Check-N-Run: a Checkpointing System for Training Deep Learning Recommendation Models. In *19th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, Renton, WA, 929–943.

[13] Yanjie Gao, Xianyu Gu, Hongyu Zhang, Haoxiang Lin, and Mao Yang. 2023. Runtime Performance Prediction for Deep Learning Models with Graph Neural Network. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 368–380.

[14] Yanjie Gao, Zhengxian Li, Haoxiang Lin, Hongyu Zhang, Ming Wu, and Mao Yang. 2022. REFTY: Refinement Types for Valid Deep Learning Models. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 1843–1855.

[15] Yanjie Gao, Yu Liu, Hongyu Zhang, Zhengxian Li, Yonghao Zhu, Haoxiang Lin, and Mao Yang. 2020. Estimating GPU Memory Consumption of Deep Learning Models. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1342–1352.

[16] Yanjie Gao, Xiaoxiang Shi, Haoxiang Lin, Hongyu Zhang, Hao Wu, Rui Li, and Mao Yang. 2023. An Empirical Study on Quality Issues of Deep Learning Platform. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 455–466.

[17] Yanjie Gao, Yonghao Zhu, Hongyu Zhang, Haoxiang Lin, and Mao Yang. 2021. Resource-Guided Configuration Space Reduction for Deep Learning Models. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 175–187.

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[19] Google. 2022. Best Practices for Performance and Cost Optimization for Machine Learning. http://web.archive.org/web/20220521055530/https://cloud.google.com/architecture/best-practices-for-ml-performance-cost.

[20] Google. 2023. Google Vertex AI. https://cloud.google.com/vertex-ai.

[21] Jiazhen Gu, Huan Liu, Yangfan Zhou, and Xin Wang. 2017. DeepProf: Performance Analysis for Deep Learning Applications via Mining GPU Execution Patterns. *CoRR* abs/1707.03750 (2017). arXiv:1707.03750

[22] Hugo Lewi Hammer, Anis yazidi, and Kyrre Begnum. 2016. Reliable Modeling of CPU Usage in an Office Worker Environment. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing* (Pisa, Italy) *(SAC '16)*. Association for Computing Machinery, New York, NY, USA, 480–483.

[23] Xue Han, Daniel Carroll, and Tingting Yu. 2019. Reproducing performance bug reports in server applications: The researchers' experiences. *Journal of Systems and Software* 156 (2019), 268–282.

[24] Xue Han and Tingting Yu. 2016. An Empirical Study on Performance Bugs for Highly Configurable Software Systems. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Ciudad Real, Spain) *(ESEM '16)*. Association for Computing Machinery, New York, NY, USA, Article 23, 10 pages.

[25] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. 2021. Characterization and Prediction of Deep Learning Workloads in Large-Scale GPU Datacenters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 104, 15 pages.

[26] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 510–520.

[27] Deepak Janardhanan and Enda Barrett. 2017. CPU workload forecasting of machines in data centers using LSTM recurrent neural networks and ARIMA models. In *2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)*. 55–60.

[28] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, unjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 947–960.

[29] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. 2020. An Empirical Study on Bugs Inside TensorFlow. In *Database Systems for Advanced Applications: 25th International Conference, DASFAA 2020, Jeju, South Korea, September 24–27, 2020, Proceedings, Part I* (Jeju, Korea (Republic of)). Springer-Verlag, Berlin, Heidelberg, 604–620.

[30] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-World Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 77–88.

[31] Jupyter. 2023. Project Jupyter. https://jupyter.org.

[32] Fabian Knorr, Peter Thoman, and Thomas Fahringer. 2021. Ndzip-Gpu: Efficient Lossless Compression of Scientific Floating-Point Data on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 93, 14 pages.

[33] Zhiling Lan and Yawei Li. 2008. Adaptive Fault Management of Parallel Applications for High-Performance Computing. *IEEE Trans. Comput.* 57, 12 (dec 2008), 1647–1660.

[34] Conglong Li, Ammar Ahmad Awan, Hanlin Tang, Samyam Rajbhandari, and Yuxiong He. 2022. 1-bit LAMB: Communication Efficient Large-Scale Large-Batch Training with LAMB's Convergence Speed. In *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 272–281.

[35] Haoyuan Li. 2018. *Alluxio: A Virtual Distributed File System*. Ph. D. Dissertation. EECS Department, University of California, Berkeley.

[36] Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S. Gunawi, Xiaohui Gu, Xicheng Lu, and Dongsheng Li. 2018. PCatch: Automatically Detecting Performance Cascading Bugs in Cloud Systems. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 7, 14 pages.

[37] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE '14)*. Association for Computing Machinery, New York, NY, USA, 1013–1024.

[38] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. 2021. Swin Transformer: Hierarchical Vision Transformer using Shifted Windows. In *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*. 9992–10002.

[39] Yucheng Lu, Conglong Li, Minjia Zhang, Christopher De Sa, and Yuxiong He. 2023. Maximizing Communication Efficiency for Large-scale Training via 0/1 Adam. In *The Eleventh International Conference on Learning Representations*.

[40] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*

(OSDI '20). USENIX Association, 881–897.
[41] Silverio Martinez-Fernandez, Justus Bogner, Xavier Franch, Marc Oriol, Julien Siebert, Adam Trendowicz, Anna Maria Vollmer, and Stefan Wagner. 2022. Software Engineering for AI-Based Systems: A Survey. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 37e (apr 2022), 59 pages.
[42] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. 2018. An Empirical Model of Large-Batch Training. *CoRR* abs/1812.06162 (2018).
[43] Hengquan Mei, Huaizhi Qu, Jingwei Sun, Yanjie Gao, Haoxiang Lin, and Guangzhong Sun. 2023. GPU Occupancy Prediction of Deep Learning Models Using Graph Neural Network. In *2023 IEEE International Conference on Cluster Computing (CLUSTER)*. 318–329.
[44] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (mar 2014).
[45] Microsoft. 2018. NNI (Neural Network Intelligence): an open source AutoML toolkit for AutoML lifecycle. https://github.com/microsoft/nni.
[46] Microsoft. 2023. AzureML Large Scale Deep Learning Best Practices. https://github.com/Azure/azureml-examples/tree/main/best-practices/largescale-deep-learning.
[47] Microsoft. 2023. Microsoft Azure Machine Learning. https://azure.microsoft.com/en-us/services/machine-learning-service.
[48] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. 2020. NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis. In *ECCV*.
[49] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 203–216.
[50] Langston Nashold and Rayan Krishnan. 2020. Using LSTM and SARIMA Models to Forecast Cluster CPU Usage. *CoRR* abs/2007.08092 (2020). arXiv:2007.08092
[51] Bogdan Nicolae, Jiali Li, Justin M. Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. 2020. DeepFreeze: Towards Scalable Asynchronous Checkpointing of Deep Learning Models. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. 172–181.
[52] Adrian Nistor. 2014. *Understanding, detecting, and repairing performance bugs*. Ph. D. Dissertation. University of Illinois at Urbana-Champaign. https://mir.cs.illinois.edu/marinov/publications/Nistor14PhD.pdf
[53] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. Caramel: Detecting and Fixing Performance Problems That Have Non-Intrusive Fixes. In *Proceedings of the 37th International Conference on Software Engineering* (Florence, Italy) *(ICSE '15)*. IEEE Press, 902–912.
[54] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. fairseq: A Fast, Extensible Toolkit for Sequence Modeling. In *Proceedings of NAACL-HLT 2019: Demonstrations*.
[55] J.K. Ousterhout. 1982. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*. 22–30.
[56] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, Vol. 32. Curran Associates, Inc., 8024–8035.
[57] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12, 85 (2011), 2825–2830.
[58] Conor Power, Hiren Patel, Alekh Jindal, Jyoti Leeka, Bob Jenkins, Michael Rys, Ed Triou, Dexin Zhu, Lucky Katahanas, Chakrapani Bhat Talapady, Joshua Rowe, Fan Zhang, Rich Draves, Marc Friedman, Ivan Santa Maria Filho, and Amrish Kumar. 2021. The Cosmos Big Data Platform at Microsoft: Over a Decade of Progress and a Decade to Look Forward. *Proc. VLDB Endow.* 14, 12 (jul 2021), 3148–3161.
[59] PyTorch. 2022. Data Loading Utility. https://pytorch.org/docs/1.12/data.html.
[60] Hang Qi, Evan R. Sparks, and Ameet Talwalkar. 2017. Paleo: A Performance Model for Deep Neural Networks. In *Proceedings of ICLR*.
[61] Björn Rabenstein and Julius Volz. 2015. Prometheus: A Next-Generation Monitoring System (Talk). In *SREcon15 Europe*. USENIX Association, Dublin.
[62] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).
[63] Deepti Raghavan, Philip Levis, Matei Zaharia, and Irene Zhang. 2021. Breakfast of Champions: Towards Zero-Copy Serialization with NIC Scatter-Gather. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 199–205.
[64] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory Optimizations toward Training Trillion Parameter Models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 20, 16 pages.
[65] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 3505–3506.
[66] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13.
[67] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *CoRR* abs/1802.05799 (2018). arXiv:1802.05799
[68] Chris Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-dickstein, Roy Frostig, and George Dahl. 2018. Measuring the Effects of Data Parallelism on Neural Network Training. *Journal of Machine Learning Research (JMLR)* (2018).
[69] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A Comprehensive Study of Deep Learning Compiler Bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 968–980.
[70] StackOverflow. 2011. Why is CUDA pinned memory so fast? https://stackoverflow.com/questions/5736968/why-is-cuda-pinned-memory-so-fast.
[71] TensorFlow. 2023. Get Started with TensorFlow Transform. https://www.tensorflow.org/tfx/transform/get_started.
[72] Neil C. Thompson, Kristjan H. Greenewald, Keeheon Lee, and Gabriel F. Manso. 2020. The Computational Limits of Deep Learning. *CoRR* abs/2007.05558 (2020).
[73] Kenton Varda et al. 2013. Cap'n Proto serialization/RPC system - core tools and C++ library. https://github.com/capnproto/capnproto.
[74] Thomas Wang, Simone Ferlin, and Marco Chiesa. 2021. Predicting CPU Usage for Proactive Autoscaling. In *Proceedings of the 1st Workshop on Machine Learning and Systems* (Online, United Kingdom) *(EuroMLSys '21)*. Association for Computing Machinery, New York, NY, USA, 31–38.
[75] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. 2019. HuggingFace's Transformers: State-of-the-art Natural Language Processing. *CoRR* abs/1910.03771 (2019). arXiv:1910.03771
[76] Yidi Wu, Kaihao Ma, Xiao Yan, Zhi Liu, Zhenkun Cai, Yuzhen Huang, James Cheng, Han Yuan, and Fan Yu. 2022. Elastic Deep Learning in Multi-Tenant GPU Clusters. *IEEE Transactions on Parallel and Distributed Systems* 33, 1 (2022), 144–158.
[77] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, USA, 595–610.
[78] Yifan Yang, Joel S. Emer, and Daniel Sanchez. 2021. SpZip: Architectural Support for Effective Data Compression in Irregular Applications. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE Press, 1069–1082.
[79] Gingfung Yeung, Damian Borowiec, Adrian Friday, Richard Harper, and Peter Garraghan. 2020. Towards GPU Utilization Prediction for Cloud Deep Learning. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association.
[80] Yang You, Igor Gitman, and Boris Ginsburg. 2017. Scaling SGD Batch Size to 32K for ImageNet Training. *CoRR* abs/1708.03888 (2017). arXiv:1708.03888
[81] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65.
[82] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. 2020. An Empirical Study on Program Failures of Deep Learning Jobs. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 1159–1170.
[83] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. 2019. An Empirical Study of Common Challenges in Developing Deep Learning Applications. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. 104–115.
[84] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An Empirical Study on TensorFlow Program Bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 129–140.
[85] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Mingxia Li, Fan Yang, Qianxi Zhang, Binyang Li, Yuqing Yang, Lili Qiu, Lintao Zhang, and Lidong Zhou. 2023. SiloD: A Co-Design of Caching and Scheduling for Deep Learning Clusters. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 883–898.