Check for updates

FastFlow: Accelerating Deep Learning Model Training with Smart Offloading of Input Data Pipeline

Taegeon Um*
Samsung Research
taegeon.um@samsung.com

Byungsoo Oh
Samsung Research
byungsoo.oh@samsung.com

Byeongchan Seo
Samsung Research
bchan.seo@samsung.com

Minhyeok Kweun
Samsung Research
mh.kweun@samsung.com

Goeun Kim
Samsung Research
ge326.kim@samsung.com

Woo-Yeon Lee
Samsung Research
wooyeon0.lee@samsung.com

ABSTRACT

When training a deep learning (DL) model, input data are preprocessed on CPUs and transformed into tensors, which are then fed into GPUs for gradient computations of model training. Expensive GPUs must be fully utilized during training to accelerate the training speed. However, intensive CPU operations for input data preprocessing (input pipeline) often lead to CPU bottlenecks; correspondingly, various DL training jobs suffer from GPU underutilization.

We propose FastFlow, a DL training system that automatically mitigates the CPU bottleneck by offloading (scaling out) input pipelines to remote CPUs. FastFlow carefully decides various offloading decisions based on performance metrics specific to applications and allocated resources, while leveraging both local and remote CPUs to prevent the inefficient use of remote resources and minimize the training time. FastFlow's smart offloading policy and mechanisms are seamlessly integrated with TensorFlow for users to enjoy the smart offloading features without modifying the main logic. Our evaluations on our private DL cloud with diverse workloads on various resource environments show that FastFlow improves the training throughput by $1 \sim 4.34\times$ compared to TensorFlow without offloading, by $1 \sim 4.52\times$ compared to TensorFlow with manual CPU offloading (tf.data.service), and by $0.63 \sim 2.06\times$ compared to GPU offloading (DALI).

PVLDB Reference Format:
Taegeon Um, Byungsoo Oh, Byeongchan Seo, Minhyeok Kweun, Goeun Kim, and Woo-Yeon Lee. FastFlow: Accelerating Deep Learning Model Training with Smart Offloading of Input Data Pipeline. PVLDB, 16(5): 1086 - 1099, 2023.
doi:10.14778/3579075.3579083

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

*Corresponding author
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. 16, No. 5 ISSN 2150-8097.
doi:10.14778/3579075.3579083

1 INTRODUCTION

Over the past decade, the performance of GPUs has increased by more than 15 times [16]. This has allowed researchers to accelerate the training of deep learning (DL) models in areas such as speech recognition [34], computer vision [31], and translation [23, 24]. New types of powerful GPUs have enabled the training of large-scale DL models with millions or billions of parameters, advancing human-level accuracy. Moreover, the GPUs have accelerated the training speeds of various DL models and boosted the development of diverse DL applications [26].

In the last few years, however, DL model and system developers have noted that the fast speed of the GPU does not always guarantee fast DL training, owing to CPU bottlenecks in the DL input pipeline [14, 15, 27, 43, 47, 49]. In the DL input pipeline, input data are decoded, preprocessed, and augmented on CPUs to create tensors for computations on GPUs. Although CPU and GPU operations can overlap, the time spent on CPUs for preprocessing can be longer than that spent on GPUs for tensor computations. As a result, expensive GPUs wait for CPU computations, leading to *preprocessing (prep) stalls* and GPU under-utilization.

Recently, prep stalls have become a significant problem in DL training [55] and have gained considerable attention from researchers for the following reasons. First, the speeds of CPUs are relatively low compared to those of GPUs due to the short release cycle of high-end GPUs [9, 16]. Second, as the data-centric AI paradigm [44] has shifted the approach regarding the impact on accuracy from developing a better model (model-centric) to acquiring and synthesizing better-quality data (data-centric), data preprocessing such as cleaning and augmentation (e.g., randomizing the original dataset for model accuracy) require complex and CPU-intensive operations [30, 43, 44, 51]. In addition, resources with a fixed number of CPUs per GPU in the cloud [2] or clusters cannot satisfy the various CPU demands of diverse training jobs. [33, 57].

Existing works have attempted to address prep stalls using various techniques [14, 15, 27, 40, 43, 48, 49], but they have several limitations. First, some of the works [40, 43, 48] could not address preprocessing bottlenecks when the required CPU resources exceed the allocated CPUs. To employ computing resources in addition to CPUs, DLBooster [27], TrainBox [49], and DALI [15] offload preprocessing to specialized hardware (FPGAs and GPUs). However, users must manually convert CPU operations to the limited operations supported by the specialized hardware, which is a non-trivial work. In addition, it is difficult to offload general operations (e.g., user-defined functions or third-party libraries).

To overcome these limitations, tf.data.service [14], a service running on top of TensorFlow [22], distributes and scales out the general preprocessing operations to remote CPUs. However, users should judiciously decide when to offload the input pipeline and which preprocessing operations to offload by themselves according to the workload and resource environment because naively offloading can lead to significant performance degradation. In addition, users cannot precisely control the amount of operations to be offloaded to fully leverage both the allocated local and remote CPU resources for performance improvement and resource efficiency.

In this paper, we design FastFlow, a DL training system that automates offloading decisions, *when to offload, which operations to offload, and how much data to offload to the provided resources.* To automate such decisions, FastFlow measures performance metrics specific to applications (e.g., encoding/decoding and threading overheads) and allocated resources, which are not considered thoroughly in existing work. These metrics are measured with *lightweight* metric profiling before the actual run to estimate the benefit of offloading in advance. For profiling, FastFlow traverses the input data pipeline consisting of a directed acyclic graph (DAG) and inserts profiling operators. To strike the right balance between profiling accuracy and overhead, FastFlow harnesses the iterative characteristic of DL training; a deep learning training job executes the same operations iteratively per batch, i.e., a collection of elements for tensor computations on GPUs. FastFlow estimates the metrics with a few iterations of batches and further minimizes the profiling overhead by storing and reloading metrics for repetitive DL jobs based on DAG matching.

For offloading without user intervention, FastFlow extends the existing DL framework library [10] and gathers additional configuration (e.g., remote node addresses for offloading). This is key to the usability of FastFlow in various workloads, as users can maintain the main logic of preprocessing and training without modification.

We have implemented FastFlow on top of TensorFlow 2.7.0 [22] and evaluate FastFlow with seven DL workloads (image and audio) that represent various preprocessing operations in our private DL cluster. Our comprehensive evaluations show that FastFlow improves the training throughput by $1 \sim 4.34\times$ compared to TensorFlow without offloading, by $1 \sim 4.52\times$ compared to TensorFlow with manual CPU offloading (tf.data.service [14]), and by $0.63 \sim 2.06\times$ compared to DALI [15] on various resource environments.

We make the following contributions in this paper:

- We motivate scenarios where careful decisions must be made to offload input pipeline to remote CPUs, as naively offloading all operations and data does not always guarantee training speed up (reducing the epoch time) and can lead to unnecessary remote CPU use.
- We design FastFlow, which automates offloading decisions without user intervention and modification of the main logic. We seamlessly integrate the smart offloading features on top of TensorFlow with lightweight profiling.
- We empirically show the effectiveness of FastFlow in various workloads and resource environments: FastFlow prevents offloading when there is no offloading benefit. Otherwise, FastFlow automatically offloads operators and data to achieve optimal performance in our private cluster.

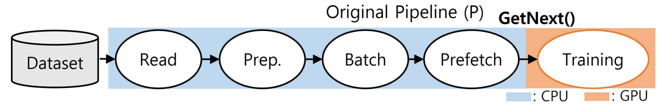

Figure 1: An example of DL training pipeline. A circle represents an operator. The training operator executes gradient computations, whereas the remaining operators are for input preprocessing.

2 BACKGROUND AND MOTIVATION

2.1 Deep Learning Training

Deep learning (DL) model training usually consists of three phases: 1) offline input preprocessing, 2) online input preprocessing, and 3) gradient computation. In this study, we mainly focus on the online input preprocessing step; however, we also explain the remaining steps for a complete understanding.

Offline Preprocessing. The offline preprocessing phase converts raw data collected from various sources to training datasets used in the remaining phases. For offline preprocessing, distributed data processing systems such as Apache Spark [56] and Flink [25] are widely used to deal with tens of GBs or TBs of data [57], leading to a massive data preprocessing overhead. Therefore, users often attempt to reduce the offline preprocessing overhead by storing the preprocessed data as a training dataset into a shared storage, such as an object store [3] or local solid-state drive (SSD) storage. In this study, we do not focus on the offline preprocessing overhead, assuming that it can be removed by caching on the data storage.

Online Preprocessing. The training dataset are then loaded from the data storage, decoded, and augmented [29, 30] to create and feed tensors, multi-dimensional arrays, into GPUs for the gradient computations. Unlike offline preprocessing which can be cached and executed once, online preprocessing inevitably incurs multiple times for each training. This is because decoding is required to load a large amount of data that may exceed the memory size (so we cannot cache the dataset), and augmentation is a random operation that randomly tweaks each element to create diverse data and improve model accuracy. These decoding and augmentation operations are typically executed on CPUs with CPU-intensive operations (e.g., a room impulse response simulation [51]). We use the term preprocessing to refer to online preprocessing.

The online preprocessing phase, which we call the input pipeline throughout this paper, can be expressed as a directed acyclic graph (DAG) [18, 48], where a node is an operator and an edge represents the data flow between operators. Figure 1 shows an example of the input pipeline in which the training dataset is read and preprocessed. Here, an operator is coarse-grained. For instance, we represent transformation and augmentation operations (e.g., map function) as the single `Prep.` operator. The training operator fetches tensors for gradient computation (whenever `GetNext()` is called). [1] To transform the data into tensors and fully utilize the massive parallelism of the GPUs, users generally mini-batch several elements as tensors for gradient computations. The batch operator is then applied after preprocessing (e.g., decoding and augmentation) and before the gradient computations to create tensors and to maximize GPU utilization. Users can overlap the preprocessing

[1] As we focus on the input pipeline, we abstract the training with a single operator. However, training can also be represented as a DAG (multiple layers for DL).

```
1  import tensorflow as tf
2  # Input pipeline
3  ds = tf.data.Dataset(data_path)
4  ds = ds.map(...). distribute (dispatcher_addr..)
5          .batch(..).prefetch(..)
6  # define model
7  class MyModel(tf.keras.Model):
8      def __init__(self,...):
9          ...
10 model = MyModel(..)
11 # training
12 model.compile(..)
13 model.fit(ds, epoch=..,)
```

Source Code 1: tf.data.service API for CPU offloading to the allocated resources. Users should manually insert a `distribute` operator.

on CPUs and gradient computations on GPUs with prefetching to improve the GPU utilization. We explain the details in § 2.2.

Gradient Computation. During gradient computation, the weight of the model is updated according to forward and backward computations on the GPUs. The forward computation feeds the tensors into the layers of the model and compares the predicted value with the target value to calculate the loss. To minimize the loss, backward computation computes the gradient of each layer for the loss and adjusts the weights to reduce the loss.

The DL training involves *iterative* processing. The weights must be updated and tuned iteratively to improve model accuracy. While training, the training dataset is consumed entirely for the gradient computations, which is referred to as one *epoch* of training. Usually, multiple epochs are performed for higher accuracy by iterating the same dataset multiple times, and augmentation is required to prevent the model from being overfitted on the same dataset.

2.2 Preprocessing Stall in DL Training

Existing DL frameworks support prefetching to overlap CPU and GPU computations [18, 48]. However, despite prefetching, GPUs can be underutilized if the computation time on the CPUs is *relatively* higher than that on GPUs. We refer to this situation as a *preprocessing (prep) stall.*

Recently, various workloads suffer from the prep stall because the computational complexity of augmentation techniques has increased [30, 51] to generate more realistic data (e.g., simulations). In contrast, the gradient computation speed on GPUs is increasing due to the short release cycle of high-performance GPUs. Therefore, the relative speed of the preprocessing on CPU decreases, which results in prep stall.

In addition, widely-used cloud resources for DL training commonly have a fixed CPU:GPU ratio [2], and therefore, they cannot satisfy the various required number of CPUs per GPU of diverse DL training jobs for no prep stalls (see Figure 7 in our evaluation). The fixed allocation of CPUs leads to the CPU bottleneck when the input data preprocessing requires more CPU powers than the allocated ones. Simply choosing instances with a high number of CPUs per GPU may lead to CPU under-utilization and cost inefficiency.

Furthermore, a recent study on real cluster ML workloads [55] showed that many ML jobs use CPUs more extensively than GPUs. This is a challenging real-world problem that leads to GPU underutilization. We also investigated various DL training workloads and found prep stalls in our GPU cluster environment. In short, up to 80% of the epoch time is stalled by the preprocessing in our evaluation. The detailed results are presented in § 7.

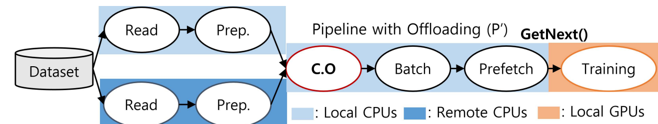

Figure 2: Example of input pipeline offloading to remote CPUs. C.O represents a conditional operator that fetches an element either from operators running on the local CPUs or from the remote CPUs. The C.O can control the amount of data to be processed on the remote CPUs.

2.3 Limitations of Existing Work

Existing work [14, 15, 27, 40, 48, 49] mitigates the prep stall with various techniques, but they have several limitations.

Auto-Tuning on Local Node. Recent studies such as Plumber [40] and tf.data [48] tries to resolve the prep stall with auto parallelism and caching on the local training node, where the training code is executed, and GPUs are used for gradient computations. However, parallelization is limited to the allocated local CPUs, so it cannot resolve CPU bottlenecks if preprocessing requires more CPUs than allocated local CPUs. Caching could not be possible if the data size is larger than memory size, and reusing cached data hinders random operations of augmentation, which is critical for model accuracy. Revamper [43] partially caches augmented data while preserving accuracy, but requires users to divide augmentations into multiple layers. It cannot be generally applied if if users use third-party libraries or a single map function that augments data.

Offloading to Specialized Hardware. DALI [15] enables input preprocessing on GPUs, and DLBooster [27] and TrainBox [49] focus on offloading operations (e.g., image decoding, resizing) to FPGAs. However, they are limited because users may write CPU operations, which are not supported by the specialized hardware. Moreover, manually transforming CPU operations into the operations supported by the systems is not straightforward and requires knowledge for the operator mapping between different systems.

Offloading to Remote CPUs. tf.data.service [14] is a service that offloads general operations of input data pipelines to remote CPU clusters on top of TensorFlow. tf.data.service consists of two components: a dispatcher and workers. A worker process runs on a remote node and executes data preprocessing of offloaded operations. The worker sends the preprocessed results back to the local (client) TensorFlow node, where the gradient computations are performed on GPUs. A dispatcher communicates with the workers and manages their life cycle. With such offloading, preprocessing is not limited to the allocated CPUs of the local GPU node, which can mitigate the CPU bottleneck in the local GPU node.

tf.data.service requires users to make the following decisions by themselves. Source Code 1 shows an example code snippet of using tf.data.service. First, to enable offloading, users must insert a `distribute` operator. As an example, when a `distribute` operator is added after the `map` operator (line 4 in Source Code 1), the previous operations before the `distribute` operator (e.g., dataset read and map in Source Code 1) are executed on remote workers.

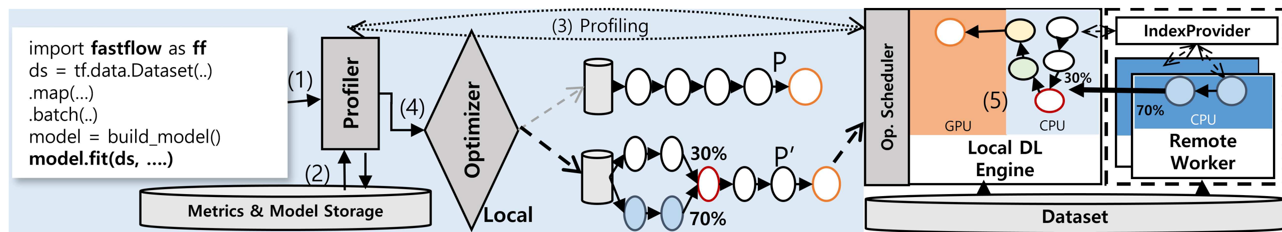

Figure 3: The overall workflow of FastFlow to execute a DL training job with smart offloading.

Second, users must decide where to place the `distribute` operator to control which operations to offload. The operator can be inserted before the map operator not to offload the map operations to remote workers. Third, to leverage the CPUs of the local client node for preprocessing, users must create a local worker on the local client node. When the local worker is added, the tf.data.service `distribute` operator distributes the preprocessing across the local and remote workers to harness both local and remote CPUs.

Although tf.data.service enables users to offload preprocessing to remote CPUs manually, it has several limitations. First, making decisions for optimal performance and efficient resource use is difficult because the decisions can vary according to the application's characteristics and resource capacity. For instance, an application does not have a prep stall in a resource environment, but users may offload input pipeline and lead to unnecessary remote resource use. In addition, the capacity of remote CPU resources may be lower than the required CPU power for offloading, and the network bandwidth between local and remote machines could also be lower than the required data transfer. Second, even if tf.data.service offers a mechanism of leveraging both local and remote CPUs with local workers, it does not provide a knob that fine-controls the amount of data to offload across local and remote resources. The optimal amount of data to be offloaded varies according to applications and allocated resources to achieve best performance (see Figure 9 in § 7) but users cannot configure the optimal ratio of data offloading.

Recent work, Cachew [33], provides an auto-scaling of workers on top of tf.data.service, but Cachew users must also decide when and which operations to offload like Source Code 1. Moreover, as Cachew is designed for disaggregated environments with high network and disk bandwidth and abundant remote CPU resources, it offloads all data preprocessing to remote workers without harnessing local CPUs. This approach may result in inefficient remote resource use and degrade performance in the environments where the network and disk bandwidth and CPU resources are not enough.

2.4 Our Approach and Challenges

To address the limitations, we propose *automatic offloading* of input pipeline to remote CPUs. Automatic offloading, or smart offloading, means that systems offload input pipeline without user intervention to achieve best performance on the given resources. Figure 2 illustrates an example of input pipeline offloading, where the `C.O` operator is automatically inserted after `Prep.` operator and fine-controls the amount of data to offload based on our decisions.

To the best of our knowledge, it is the first work that addresses the below challenges for automatic and optimal offloading decisions, while seamlessly integrating such decision making with existing deep learning frameworks to lessen users' burden.

- **When to offload?** If there is no prep stall or CPU bottleneck, the input pipeline should not be offloaded and the remote CPU resources should be saved.
- **Which operations to offload?** Because there are multiple operators for preprocessing (e.g., decode, augmentation, and, batch), we should decide which operators to offload for best performance.
- **How much data to offload?** To leverage both local and remote CPUs, we should automatically fine-tune the right amount of data for offloading.

3 SYSTEM OVERVIEW

To address the above challenges, we design and implement FastFlow, an extension of existing DL systems for smart offloading. Based on profiled metrics, we design a new smart offloading decision policy (§ 4) on top of TensorFlow. For an efficient offloading mechanism, we create a new conditional operator and profiling feature in the TensorFlow core (§ 5). Although we design FastFlow on top of TensorFlow, we believe that our design for auto-offloading is generally applicable to other DL systems (e.g., Pytorch [50]) with some engineering efforts for the offloading mechanism.

FastFlow is designed to maximize DL training job performance in environments where each training job runs on allocated resources. However, we believe that FastFlow can be easily extended for disaggregated or cloud resources and be easily integrated with Cachew [33] for autoscaling. We will briefly discuss the extension of FastFlow for autoscaling in § 9. In the following section, we describe the overall workflow of how FastFlow enables smart offloading (§ 3.1) and show how users can easily use FastFlow with minor code modifications (§ 3.2).

3.1 Overall Workflow

Figure 3 illustrates an architectural overview of FastFlow and how FastFlow is integrated with TensorFlow. The current implementation covers the single-node/multi-GPU training. Extending FastFlow to multi-node/multi-GPU training remains as future work.

For automatic decision and profiling system metrics, FastFlow requires information such as input pipelines, models, and remote resource environments for offloading. To acquire this information with minimal code modifications, FastFlow's model interface inherits the existing classes of DL frameworks (e.g., keras.Model), where users can set the configuration and deep-copy method for the profiling without modifying the main logic (Figure 3(1)). The detailed example is presented in § 3.2.

Once the training code is written, FastFlow then profiles the performance metrics before the actual training if the input pipeline and training model is newly executed (Figure 3(2)). For profiling,

```python
import fastflow as ff
import tensorflow as tf
# input pipeline
ds = tf.data.Dataset(data_path)
ds = ds.map(...).batch(..).prefetch(..)

# define model
class MyModel( ff.Model ):
    def __init__(self,...):
        ...
    def __deepcopy__(self):
        # must be implemented
model = MyModel(..)
# training with auto-offloading
model.compile(..)
model.fit(ds, epoch=.., ff_config=config)
```

Source Code 2: FastFlow's API for smart offloading. Note that users keep the main logic of building the input pipeline, model, and training code without manual configuration of the `distribute` operator.

FastFlow executes a few iterations of training on local nodes (Figure 3(3)) and stores the metrics in the metric storage with the corresponding input pipeline, model, and configuration. When the same input pipeline and model are executed again with the same resources, FastFlow loads the saved metrics to remove the profiling overhead. Based on the profiled metrics, FastFlow executes the original input pipeline (P) if there is no performance benefit for offloading (if there is no prep stall). Otherwise, FastFlow modifies the input pipeline DAG (P') to partially offload data and operators to remote CPU resources (Figure 3(4)).

For execution (Figure 3(5)), FastFlow harnesses the existing DL engine (TensorFlow in this paper) with an extension of operator kernels for offloading. In TensorFlow, as explained in § 2, tf.data.service [14] enables the horizontal scaling of the input pipeline to remote CPUs. For pipeline execution on the remote CPUs, tf.data-.service clones the pipeline operators and executes the operators on workers, with each worker inheriting the same code-base of TensorFlow's internal execution engine. To feed the preprocessed data from the workers to the GPUs, the `distribute` operator fetches data through remote procedure calls from the workers.

FastFlow automatically creates a tf.data.service worker process when a job starts and destroys the workers after the job is finished. To enable partial offloading and control the data preprocessing ratio for the remote workers, FastFlow creates and injects a conditional operator, which is the extension of the `distribute` operator. The conditional operator receives the local-remote data processing ratio (as configured by FastFlow's auto-decision logic) as a parameter. Currently, FastFlow assumes homogeneous remote workers with the same amount of CPUs, and thus evenly distributes data across the remote workers when offloading.

Note that the input data should be stored in the shared storage or need to be copied into the worker's local data storage, in order for the remote workers and local DL engine to have the same view of the input data and for FastFlow to dynamically control data preprocessing across the remote workers and local DL engine. Users can choose to store their dataset in shared storage if the network bandwidth is high; or to copy their datasets once into remote nodes.

When fetching data, FastFlow must synchronize the data indices to be processed across machines. tf.data.service already provides IndexProvider within a dispatcher for dynamic data sharding, which communicates with local and remote workers. FastFlow executes

```python
class Model(keras.Model): # FastFlow Model
  ...
  def fit(self, x=None, # input pipeline
          conf=None, # auto-offloading conf
          **kwargs):
    with self.distribute_strategy.scope():
      M = self.__deepcopy__()
      M.compile(..)
    launch_workers(conf)
    P = smart_offloading(x,M,conf)
    # Reuse the main training logic
    super(Model, self).fit(x=P, **kwargs)
    destroy_workers(conf)
```

Source Code 3: FastFlow's `model.fit` method.

IndexProvider on a designated worker node (or on the local node). Workers request an index from the IndexProvider, and it basically increments the index for each request to synchronize the data preprocessing and sends it back to the workers. The workers then preprocess the specific data with the index to prevent duplicate data preprocessing. As the local and remote CPUs can preprocess data concurrently, the preprocessed data are gathered in the local machine in out-of-order.

3.2 API with Code Example

In designing FastFlow, we factor in usability as a key requirement. This section illustrates how existing TensorFlow users can easily use FastFlow with relatively minor code modifications (Source Code 2).

To use FastFlow, users should import FastFlow's user-facing library (line 1 in Source Code 2), and FastFlow's customized TensorFlow core (line 2 in Source Code 2). Users can define their input pipelines without code modifications. Typically, tf.data [48] is used for data preprocessing and the input pipeline (lines 4–5). The users only need to set the data path as a shared data storage path or a local data storage path (line 4), where the same dataset is located.

After defining the input pipeline, the users define their training model. To easily define and train their model, most TensorFlow users use the Keras library [10]. FastFlow extends the predefined Keras model and provides `ff.Model` (`ff.Model` extends `keras.Model`) to minimize code modifications. Users then only need to extend `ff.Model` to build their customized model, and to additionally write the `__deepcopy__` method as described in lines 11—12. The method returns a new copy of the model and is internally used in FastFlow's auto-offloading logic to prevent the model parameters from being modified during the profiling phase (§ 4). The other methods of `keras.Model` can be used without modification. In the `MyModel` class, we omitted the model building code due to space constraints; in fact, users can write hundreds of lines of code for their model. Once the input pipeline and model are defined, training is performed with the `model.fit` method in Keras. FastFlow receives arguments for auto-offloading configuration (line 16) such as resource environments (e.g., addresses of remote nodes) in addition to the input pipeline.

Within the `model.fit` method, FastFlow profiles the performance metrics and makes offloading decisions, as described below.

4 SMART OFFLOADING POLICY

FastFlow's offloading policy automatically decides the following:

- Is there a prep stall, and is offloading beneficial (§ 4.1)?

Algorithm 1: The smart offloading policy.

```
1  Input: Input Pipeline P, model M, conf Conf
2  Profile Gthp = thp(M) and Lthp = thp(P+M)
3  if Gthp <= Lthp or Gthp/Lthp < SpeedUpThreshold then
4      Return P                                   // No data offloading
5  else
6      ℙ = {p'1, p'2, p'3} = CandidatePipelines(P, Conf)
7      P' = argmax_{p'∈ℙ} {Lthp * (1 − Ocycle(p')/Pcycle) + Rthp(p')}
8      Upper = Rthp(P')
9      Lower = (Gthp - Lthp) * (1 + Ocycle(P')/Pcycle)
10     if Lower < Upper then
11         OffRatio = Upper / Gthp
12     else
13         OffRatio = Rthp / (Lthp + Rthp)
14     Return ApplyOffRatio(P', OffRatio)
```

Table 1: Profiled metrics for optimal offloading decision. *P* is the original input pipeline, *M* is the model, and *P'* is the modified input pipeline of *P* for offloading. All of the metrics are measured on the local GPU machine. All units are MB/s except for *Ocycle* and *Pcycle*.

Metric	Description
Gthp	The maximum GPU throughput of gradient computations of model (M) ($thp(\boldsymbol{M})$)
Lthp	The maximum training throughput of input pipeline (P) and gradient computations of model (M) ($thp(\boldsymbol{P}+\boldsymbol{M})$)
Rthp	The maximum training throughput when all data is preprocessed on the remote nodes with the offloaded input pipeline (P') and gradient computations of model ($thp(\boldsymbol{P'}+\boldsymbol{M})$)
Ocycle	CPU cycles on local CPUs for training with offloaded pipeline
Pcycle	CPU cycles on local CPUs for training without offloading

- Which operations should be offloaded (§ 4.2)?
- How much data should be offloaded (§ 4.3)?

Source Code 3 shows a snippet of FastFlow's `Model` internal code. As profiling may change the weights of the model (because profiling performs model training with sample dataset) and affect the model accuracy, FastFlow creates a copy of the model with the `__deepcopy__` method and uses this copied model for the smart offloading decision (line 7) to separate the actual model training and profiling. If the model is trained on multi-GPUs, FastFlow applies the distributed strategy to enable profiling on multiple GPUs (line 6). FastFlow launches remote workers before profiling to execute preprocessing on the remote workers during profiling (line 9). Once the smart offloading policy (`smart_offloading`) profiles metrics and returns a modified input pipeline for offloading, FastFlow executes the existing `keras.Model.fit` method with the modified pipeline for model training (line 12) and destroys the remote workers once the training is done (line 13).

For smart offloading decision (line 10, `smart_offloading`), FastFlow uses five metrics: *Gthp*, *Lthp*, *Rthp*, *Ocycle*, and *Pcycle* (summarized in Table 1 and Figure 4). The detailed metric profiling process is described in § 5.1. Next, we will illustrate Algorithm 1 for prep stall detection (§ 4.1), offloading operator decision (§ 4.2), and offloading data ratio decision (§ 4.3).

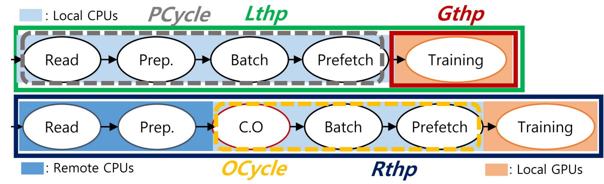

Figure 4: An illustration of measuring metrics for offloading decisions.

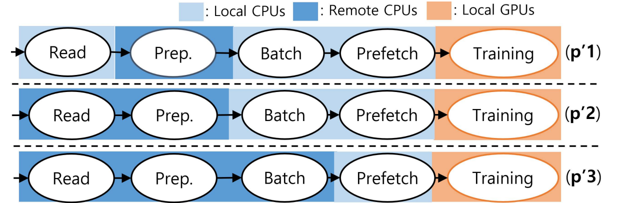

Figure 5: Candidate pipelines for offloading.

4.1 Preprocessing Stall Detection

Lines 2—4. Before starting the actual training on the DL engine with the whole dataset, FastFlow checks whether there is a prep stall, so as to prevent additional profiling. This can be determined with two metrics: *Gthp* and *Lthp* (see Table 1 and Figure 4). *Lthp* = *Gthp* means that the GPUs consume the preprocessed data on CPUs without waiting and stalls. In this case, FastFlow executes the original input pipeline on local CPUs without offloading (line 4). Otherwise, FastFlow checks whether or not FastFlow can increase the speed by more than the *SpeedUpThreshold* (line 3). The ideal increase in speed is calculated by $\frac{Gthp}{Lthp}$ (the speed-up when there is no prep stall). By default, FastFlow does not offload preprocessing if the maximum increase in speed is negligible (less than 10%), so as to prevent inefficient remote resource use.

4.2 Offloading Operator Selection

Lines 6—7. Once FastFlow decides to offload preprocessing to remote CPUs, FastFlow next decides which operators to offload in the input pipeline DAG (e.g., where to put the `distribute` operator). As the performance may vary according to the decisions, FastFlow creates several candidate pipelines (line 6) for offloading by traversing the DAG. FastFlow then compares the estimated training throughput with offloading and selects the pipeline that leads to the maximum training throughput and minimum prep stall (line 7).

In our current design, FastFlow judiciously chooses three candidate pipelines (illustrated in Figure 5) to comprehensively consider network, disk I/O, offloading overheads (e.g., encoding/decoding and threading), and computation capacity while offloading. The first pipeline (***p'1***) offloads only the computation (`Prep.`), while reading the data in the local node. The second candidate (***p'2***) pipeline offloads operators sequentially from the data read operator (data source), except for the batch operator. The third pipeline (***p'3***) naively offloads all operations including batch. FastFlow currently considers the three candidates because usually users express the `Prep.` operation as a single map operator to reduce the overheads of function calls of multiple operators. In the future, we plan to enable users to give a hint for candidate pipelines with annotations when multiple operators exist for the `Prep.` operation.

According to the workloads and resource environments, the best candidate pipeline may change among the three candidates. For instance, ***p'1*** can lead to the best performance compared to the others when the data transfer throughput (`Read`→`Prep.` in ***p'1***) from the local to the remote node is higher than the data fetching throughput on the remote node (`Read` in ***p'2*** and ***p'3***). This can happen when the I/O performance or network bandwidth of the remote node storage is too low, or the overheads for offloading (e.g., encoding and decoding) is negligible.

Otherwise, choosing ***p'2*** or ***p'3*** is better than ***p'1*** as ***p'1*** requires additional encoding overheads. Between ***p'2*** and ***p'3***, the performance benefit varies according to the computing capacity, network bandwidth, and threading overheads of remote nodes. For ***p'3***, the granularity of the data offloading becomes a batch, and therefore, one fetch request leads to preprocessing the multiple elements of a batch entirely on the remote side. In contrast, keeping the batch operator on the local machine (***p'2***) enables element-wise request for data preprocessing on the remote side, but it causes network I/O and encoding/decoding overheads for each element for offloading.

When the remote computing capacity (or network bandwidth) is enough to promptly process a batch, ***p'3*** is better than ***p'2*** because it reduces the network I/O and encoding/decoding overheads. However, in some cases, multiple batch requests by the prefetch operator may result in significant threading overheads in the remote node, which degrades the performance. In contrast, ***p'2*** causes less threading overheads than ***p'3*** as each request just requires preprocessing one element. FastFlow considers all of the cases by comparing the estimated training throughput with the candidate pipelines, and we will explain how to calculate the estimated training throughput in the following section.

4.3 Offloading Data Ratio Decision

Lines 8—14. Once the candidate pipeline is decided, FastFlow next decides the right amount of data to offload by considering the local and remote computing/network capacity and threading overheads (*Lthp*, *Rthp*) as well as offloading overheads (encoding/decoding) in the local node (*OCycle* and *PCycle*).

To harness both local and remote CPUs, we consider the $Lower$ and the $Upper$ amount of data to offload. Offloading data more than $Upper$ will lead to the remote CPU or network bottleneck (line 8), whereas offloading data less than $Lower$ will lead to the local CPU bottleneck (line 9). Intuitively, offloading more than *Rthp* to the remote nodes will lead to the remote bottleneck, and therefore, $Upper$ is *Rthp*. For $Lower$, as the local CPU will be the bottleneck when we try to process more than *Lthp* in the local, it seems that $Lower = Gthp - Lthp$. However, as encoding data and decoding offloaded results require additional CPU cycles (*Ocycle*) in the local node, we also consider the overheads to prevent local CPU from being the bottleneck. As a result, FastFlow tries to offload more data to the remote nodes if the additional overhead for offloading is large compared to preprocessing overhead (*Pcycle*). This is represented by multiplying the ratio of offloading and preprocessing cycles $(1 + \frac{Ocycle}{Pcycle})$ to $Gthp - Lthp$. Similarly, $Lthp * (1 - \frac{Ocycle}{Pcycle})$ represents the effective training throughput in the local node with offloading overheads (the larger offloading overheads are, the lower effective throughput is in the local node), so the estimated training throughput with offloading is $Lthp * (1 - \frac{Ocycle}{Pcycle}) + Rthp$ (line 7).

Table 2: The key-value pair for storing and loading profiled metrics. *P* is the original input pipeline, *M* is a model, *P'* is one of *p'1*, *p'2*, and *p'3*. For the model, FastFlow also stores its batch size, as the batch size affects GPU throughput. If the key is matched, FastFlow loads the corresponding metric and skips profiling.

Key	Value
M	$Gthp$
(P, M)	$(Lthp, Pcycle)$
(P, M, RemoteNodeInfo)	$(Rthp, P', Ocycle)$

In some cases, $Upper$ is lower than $Lower$, e.g., when the remote CPU power or network bandwidth is low (line 13). It represents that FastFlow cannot avoid a local or remote resource bottleneck. The best way in this case is to evenly balance the load among local and remote CPUs based on the resource capacity $(\frac{Rthp}{Lthp+Rthp})$.

Once the offloading ratio is decided, FastFlow modifies the input pipeline ***P'*** by applying the ratio value (line 14). In this transformation, FastFlow provides the offloading ratio as a parameter for the partial offloading operator to control the amount of data for offloading at runtime (based on the ratio).

5 SMART OFFLOADING MECHANISM

As described above, the profiled metrics are key to the offloading decisions (§ 5.1). To minimize the profiling overheads, FastFlow stores and reloads the metrics whenever the same input pipeline and model are trained (§ 5.2). Once the decision is made, FastFlow evenly balances and controls the amount of data to fetch from the remote workers (§ 5.3). In this section, we illustrate FastFlow's mechanism for smart input pipeline offloading.

5.1 Metric Profiling

To strike the right balance between profiling overhead and accuracy, FastFlow harnesses the iterative characteristic of DL training. As the DL training iteratively processes the input data with multiple epochs and batches, FastFlow executes only a few iterations and estimates the metrics without processing the entire data. We empirically found that profiling 50 ∼ 100 steps (batches) leads to high accuracy with small overheads [33], as training a model takes hundreds of epochs, each of which consists of a few hundreds of steps. We show the profiling overhead in our evaluation.

There are three types of metrics for profiling. First, to measure the GPU throughput of the model gradient computations ($Gthp = thp(M)$), FastFlow removes the prep stall by caching the preprocessed data of the first step in memory and reusing them to feed the GPUs. Second, to measure the throughput of the entire input pipeline and model training ($Lthp = thp(P + M)$ and $Rthp = thp(P' + M)$), FastFlow simply executes the original and the modified input pipeline and measures their throughputs (*Lthp* and *Rthp*). Third, FastFlow estimates the CPU cycles (*Ocycle* and *Pcycle*) by executing the input pipeline without gradient computations.

5.2 Metric Store and Load

FastFlow further reduces the profiling overhead by storing metrics and reloading them if the same input pipeline and model are trained

again [55]. To detect the same input pipeline and model, FastFlow matches the DAG graph information of a job, i.e., the number of nodes, node information, and edge information.

To detect the same model, FastFlow removes parameters and compares only the model DAG information, as the training throughput usually depends on the model architecture (e.g., number of layers, nodes), not on its parameters. FastFlow also stores and matches the remote node information when the metrics are profiled, because different remote nodes may change the offloaded throughput of the pipeline (*Rthp*). FastFlow stores the metrics in a key-value store, where the key is the combination of the input pipeline, model, or remote node information, and the value is the corresponding metric and an offloaded pipeline (***p'1***, ***p'2***, and ***p'3*** in Figure 5) depending on the key. These key-value pairs are described in Table 2.

5.3 Balanced Offloading

In controlling data to offload, FastFlow evenly distributes data processing (the `GetNext` call) between local and remote CPUs to minimize prep stall. For balanced offloading, FastFlow uses a counter-based approach. FastFlow maintains counters for local and remote CPUs and increments the corresponding counter whenever it fetches and preprocesses data from the resources. FastFlow then decides data preprocessing based on the ratio calculated by the counters. Another alternative approach is the static data partitioning approach, like splitting the index of data for local and remote (e.g., $2x$-th data in local, and $2x+1$ data in remote where $0 \leq x < 50$). This approach is beneficial when the number of remote workers is static, but we choose the counter-based approach to easily support dynamic load rebalancing across workers in the future.

6 IMPLEMENTATION

We have implemented FastFlow on top of TensorFlow 2.7.0 with around 1,200+ lines of code. To reduce the engineering effort, we use the base offloading mechanism implemented in tf.data.service [14]. For the profiling metrics, we modify TensorFlow's iterator operator. The iterator operator measures the processed data size and elapsed time when a batch is preprocessed. The profiler then extracts this information for each batch to calculate the throughput.

To calculate the throughput without data preprocessing overhead (*Gthp*), FastFlow inserts the following tf.data operators for caching and reusing data by traversing the input pipeline DAG. FastFlow adds the `take(1).cache().repeat()` operators before training to remove preprocessing overhead. The `take(1)` operator takes one batch from the dataset, the `cache()` operator caches the batch, and the `repeat()` operator repeatedly uses the cached batch. For *Ocycle* and *Pcycle*, we use `psutil` [17], which calculates the system and CPU times without I/O time.

7 EVALUATION

In this section, we answer the following questions:

- Does FastFlow improve performance compared to baselines in various workloads and resource environments (§ 7.2)?
- How do the offloading decisions affect the performance of FastFlow (§ 7.3)?
- How long the profiling takes (§ 7.4)?
- Does FastFlow achieve performance gains without affecting model convergence (§ 7.5)?
- Does FastFlow work well on multi-GPU environments (§ 7.6)?

Table 3: Dataset used in the evaluation.

Dataset	Domain	Size	Workloads
ImageNet [31]	Image	150 GB	**R**
LJSpeech1.1 [37]	Audio	2.6 GB	**C**, **T**, **M**
Caltech Birds [53]	Image	1.1 GB	**G**
Cats and Dogs [32]	Image	786 MB	**L**
16000 PCM [20]	Audio	256 MB	**S**

7.1 Environment and Setup

We conduct experiments on our private cloud system (PCS), built on Kubernetes [11]. The PCS provides GPU containers, each of which has $7x$ vCPUs and $1y$ NVIDIA V100 GPUs, and CPU containers, each of which has $7z$ vCPUs.[2] In our evaluation, we vary x ($1 \leq x \leq 2$), y ($1 \leq y \leq 4$), and z ($1 \leq z \leq 4$) to analyze performance in diverse resource environments. We measure the average epoch time except for the first epoch to remove warm-up overheads.

Local GPU Node. For the local GPU node that executes input pipeline and gradient computations, we use one PCS GPU container. By default, we allocate a container with 7 vCPUs and one GPU because its ratio is similar to that of the popular cloud GPU instance (e.g., EC2 p3 instances (8:1) [2]). We also vary the number of GPUs from 1 to 4 and vCPU:GPU ratios from 7 : 1 to 14 : 1.

Remote CPU Nodes with High Network Bandwidth. We also vary the number of remote vCPUs of PCS containers from 7 to 28 for offloading. Inter-container communication between PCS containers guarantees network bandwidth more than 20 Gbps with InfiniBand, a high-performance networking standard. Therefore, experiments on the PCS do not cause the network bottleneck for offloading. The local and remote PCS containers read dataset from a shared SSD storage connected with Infiniband.

Remote CPU Nodes with Low Network Bandwidth. To show performance on low network environments, we also use an external on-premise server called PRD [3] as the offloading node. The PRD is outside of the PCS clusters, and therefore, the network bandwidth between PCS containers and the PRD is unstable and relatively low (less than 1 Gbps). We copy the same dataset to the local storage of the PRD node once before the evaluation. We run IndexProvider (dispatcher) on the remote PCS container and PRD node.

Workloads. We use 7 DL workloads in image and audio domains that have various preprocessing. Commonly, the input pipeline consists of the following operations: 1) loading data, 2) applying a map operator for decoding and augmentation, 3) batching, and 4) prefetching. We briefly describe each job as follows. Table 3 summarizes the dataset used in the workloads.

- **Transformer ASR (T)** [5]: Automatic speech recognition using Transformer. Preprocessing includes short-time Fourier transform to get the spectrogram and normalizing.
- **GAN ADA (G)** [6]: A generative adversarial network (GAN) with adaptive discriminator augmentation. Preprocessing includes cropping, resizing, and clipping images.

[2]Intel Xeon Gold 6142 CPU @ 2.60GHz.

[3]30 CPUs of Intel Core Processor (Skylake) @ 2.2GHz

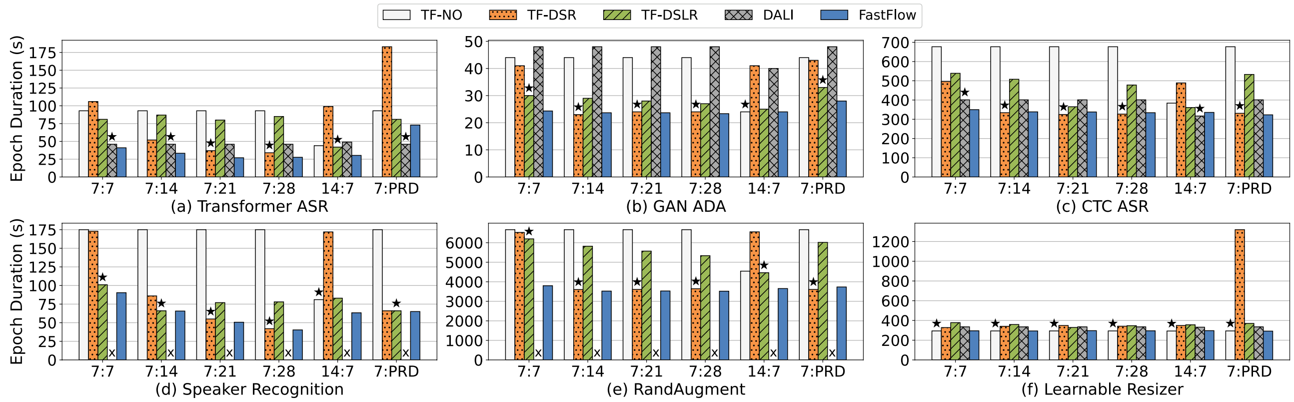

Figure 6: Performance in various workloads and resource environments (lower is better). In x:y and x:PRD, x indicates a local PCS GPU container with x vCPUs and one GPU, y is a remote PCS container for offloading with y vCPUs, and PRD is the remote PRD node for offloading. We omitted MelGAN as it has no prep stall and has the same performance in all systems and resource environments. Bars with $*$ indicate the system that achieves the lowest epoch time among TF-NO, TF-DSR, TF-DSLR, and DALI in each workload. DALI results marked with x represent applications where data pipeline operations cannot be converted into DALI operations due to the limited GPU operation support.

- **CTC ASR (C)** [4]: Automatic speech recognition using two-dimensional (2-D) convolutional neural network (CNN), recurrent neural network (RNN), and connectionist temporal classification (CTC) loss. Same preprocessing as Transformer ASR.
- **Speaker Recognition (S)** [19]: Speaker classification using fast Fourier transform (FFT) and a 1-D convolutional network. Preprocessing includes adding noise randomly and transforming audio waves with FFT.
- **ResNet50 with RandAugment (R)** [30]: Image classification for improved robustness using RandAugment [30]. Preprocessing includes image transformations such as rotate, posterize, solarize, sharpness, etc.
- **MelGAN (M)** [13]: Conditional waveform synthesis using GANs. Preprocessing includes decoding audio files into instances of the wave.
- **Learnable Resizer (L)** [12]: Learning to resize images for a given image resolution and a model. Preprocessing includes resizing and one-hot-encoding.

Baselines. Our baselines are **TF-NO** (TensorFlow 2.7.0 with no offloading), **TF-DSR** (TensorFlow+tf.data.service with remote worker only) that offloads the entire preprocessing load to remote CPUs, **TF-DSLR** (TensorFlow+tf.data.service with local and remote workers) that attempts to distribute the preprocessing load to both local and remote CPUs [4], and **DALI** [15] that offloads preprocessing load to the local GPU without using remote CPU resources. For TF-DSR and TF-DSLR, we choose to offload all preprocessing operations like ***p'3*** in Figure 5. Parameters such as number of operator threads (parallelism), prefetch size, and request buffer size are automatically tuned (`AUTOTUNE`) by tf.data and tf.data.service.

For DALI, we try our best to offload preprocessing operations on the local GPU for supported operations. Note that some operations are not supported by DALI, and converting the original CPU operations to DALI (GPU) operations is not straightforward and time consuming [5], which is the main hurdle for wide adoptions of DALI.

[4]We applied the recent commit [21] that improves tf.data.service performance.
[5]It takes a few weeks for us to port the TensorFlow operations to run on DALI, as we are not familiar with the DALI operations.

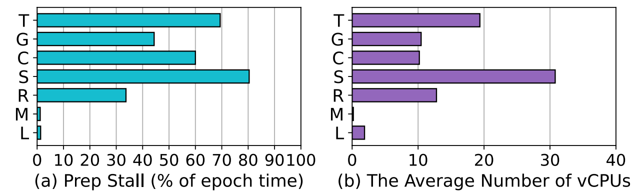

Figure 7: (a) Prep stalls in various workloads measured on 7 vCPUs and one V100 GPU container. (b) The average number of vCPUs for preprocessing without CPU bottlenecks on a V100 GPU.

7.2 Performance Comparison

Figure 6 shows the performance of diverse systems on various CPU resources and network bandwidths. Overall, FastFlow improves training throughput up to 4.34×, 4.52×, 3.07×, and 2.06× compared to TF-NO (in 7:28 of (d)), TF-DSR (in PRD of (f)), TF-DSLR (in 7:28 of (a)), and DALI (in 7:28 of (b)), respectively.

According to applications and resource environments, the system that achieves best performance varies among TF-NO, TF-DSR, TF-DSLR, and DALI, whereas FastFlow always achieves comparable or better performance than the others with optimal and automatic decisions based on the profiled metrics. For instance, in Figure 6(a) 7:7, DALI is better than TF-NO, TF-DSR, and TF-DSLR in performance, but in Figure 6(a) 7:21, TF-DSR has better performance than others. This result indicates that users must judiciously decide offloading decisions to achieve best performance whenever the application and remote resources change.

Comparison with TF-NO. Since TF-NO does not offload preprocessing, it is obvious that FastFlow has better performance than TF-NO when there is a prep stall. However, the performance gain is different from each workload. The higher prep stall and CPU bottlenecks workloads have, the higher speed-up FastFlow achieves compared to TF-NO because FastFlow minimizes the prep stall. In Figure 7(a), Speaker Recognition has the highest prep stall (around 80%), and FastFlow shows the best performance gain (4.34×) against TF-NO in the 7:28 environment (Figure 6(d)). For workloads with

negligible prep stalls (e.g., MelGAN and Learnable Resizer), FastFlow has similar training time with TF-NO because there is no room for minimizing the prep stall.

When the number of local CPUs per GPU increases, i.e., comparing 7:7 with 14:7 in Figure 6, the prep stall and the performance benefit of FastFlow decrease compared to TF-NO. For example, GAN ADA has no prep stall in the 14:7 environment because it requires 10 vCPUs on average for preprocessing with one v100 GPU (Figure 7(b)). However, choosing the right number of CPU:GPU is hard due to the different number of required CPUs in various applications. As shown in Figure 7(b), the required number of average vCPUs per one GPU varies from 7 to 30. FastFlow can lessen the burden of such decisions; users can choose 7 vCPUs:1 GPU for the local node, and FastFlow automatically scales out preprocessing to remote CPUs only when necessary.

Comparison with TF-DSR. As TF-DSR offloads all operations and data to the remote nodes for preprocessing, its performance degrades when the remote resources become the bottleneck for preprocessing. For instance, in the environment with 14 local vCPUs:7 remote vCPUs, harnessing the local CPUs is better than using the remote CPUs for preprocessing. However, TF-DSR tries to preprocess all operations on the 7 remote vCPUs rather than preprocessing on the 14 local vCPUs, which results in the remote CPU bottleneck for the applications that require more than 7 vCPUs for preprocessing.

The performance of TF-DSR also degrades in Figure 6(a), (b), and (f) compared to TF-NO because fully offloading data to the PRD worker causes the network bottleneck. Transformer ASR and Learnable Resizer require higher data transfer rate on average (around 410 and 275 MB/s, respectively) than the PRD network bandwidth (less than 1Gbps). In contrast, FastFlow achieves 3.26× and 4.52× speedup against TF-DSR in the workloads, respectively, because FastFlow decides the operators to offloading and the right amount of data to offload considering the network bottleneck (e.g., *Rthp*).

FastFlow also enables the efficient use of remote resources of shared CPU clusters by harnessing local CPU resources. In Figure 7(b), Transformer ASR requires at most 20 vCPUs for preprocessing, and therefore, TF-DSR requires more than 20 remote vCPUs to saturate the speedup of Transformer ASR (Figure 6(a)). In contrast, when local 7 vCPUs are provided, FastFlow requires only around 13 remote vCPUs to saturate the speedup.

Comparison with TF-DSLR. Compared to TF-DSR, TF-DSLR harnesses the local CPUs for preprocessing, but FastFlow still achieves higher performance than TF-DSLR due to the following two reasons. First, FastFlow compares the estimated training throughput among the three candidates (***p'1***, ***p'2***, and ***p'3***) and finds the best one that leads to maximum performance, whereas TF-DSLR naively offloads all preprocessing operations (like ***p'3***).

The second reason is the effect of offloading data ratio decision. TF-DSLR relies on the operating system's thread scheduling for deciding the amount of data to offload. Therefore, TF-DSLR can request more data preprocessing to the remote nodes, even if the remote nodes become the bottleneck and suffer from high threading overheads. Especially, in the shared container environment (like PCS containers), threading overheads may be exacerbated because the number of threads for preprocessing is larger than the allocated vCPUs of a container. As the tf.data's auto-parallelism logic sets the operator parallelism to the number of CPU cores of the host node by default, [6] in our PCS environment, 64 operator threads are created on a PCS container with 7 vCPUs. As a result, a container can process 64 batches concurrently on the 7 vCPUs, which exacerbates the threading overheads and degrades performance. In contrast, FastFlow decides the optimal offloading ratio not to exacerbate the threading overheads in the local and remote containers with profiled metrics (e.g., *Rthp* and *Ocycle*).

Comparison with DALI. Despite harnessing GPUs for preprocessing, DALI is slower than FastFlow in most training workloads. The first reason is because of GPU intervention between processing and model training, which can lead to the local GPU bottleneck. For instance, in Transformer ASR and CTC ASR, FastFlow shows 11% speedup compared to DALI in 7 vCPUs, where the performance gap further increases as CPU resources increase (up to 41%). In addition, DALI degrades performance in GAN ADA because the data pipeline is partially converted into the GPU operations. As a result, tensors produced in the GPU need to be copied again to the CPU for further preprocessing, which incurs additional overheads. DALI is beneficial than FastFlow only when the performance benefit of remote nodes is smaller than that of GPUs (e.g., (a) 7:PRD).

Aside from the performance, DALI has several limitations in usability. It is non-trivial to migrate data pipelines originally written by tf.data to DALI due to the limited operations supported and incompatible interfaces. Accordingly, it is not possible to implement data pipelines using DALI for Speaker Recognition (no support for Fourier transform operator [8]) and RandAugment (no support for several image transformations including `sharpness` function [1]). In addition, DALI could also suffer from GPU OOM and slowdown due to resource contention between preprocessing and model training for workloads having high prep stalls, so users must judiciously use GPU resources for preprocessing.

7.3 Performance Effect of Offloading Decisions

As described in the previous section, FastFlow outperforms the baselines in various applications and resource environments. In this section, we show the performance effect of offloading decisions. In the remaining sections, we show the evaluation results on the 7:7 environment by default.

When to Offload. FastFlow automatically decides to offload only if there is a prep stall. For example, FastFlow prevents offloading of MelGAN and Learnable Resizer in the 7:x and 14:7 environments, and GAN ADA and RandAugment in the 14:7 environment because they have negligible prep stalls.

Which Operations to Offload. To decide which operations to offload, FastFlow compares the three candidate pipelines as illustrated in Figure 5. Figure 8 shows the epoch time when each input data pipeline is selected for offloading. In the 7:7 PCS environment, ***p'2*** always leads to better performance than ***p'3*** due to the threading overheads and than ***p'1*** because the disk I/O is not the bottleneck. The thread overheads differ from applications, and applications with CPU intensive operations have high threading overheads. For instance, Transformer ASR has the highest CPU cycles for preprocessing among the applications, so choosing ***p'3*** significantly degrades the performance due to the threading overheads in the PCS container.

[6] A PCS host node has 32 cores with hyperthreading (64 vcores)

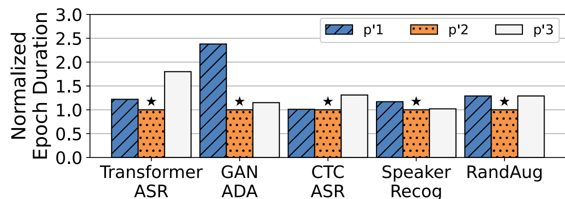

Figure 8: Impact of the offloading operator selection of FastFlow. * mark shows the selection of FastFlow.

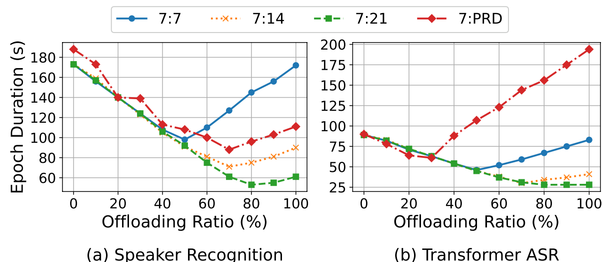

Figure 9: Epoch duration in various offloading ratios on different remote workers. (a) is compute-intensive, and (b) is data-intensive preprocessing workload.

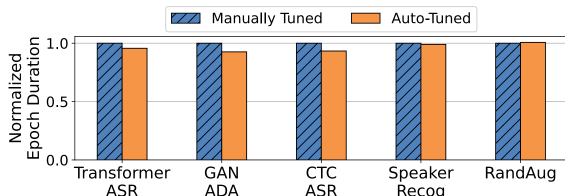

Figure 10: The offloading ratio decision accuracy of FastFlow.

When the resource environment changes, FastFlow can make different choices for the same application. For instance, when executing Transformer ASR on the PRD node, FastFlow chooses ***p'3*** instead of ***p'2*** because the threading overheads are negligible in the PRD node (30 operator threads are created on 30 CPU cores), and the network does not become the bottleneck with partial offloading. In addition, when we evaluate the same application on the resource environment where the remote node reads data from the disk with low I/O bandwidth (e.g., HDD), FastFlow chooses ***p'1*** to avoid the disk read bottleneck from the remote node.

How Much Data to Offload. To evaluate whether FastFlow finds the optimal data offloading ratio, we first measure the epoch time by manually varying the partial offloading ratio from 0% to 100% for every 10% increment. As representative examples, we select Speaker Recognition and Transformer ASR, which is a CPU-intensive and data-transfer-intensive workload, respectively. FastFlow chooses the best data pipeline for operator selection as shown in Figure 8.

Figure 9 shows the epoch time in different offloading ratios. For CPU-intensive workloads, the optimal offloading ratio increases as the number of remote CPUs increases to utilize the remote CPU powers and to prevent the local CPU bottleneck. As shown in Figure 9(a), offloading 50% of data is optimal in 7:7, but in 7:21, 80% of data needs to be preprocessed for the optimal performance. For data-transfer-intensive workloads, the optimal offloading ratio decreases as the network bandwidth decreases to prevent the network bottleneck. Figure 9(b) shows that only about 30% of data needs to be offloaded to the PRD for optimal performance.

We compare FastFlow's epoch time with the manually tuned results for decision accuracy. As illustrated in Figure 10, FastFlow successfully finds the optimal point for the minimum epoch time. In this graph, the epoch time of FastFlow is slightly lower than that of the manually-tuned results. This is because we increment the offloading ratio by every 10% to find the optimal point, but FastFlow automatically finds the optimal point more precisely than the manual selection based on the profiled metrics.

Table 4: Profiling overheads of FastFlow in various applications. Fraction is $\frac{\text{Profile time}}{\text{Train time}} * 100$. Values with * are the results with metric reload.

Job	Profile time (min)	Train time (min)	Fraction (%)
T	1.91 / 0.08*	68.37	**2.80 / 0.11***
G	0.61 / 0.08*	162.22	**0.37 / 0.05***
C	9.27 / 0.20*	299.32	**3.10 / 0.07***
S	1.28 / 0.02*	150.58	**0.85 / 0.01***
R	7.01 / 0.24*	11396.50	**0.06 / 0.002***
M	0.78 / 0.53*	933.35	**0.08 / 0.06***
L	0.65 / 0.26*	49.23	**1.31 / 0.53***

7.4 Profiling Overhead

To evaluate the profiling overheads that FastFlow incurs, we measure the fraction of profiling time to end-to-end training time. To clarify, profiling time includes the time of model deep-copy (§ 4), measuring metrics, and scanning the same graph architecture of the input pipeline and model for metric store/reload. For training time, we measure the elapsed time until the models reach the target accuracy as suggested in their papers [28, 30, 35, 38, 42, 45, 52]. Table 4 shows that profiling takes only from 0.06% to 3.1% of training time without metric reload. Considering the speed-up that FastFlow achieves, profiling overheads are negligible. Workloads with low prep stalls such as MelGAN and Learnable Resizer also have little overheads as FastFlow prevents measuring other metrics after a short profiling of *Lthp* and *Gthp*, which takes a little fraction of time. FastFlow further reduces the profiling time by 81.1% on average with the metric saving and reloading when the same input pipeline or model is trained again. Still, there is a little overhead for identifying the same input pipeline and model, but it is negligible.

7.5 Effect on Model Convergence

To demonstrate that FastFlow accelerates training speed without affecting learning algorithms, we measure the loss and training time until models converge. We then compare the result of TF-NO with that of FastFlow. In this evaluation, we show the experimental result of the CTC ASR due to the space constraint, but we already observed that other workloads produce similar results. Figure 11 shows the training time and Word Error Rate (WER) of CTC ASR. In this graph, FastFlow achieves 40% reduction of the training time to reach the desirable Word Error Rate (0.16 as described in [4]) compared to TF-NO. The result proves that our smart offloading technique boosts the training without affecting the model convergence, and also indicates that the reduction of the epoch time directly affects

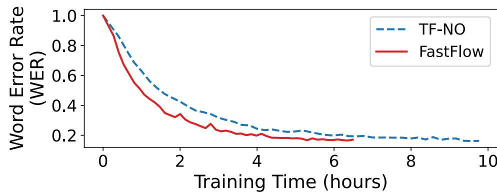

Figure 11: Model convergence of CTC ASR.

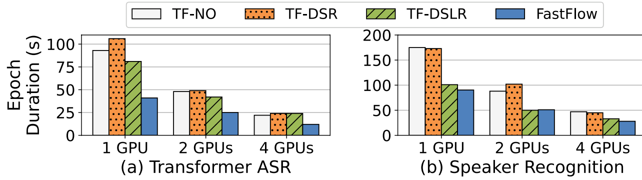

Figure 12: Multi-GPU training for (a) Transformer ASR and (b) Speaker Recognition.

the total training time (FastFlow also reduces the CTC ASR epoch time by 40% than TF-NO in the 7:7 environment).

7.6 Multi-GPU Training

Our evaluation shows that FastFlow is still effective on multiple GPUs. While keeping the vCPU:GPU ratio to 7:1, we train models by increasing the number of GPUs from 1 to 4 with synchronous data parallelism strategy [7]. To keep up with the increased parallelism on the GPU-side, we set the same number of remote PCS vCPUs with the local vCPUs (from 7 to 28). For evaluation, we select Transformer ASR and Speaker Recognition because they have high prep stalls. Figure 12 shows the results on the applications. Although the speedup slightly decreases as the number of GPU increases due to the synchronization overheads of weight updates in the multi-GPU training, FastFlow offloads preprocessing well by efficiently using both local and remote CPU resources. As a result, compared to TF-NO, FastFlow improves the performance of Transformer ASR by 2.27×, 1.92× and 1.83×, and Speaker Recognition by 1.89×, 1.73× and 1.67× on 1, 2 and 4 GPUs, respectively.

8 RELATED WORK

In this section, we discuss related works except for the work discussed in § 2.2 that handles preprocessing stalls.

Bottleneck Analysis. PRESTO [36] analyzes the trade-off of throughput, preprocessing time, and storage consumption by individually caching preprocessing operators. With this analysis, we can determine which preprocessing to be performed on offline preprocessing (only once), or on online preprocessing (every epoch) based on the analyzed throughput. DS-Analyzer [47] analyzes fetch stall and preprocessing stall, with the comparison of I/O, CPU, and GPU speed. Further from analysis, FastFlow makes automatic decisions for offloading according to workloads and resource environments.

Dynamic-CPU Allocation. Synergy [46] addresses the problem of CPU bottlenecks in DL training on shared GPU clusters. Synergy allows each job to acquire a different number of CPUs per GPU and develops scheduling policies to allocate the appropriate CPUs for each job. However, when the total amount of CPUs required by the multiple jobs is beyond the CPUs of the shared GPU cluster, the GPU cluster will suffer from the CPU bottleneck. FastFlow can resolve the CPU bottleneck of the shared GPU cluster by offloading input pipeline to remote CPU clusters and accelerates DL jobs by considering the remote CPU capacity and network bandwidth.

Caching. Cachew [33] enables the autocaching of a job by allowing users to set an `autocache` operator before random operations to reduce deterministic-preprocessing overheads. In multi-user and multi-job environments, Quiver [41] reduces I/O overheads with workload-aware caching techniques. These caching techniques can be integrated with FastFlow, which focuses on prep stalls caused by intensive CPU uses of diverse (random) operations.

Domain-Specific Preprocessing. DIESEL+ [54] optimizes the entire image input pipeline with domain-specific knowledge. Meta [57] shows the characteristics of datacenter-scale input data pipeline for recommendation models. PCR [39] proposes an efficient compression for images to reduce the overhead of fetching and decoding. FastFlow can accelerate and scale out general operations without domain-specific knowledge.

9 DISCUSSION

Although FastFlow targets resource environments where each job runs on a fixed amount of allocated resources, we believe that FastFlow can be easily extended to disaggregated or cloud environments where CPU resources for offloading are abundant and on-demand resource allocation is supported. In this section, we discuss how to support auto-scaling in FastFlow on the environments.

To automatically decide the number of workers to offload before the actual training, FastFlow can use the profiled metrics. For instance, for homogeneous workers, FastFlow can estimate the required number of workers by measuring $Rthp$ with one worker. The number of workers for offloading can be simply derived by $n_workers = \lceil Lower/Rthp_{a_worker} \rceil$. Compared to the auto-scaling of Cachew [33] that dynamically adjusts the number of workers to find the optimal performance, FastFlow can find the optimal number of workers at one time with the profiled metrics. The profile-based autoscaling can also be easily integrated with the Cachew's dynamic autoscaling by measuring performance at runtime and adjusting the number of workers from the number decided by the prior profiling.

10 CONCLUSION

We design FastFlow, a DL training system that carefully offloads DL input pipeline to remote CPUs for performance improvement based on lightweight profiling. Our evaluation with seven diverse workloads shows that FastFlow always achieves comparable or better performance than the manual configuration of TensorFlow for offloading and than DALI by automatically finding the best offloading decisions—when, which, and how much data to offload—in diverse resource environments. FastFlow can easily be adopted by TensorFlow's users with minimal code modification, and we believe that FastFlow can improve not only training throughput, but also resource utilization of both GPU and CPU clusters.

ACKNOWLEDGMENTS

We thank all reviewers for their insightful comments. We also gratefully acknowledge Kyungjae Kim, Jaehun Uhm, and Yunsu Lee for helping us to initiate and complete this work.

REFERENCES

[1] Accessed in January 2023. Add more choices for data augmentation. https://github.com/NVIDIA/DALI/issues/1610.

[2] Accessed in January 2023. Amazon EC2 P3 Instances. https://aws.amazon.com/ec2/instance-types/p3/.

[3] Accessed in January 2023. Amazon S3. https://aws.amazon.com/s3.

[4] Accessed in January 2023. Automatic Speech Recognition using CTC. https://keras.io/examples/audio/ctc_asr/.

[5] Accessed in January 2023. Automatic Speech Recognition with Transformer. https://keras.io/examples/audio/transformer_asr/.

[6] Accessed in January 2023. Data-efficient GANs with Adaptive Discriminator Augmentation. https://keras.io/examples/generative/gan_ada/.

[7] Accessed in January 2023. A distribution strategy for synchronous training on multiple workers. https://www.tensorflow.org/api_docs/python/tf/distribute/experimental/MultiWorkerMirroredStrategy.

[8] Accessed in January 2023. Does DALI support GPU operation equivalent to tf.signal.fft? https://github.com/NVIDIA/DALI/issues/4331.

[9] Accessed in January 2023. Google TPU v4. https://cloud.google.com/tpu/docs/system-architecture-tpu-vm#tpu_v4.

[10] Accessed in January 2023. Keras: Deep Learning for humans. https://github.com/keras-team/keras.

[11] Accessed in January 2023. Kubernetes: an open source system for managing containerized applications across multiple hosts. https://github.com/kubernetes/kubernetes.

[12] Accessed in January 2023. Learning to Resize in Computer Vision. https://keras.io/examples/vision/learnable_resizer/.

[13] Accessed in January 2023. MelGAN-based spectrogram inversion using feature matching. https://keras.io/examples/audio/melgan_spectrogram_inversion/.

[14] Accessed in January 2023. Module: tf.data.experimental.service. https://www.tensorflow.org/api_docs/python/tf/data/experimental/service.

[15] Accessed in January 2023. NVIDIA Data Loading Library (DALI). https://developer.nvidia.com/dali.

[16] Accessed in January 2023. NVIDIA H100 Performance. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/.

[17] Accessed in January 2023. psutil (process and system utilities): a cross-platform library for retrieving information on running processes. https://github.com/giampaolo/psutil.

[18] Accessed in January 2023. Pytorch Dataloader. https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader.

[19] Accessed in January 2023. Speaker Recognition. https://keras.io/examples/audio/speaker_recognition_using_cnn/.

[20] Accessed in January 2023. Speaker Recognition Dataset. https://www.kaggle.com/datasets/kongaevans/speaker-recognition-dataset.

[21] Accessed in January 2023. tf.data.service commit that does not prefer local reads. https://github.com/tensorflow/tensorflow/commit/17e7f5e01bbcdde893309bc302f144e25d43bb81.

[22] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *OSDI*. 265–283.

[23] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, USVSN Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. 2022. GPT-NeoX-20B: An Open-Source Autoregressive Language Model.

[24] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[25] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[26] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. https://arxiv.org/abs/2107.03374

[27] Yang Cheng, Dan Li, Zhiyuan Guo, Binyao Jiang, Jinkun Geng, Wei Bai, Jianping Wu, and Yongqiang Xiong. 2021. Accelerating End-to-End Deep Learning Workflow With Codesign of Data Preprocessing and Scheduling. *IEEE TPDS* 32, 7 (2021), 1802–1814.

[28] Joon Son Chung, Arsha Nagrani, and Andrew Zisserman. 2018. VoxCeleb2: Deep Speaker Recognition. In *Interspeech*. 1086–1090.

[29] Ekin D. Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V. Le. 2019. AutoAugment: Learning Augmentation Strategies From Data. In *CVPR*.

[30] Ekin D. Cubuk, Barret Zoph, Jonathon Shlens, and Quoc V. Le. 2020. Randaugment: Practical Automated Data Augmentation With a Reduced Search Space. In *CVPR*.

[31] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *CVPR*.

[32] Jeremy Elson, John (JD) Douceur, Jon Howell, and Jared Saul. 2007. Asirra: A CAPTCHA that Exploits Interest-Aligned Manual Image Categorization. In *ACM CCS*.

[33] 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 22)*. 689–706.

[34] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. 2013. Speech recognition with deep recurrent neural networks. In *IEEE international conference on acoustics, speech and signal processing*. 6645–6649.

[35] Yosuke Higuchi, Shinji Watanabe, Nanxin Chen, Tetsuji Ogawa, and Tetsunori Kobayashi. 2020. Mask CTC: Non-Autoregressive End-to-End ASR with CTC and Mask Predict. In *Interspeech*. 3655–3659.

[36] Alexander Isenko, Ruben Mayer, Jeffrey Jedele, and Hans-Arno Jacobsen. 2022. Where Is My Training Bottleneck? Hidden Trade-Offs in Deep Learning Preprocessing Pipelines. In *SIGMOD*.

[37] Keith Ito and Linda Johnson. 2017. The LJ Speech Dataset. https://keithito.com/LJ-Speech-Dataset/.

[38] Tero Karras, Miika Aittala, Janne Hellsten, Samuli Laine, Jaakko Lehtinen, and Timo Aila. 2020. Training Generative Adversarial Networks with Limited Data. In *NeurIPS*. 12104–12114.

[39] Michael Kuchnik, George Amvrosiadis, and Virginia Smith. 2021. Progressive Compressed Records: Taking a Byte out of Deep Learning Data. *Proc. VLDB Endow.* 14, 11 (2021), 2627–2641.

[40] Michael Kuchnik, Ana Klimovic, Jiri Simsa, Virginia Smith, and George Amvrosiadis. 2022. Plumber: Diagnosing and Removing Performance Bottlenecks in Machine Learning Data Pipelines. *Proceedings of Machine Learning and Systems* 4 (2022).

[41] Abhishek Vijaya Kumar and Muthian Sivathanu. 2020. Quiver: An Informed Storage Cache for Deep Learning. In *FAST*. 283–296.

[42] Kundan Kumar, Rithesh Kumar, Thibault de Boissiere, Lucas Gestin, Wei Zhen Teoh, Jose Sotelo, Alexandre de Brébisson, Yoshua Bengio, and Aaron C Courville. 2019. MelGAN: Generative Adversarial Networks for Conditional Waveform Synthesis. In *NeurIPS*.

[43] Gyewon Lee, Irene Lee, Hyeonmin Ha, Kyunggeun Lee, Hwarim Hyun, Ahnjae Shin, and Byung-Gon Chun. 2021. Refurbish Your Training Data: Reusing Partially Augmented Samples for Faster Deep Neural Network Training. In *ATC*. 537–550.

[44] Youngjune Lee, Oh Joon Kwon, Haeju Lee, Joonyoung Kim, Kangwook Lee, and Kee-Eung Kim. 2021. Augment & Valuate: A Data Enhancement Pipeline for Data-Centric AI. *arXiv preprint arXiv:2112.03837* (2021).

[45] Abdelrahman Mohamed, Dmytro Okhonko, and Luke Zettlemoyer. 2019. Transformers with convolutional context for asr. *arXiv preprint arXiv:1904.11660* (2019).

[46] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. 2022. Looking Beyond GPUs for DNN Scheduling on Multi-Tenant Clusters. In *OSDI*. 579–596.

[47] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. 2021. Analyzing and Mitigating Data Stalls in DNN Training. *Proc. VLDB Endow.* 14, 5 (2021), 771–784.

[48] Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. 2021. Tf.Data: A Machine Learning Data Processing Framework. *Proc. VLDB Endow.* 14, 12 (2021), 2945–2958.

[49] Pyeongsu Park, Heetaek Jeong, and Jangwoo Kim. 2020. TrainBox: An Extreme-Scale Neural Network Training Server Architecture by Systematically Balancing Operations. In *MICRO*. 825–838.

[50] 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 *NeurIPS*.

[51] Robin Scheibler, Eric Bezzam, and Ivan Dokmanić. 2018. Pyroomacoustics: A Python Package for Audio Room Simulation and Array Processing Algorithms. In *IEEE ICASSP*. 351–355.

[52] Hossein Talebi and Peyman Milanfar. 2021. Learning to resize images for computer vision tasks. In *ICCV*. 497–506.

[53] Catherine Wah, Steve Branson, Peter Welinder, Pietro Perona, and Serge Belongie. 2011. The caltech-ucsd birds-200-2011 dataset. (2011).

[54] Lipeng Wang, Qiong Luo, and Shengen Yan. 2022. DIESEL+: Accelerating Distributed Deep Learning Tasks on Image Datasets. *IEEE TPDS* 33 (2022), 1173–1184.

[55] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *NSDI*. 945–960.

[56] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A {Fault-Tolerant} Abstraction for {In-Memory} Cluster Computing. In *NSDI*. 15–28.

[57] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. 2022. Understanding Data Storage and Ingestion for Large-Scale Deep Recommendation Model Training: Industrial Product. In *ISCA*. 1042–1057.