Analyzing and Mitigating Data Stalls in DNN Training

Jayashree Mohan*
University of Texas at Austin
jaya@cs.utexas.edu

Amar Phanishayee
Microsoft Research
amar@microsoft.com

Ashish Raniwala
Microsoft
ashish.raniwala@microsoft.com

Vijay Chidambaram
University of Texas at Austin & VMWare Research
vijay@cs.utexas.edu

ABSTRACT

Training Deep Neural Networks (DNNs) is resource-intensive and time-consuming. While prior research has explored many different ways of reducing DNN training time, the impact of *input data pipeline*, i.e., fetching raw data items from storage and performing data pre-processing in memory, has been relatively unexplored. This paper makes the following contributions: (1) We present the first comprehensive analysis of how the input data pipeline affects the training time of widely-used computer vision and audio Deep Neural Networks (DNNs), that typically involve complex data pre-processing. We analyze nine different models across three tasks and four datasets while varying factors such as the amount of memory, number of CPU threads, storage device, GPU generation etc on servers that are a part of a large production cluster at Microsoft. We find that in many cases, DNN training time is dominated by *data stall time*: time spent waiting for data to be fetched and pre-processed. (2) We build a tool, DS-Analyzer to precisely measure data stalls using a differential technique, and perform predictive what-if analysis on data stalls. (3) Finally, based on the insights from our analysis, we design and implement three simple but effective techniques in a data-loading library, CoorDL, to mitigate data stalls. Our experiments on a range of DNN tasks, models, datasets, and hardware configs show that when PyTorch uses CoorDL instead of the state-of-the-art DALI data loading library, DNN training time is reduced significantly (by as much as 5× on a single server).

PVLDB Reference Format:
Jayashree Mohan*, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and Mitigating Data Stalls in DNN Training. PVLDB, 14(5): 2021.
doi:10.14778/3446095.3446100

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/msr-fiddle/DS-Analyzer.

Proceedings of the VLDB Endowment, Vol. 14, No. 5 ISSN 2150-8097.
doi:10.14778/3446095.3446100

*Work done as part of MSR internship.

1 INTRODUCTION

Data is the fuel powering machine learning [71]. Large training datasets are empowering state-of-the-art accuracy for several machine learning tasks. Particularly, Deep Neural Networks (DNNs), have gained prominence, as they allow us to tackle problems that were previously intractable, such as image classification [44, 57, 83], translation [90], speech recognition[41], video captioning [88], and even predictive health-care [85].

Empowering DNNs to push state-of-the-art accuracy requires the model to be trained with a large volume of data. During training, the model predicts the output given training data; based on the output, the model's weights are tuned. This happens iteratively, in many rounds called epochs.

However, DNN training is data-hungry, resource-intensive, and time-consuming. It involves the holistic use of all the resources in a server from storage and CPU for fetching and pre-processing the dataset to the GPUs that perform computation on the transformed data. Researchers have tackled how to efficiently use these resources to reduce DNN training time, such as reducing communication overhead [43, 50, 63, 69, 92], GPU memory optimizations [29, 49, 77], and compiler-based operator optimizations [28, 52, 87]. However, the impact of storage systems, specifically the *data pipeline*, on DNN training has been relatively unexplored.

The DNN Data Pipeline. During DNN training, the data pipeline works as follows. Data items are first fetched from storage and then *pre-processed* in memory. For example, for many important and widely-used classes of DNNs in computer vision, there are several pre-processing steps: data is first decompressed, and then random perturbations such as cropping the image or rotating it are performed to improve the model's accuracy [74]. Once pre-processed, the data items are sent to the GPUs for processing. One complete pass over the training dataset is termed an epoch; models are iteratively trained for several epochs to achieve desired accuracy.

The DNN data pipeline operates in parallel with GPU computation. Ideally, the data pipeline should steadily feed pre-processed data items to the GPUs to keep them continuously busy processing data; we term this GPU-bound. Unfortunately, DNN training is often I/O-bound, bottlenecked by fetching the data from storage, or CPU-bound, bottlenecked by applying data pre-processsing in memory. Collectively, we term these bottlenecks **data stalls** and differentiate between *prep stalls* (time spent on data pre-processing) and *fetch stalls* (time spent on I/O).

Novelty over prior work. Recent work like Quiver [58] demonstrate how to speed up DNN training by efficiently caching data from remote object storage on local storage. In contrast, our work

Table 1: Key findings and implications of our analysis of data stalls

Finding	*Insights*
OS Page Cache is inefficient for DNN training due to thrashing	DNN-aware caching can eliminate thrashing across epochs
DNNs need anywhere between 3 – 24 CPU cores *per GPU* for data pre-processing	If hardware is upgraded to overcome workload bottlenecks, it must be done carefully with an eye towards designing balanced server SKUs.
DNNs spend upto 65% of the epoch time in data pre-processing, primarily on redundant decoding	Decoded data can be cached (as opposed to caching encoded data), if space amplification due to decoding can be addressed
Lack of coordination among local caches lead to redundant I/O in distributed training across servers	To overcome local storage I/O bottlenecks, local in-memory caches of servers allocated to a job can be coordinated to fetch data from distributed in-memory caches
Hyperparameter search workloads perform redundant I/O & prep	Hyperparameter search jobs must coordinate data fetch & prep to mitigate data stalls

assumes as baseline the setting where all data is available on local storage (as most publicly available datasets for popular models fit on local storage [10, 23, 25, 35, 59, 78], §4.3.1), and shows how to further optimize the data pipeline (§4.3.2, §4.3.3). This work therefore fundamentally improves performance on top of what Quiver can achieve. While prior work like Cerebro [68], and DeepIO [95] have looked at optimizing data fetch in distributed training, they do not systematically analyze data stalls in different training scenarios, or demonstrate how to accelerate single-server training.

1.1 Contributions

Categorizing, measuring, and analyzing data stalls. We present the first comprehensive analysis of data stalls (categorized as fetch and prep stalls) in DNN training. We analyze nine popular DNN models from three domains (image classification, object detection, and audio classification) and four datasets in a production cluster at Microsoft. We vary factors such as the storage media, amount of data that can be cached in memory, the number of CPU threads used to fetch and pre-process data, and GPU generation. We then analyze how these factors affect the data pipeline and DNN training. Our analysis finds that data stalls squander away the improved performance of faster GPUs, even on ML optimized servers like the DGX-2 [14]. Revisiting the insights from Stonebraker *et al.* [84], our analysis corroborates that relying on OS abstractions (like Page Cache) is inefficient for DNN workloads. We also find that the data pipelines in popular training frameworks like PyTorch and TensorFlow are inefficient in their use of CPU and memory resources, despite using state-of-the-art data-loading libraries like DALI [8] that reduce prep stalls using GPU-accelerated data pre-processing. Table 1 summarizes the findings and insights of our analysis.

Performing predictive what-if analysis of data stalls. Performing an analysis of how the data pipeline impacts DNN training is challenging since DNN training has a high degree of concurrency; it is hard to isolate the time taken to perform a single task especially as data prefetching and pre-processing are pipelined with GPU computation. We develop a tool, DS-Analyzer, that uses differential analysis between runs (*e.g.,* comparing a run where data is completely cached vs when data needs to be fetched from storage) to accurately identify data-stall bottlenecks. Using the measured data stalls, DS-Analyzer answers what-if questions to help practitioners predict and analyze data stalls (*e.g.,* What would be the impact on data stalls if DRAM capacity increased by 2×?).

Mitigating data stalls. We use the insights from our analysis to identify opportunities for improvement. We build a new data-loading library, CoorDL, that does not require any changes to the cluster infrastructure. CoorDL uses three main techniques to mitigate data stalls. First, inspired by the pioneering work of Stonebraker *et al.* on database caching [84], we demonstrate that relying on the OS page cache is sub-optimal for DNN training. We implement MinIO, a software cache that is specialized for DNN training. Second, we describe the *partitioned caching* technique to coordinate the MinIO caches of servers involved in distributed training over commodity network stack. Third, we discuss the *coordinated prep* technique to carefully eliminate redundancy in data prep among concurrent hyperparameter search jobs in a server. We implement these techniques as part of the user-space library CoorDL, built on top of the state-of-the-art data pipeline DALI [8]. We evaluate CoorDL across different models, datasets, and hardware and show that it can accelerate training by up-to 5× on a single server by mitigating data stalls over DALI.

2 BACKGROUND

Deep Neural Networks (DNNs) are a class of ML models that automatically extract higher level features from the input data. The DNN is trained over multiple rounds termed *epochs*. Each epoch processes all items in the dataset exactly once, and consists of multiple *iterations*; each iteration processes a random, disjoint subset of the data termed a *minibatch*. The DNN is trained until a target accuracy is reached. Training a DNN model to reach a given accuracy consists of two steps:

(1) **Hyperparameter (HP) search**. There are many parameters for the learning algorithm that must be provided before the start of training. These hyperparameters (for *e.g.,* learning rate, its decay, dropout, and momentum) influence the speed and quality of learning. During the search process, we start several training jobs; each job trains the model with different hyperparameters, on each available GPU (or a distributed job across several GPUs); progress is checked after a few epochs and the worst-performing candidates are killed and replaced by new jobs with different hyperparameters that are chosen algorithmically [26, 38, 48, 60]. Tuning hyperparameters is crucial for generating DNN models that have high accuracy [75].

(2) **Training the model to target accuracy**. The second step is to obtain models with high accuracy by training it with input data, using the hyperparameters chosen in the previous step.

2.1 The DNN ETL Requirements

In every epoch of training, the input dataset is subjected to a ETL (extract-transform-load) before being processed at the GPU (or any other accelerator). The ETL process in the data pipeline of DNN

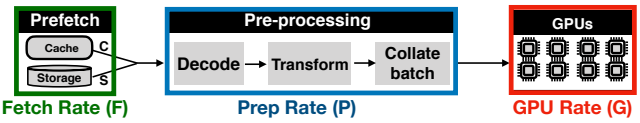


Figure 1: Data Pipeline in DNN training. This figure shows the different stages in the data pipeline.

training imposes several unique data ordering constraints to ensure model convergence and achieve state-of-the-art accuracy.

- The dataset must be shuffled every epoch to ensure the order in which data items are accessed are random in each epoch
- An epoch must use *all* data items in the dataset *exactly* once
- In every epoch, the data transformations(pre-processing) must be random; the same transformed item should not be used across epochs

Several prior work have theoretically and empirically demonstrated that relaxing these constraints will affect the convergence rate of SGD [31, 62, 66, 74]. Therefore, in this work, all our experiments abide by the aforementioned ETL requirements.

2.2 DALI : Fast Data Pipelining

State-of-the-art data loading and pre-processing libraries like DALI can be used as a drop in replacement for the default dataloaders in frameworks like PyTorch, TensorFlow, or MxNet. DALI accelerates data pre-processing operations using GPU-accelerated data pre-processing operations. DALI also *prefetches and pipelines* the data fetch and pre-processing with the GPU compute, similar to the default dataloader in PyTorch. We empirically verified that DALI outperforms the default data pipelines of PyTorch, TensorFlow, and MxNet. Therefore, throughout this work, unless and otherwise stated, we use DALI, as it is the strongest baseline.

3 DATA STALLS IN DNN TRAINING

We now discuss our formulation of data stalls. Consider the training process of a typical DNN. It executes the following steps in each iteration of an epoch:

(1) A minibatch of data items is fetched from storage.
(2) The data items are pre-processed, for *e.g.*,, for image classification, data items are decompressed, and then randomly cropped, resized, and flipped.
(3) The minibatch is then processed at the GPU to obtain the model's prediction
(4) A loss function is used to determine how much the prediction deviates from the right answer
(5) Model weights are updated using computed gradients

Ideally, most of the time in each epoch should be spent on Steps 3–5 (which we collectively term the *GPU compute* time), i.e., training is *GPU bound*. When performing multi-GPU training, individual GPUs (workers) exchange weight gradients with other workers before performing weight update. For this work, we roll the communication time for gradient exchange during multi-GPU training into computation time.

In most frameworks including PyTorch, TensorFlow, and MxNet, data preparation (Steps 1 and 2) and GPU computation execute in a pipelined fashion; i.e., subsequent minibatches are *prefetched and pre-processed* by data preparation threads, using multiple CPU cores on the machine, as the GPU computes on the current minibatch of data. If the GPU is waiting for Steps 1–2 to happen, we term it a *data stall*. Specifically, if training is blocked on Step 1, we call it a fetch stall; the training is *I/O bound* in this case. Training blocked due to Step 2 is termed prep stall; this causes the training to be *CPU bound*. Data stalls cause the GPU to be idle, and must be minimized to increase GPU utilization.

Table 2: Models and datasets used in this work.

Task	Model	Dataset (Size)
Image Classification	Shufflenetv2 [93]	ImageNet-22k [10] (1.3TB) OpenImages-Extended [59, 81] (645GB) Imagenet-1k [78] (146GB)
	AlexNet [57]	
	Resnet18 [44]	
	SqueezeNet [46]	
	MobileNetv2 [80]	
	ResNet50 [44]	
	VGG11 [83]	
Obj Detection	SSD+Res18 [64]	OpenImages [59] (561GB)
Audio Classify	M5 [33]	Free Music [35] (950GB)

The rate at which data items can be fetched from storage (Step 1) depends primarily on the storage media. The rate at which data items can be pre-processed (Step 2) depends upon the pre-processing operations and the number of CPU cores available for pre-processing.

In general, if we prefetch data at rate F, pre-process it at rate P and perform GPU computation on it at rate G, then data stalls appear if $G > min(F, P)$, i.e., GPU processes data at a rate faster than it can be prefetched or pre-processed.

Any fetch or prep stall implies idle GPU time, which must be minimized. The fetch and prep stalls reported in this work are unmasked stall time; i.e., the stall time that shows up in the critical path, inspite of being pipelined with compute. From now on, we call data prefetching simply *fetch*, and pre-processing *prep*.

4 ANALYZING DATA STALLS

To understand data stalls in DNN training and the fundamental reasons why data stalls exist, we perform a comprehensive analysis on several DNNs by varying a number of factors, such as the number of GPUs, GPU generation, the size of the DRAM cache, the number of CPU threads etc.

4.1 Methodology

Models and Datasets. We analyze **nine** state-of-the-art DNN models across three different tasks and four different datasets as shown in Table 2. This section focuses on the smaller ImageNet-1K dataset for image classification models. Evaluation with large datasets like ImageNet-22k and OpenImages is presented in Section §7. The image and audio classification models are taken from TorchVision [22] and TorchAudio [21] respectively; for object detection, we use NVIDIA's official release of SSD300 v1.1 [16].

Pre-processing. For all DNNs, we use the same pre-processing as in their original papers. More precisely, for the image classification task, pre-processing includes image decoding, random crop, resizing

Table 3: Server configurations used. We use two SKUs; each server has 24 CPU cores, 500GiB DRAM, and 8 GPUs.

	GPU Config	GPU Mem(GB)	Storage Media	Rand Read (MBps)
SSD-V100	8xV100	32	SSD	530
HDD-1080Ti	8x1080Ti	11	HDD	15 – 50

to a fixed size, and a random horizontal flip of the image. The object detection task performs a color twist of the image, and a random crop and horizontal flip of the bounding box in addition to the image transformations described for image classification. The audio model decodes and down-samples input to 8kHz.

Training environment. All experiments are performed on PyTorch 1.1.0 using the state-of-the-art NVIDIA data loading pipeline, DALI. We have empirically verified that DALI's performance is strictly better than PyTorch, TF and MxNet's default data loaders; therefore we perform our analysis of data stalls using the strongest baseline, DALI. We use two distinct server configurations for our analysis as shown in Table 3. Both these are part of a large production and research cluster at Microsoft [11, 51], whose workload have guided the design of several research systems for ML training [27, 42, 65, 91]. These servers also closely resemble publicly available cloud GPU SKUs [1, 2]. `Config-SSD-V100` has configuration closest to AWS p3.16xlarge [1] with gp2 storage [7], while `Config-HDD-1080Ti` is closest to AWS p2.8xlarge [2] with st1 storage [7]. Both our servers have 500GB DRAM, 24 physical CPU cores , and 8 GPUs per server.

Training parameters. For experiments on `Config-SSD-V100`, we use a batch size of 512 per GPU for all image classification models, 128 per GPU for SSD-Res18, 16 per GPU for M5 and perform weak scaling for distributed training (while ensuring that the global batch size is consistent with those widely used in the ML community). Since V100 GPUs have tensor cores, we use Apex mixed precision training with LARC (Layer-wise Adaptive Rate Clipping), and state-of-the art learning rate warmup schedules [40]. On `Config-HDD-1080Ti`, we use the maximum batch size that fits the GPU memory (less than 256 for all models) and perform full-precision training.

Training metrics. We run all the experiments presented here for three epochs, and report the average epoch time (or throughput in samples per second), ignoring the first epoch. Since we start with a cold cache in our experiments, first epoch is used for warmup. Measuring data stall time does not require training to accuracy; per-epoch time remains stable.

4.2 Measuring data stalls using DS-Analyzer

We develop a standalone tool, *DS-Analyzer* that profiles data stalls in DNN training. Frameworks like PyTorch and TensorFlow provide an approximate time spent on data loading and pre-processing per minibatch, by simply placing timers in the training script. This is insufficient and inaccurate for two reasons. First, this technique cannot accurately provide the split up of time spent in data fetch (from disk or cache) and pre-processing operations. To understand if the training is bottlnecked on I/O or CPU, it is important to know this split. Second, frameworks like PyTorch and libraries like DALI use several concurrent processes (or threads) to fetch and pre-process data; for a multi-GPU data parallel training job, a data stall in one of the data loading processes may reflect as GPU compute time for the other processes, because all GPU processes wait to synchronize weight updates at batch boundaries. Naively adding timers around data path does not provide accurate timing information. Therefore, DS-Analyzer uses a differential approach. DS-Analyzer runs in three phases;

(1) **Measure ingestion rate**. First, DS-Analyzer pre-populates synthetic data at the GPUs and runs the job for a fixed number of epochs. This identifies the max data ingestion rate at the GPUs, with no fetch or prep stalls.

(2) **Measure prep stalls**. Next, DS-Analyzer runs the training script with a subset of the given dataset, such that it is entirely cached in memory, using all available CPU cores, and estimates the training speed. Since this run eliminates fetch stalls, any drop in throughput compared to (1) is due to prep stalls.

(3) **Measure fetch stalls**. Finally, DS-Analyzer runs the training script by clearing all caches, and setting maximum cache size to a user-given limit, to account for fetch stalls. The difference between (2) and (3) is the impact of fetch stalls.

4.3 Data Stalls in DNN Training

Our analysis aims to answer the following questions:

Fetch Stalls (Remote)	Is remote storage a bottleneck for training?	§4.3.1
Fetch Stalls (Local)	When does the local storage device (SSD/HDD) become a bottleneck for DNN training?	§4.3.2
Prep Stalls	When does data prep at the CPU become a bottleneck for DNN training?	§4.3.3
Generality	Do fetch and prep stalls exist in other training platforms like TensorFlow?	§4.3.4

4.3.1 ***When dataset resides on remote storage.*** Datasets used for training DNNs could reside locally on the persistent storage of a server, or on shared remote storage such as distributed file systems (HDFS, GlusterFS - GFS), or object stores (S3, Azure blobs). We analyze the impact of two kinds of remote backends; a distributed file system, GlusterFS (GFS) and the Azure blob object store accessed via blobfuse. When data resides remotely, the first epoch of training fetches data over the network and stores it locally for subsequent use. Cluster file systems like GFS use the OS Page Cache to speed up subsequent accesses. Blobfuse downloads the dataset on to local SSD, and mimics local training from the second epoch. Figure 2a compares the epoch time for ResNet18 on `Config-SSD-V100` using GFS, blobfuse, and local SSD for the first epoch and a stable-state epoch with warmed up cache.

The data stall overhead of BlobFuse is especially high in the first epoch when it downloads the entire dataset to local storage, and can result in 20× higher training time as compared to GFS. Unsurprisingly, during the steady state epochs, data stall overheads when using the local SSD and BlobFuse are similar (as the blob data is cached on the local SSD); GFS results in more data stalls as it validates metadata of cached data items over the network every

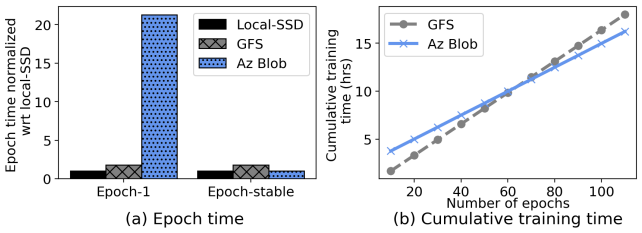


Figure 2: Training with remote stores. The high download cost of blob is amortized over training for a large # of epochs

time a data item is accessed. Blobfuse does not incur any network cost beyond first epoch, if the dataset fits on local SSD.

As shown in Figure 2b, for the ImageNet1K dataset, for BlobFuse, the cost of downloading the entire dataset in the first epoch is amortized as we train for a longer number of epochs, making the remote Blobstore a better fit compared to GFS when models are trained to accuracy for over 60 epochs.

Athough datasets are growing in size, large datasets that are publicly available fit entirely on local storage (but not in memory) [10, 23, 25, 35, 59, 78]. Therefore, a common training scenario is to pay a one-time download cost for the dataset, and reap benefits of local-SSD accesses thereafter (default and recommended mode in the Microsoft clusters). Therefore, in the rest of the work, we analyze fetch stalls in scenarios where dataset is present locally on a server, but is not entirely cached in memory.

4.3.2 ***When datasets cannot be fully cached.*** ML-optimized cloud servers with 500GB DRAM can only cache 35% of ImageNet-22K, or 45% of the FMA dataset, or 65% of the OpenImages dataset, although they entirely fit on local storage. Popular datasets like ImageNet-1K cannot be fully cached on commonly used cloud SKUs like AWS `p3.2xlarge`, which has 61 GiB DRAM. When datasets don't fit in memory, and the fetch rate(F) < compute rate ($min(P, G)$), fetch stalls occur.

Fetch stalls are common if the dataset is not fully cached in memory. Figure 3 shows the percentage of per epoch time spent on I/O for nine different DNNs when 35% (for *e.g.*, ImageNet-22k on 500GB server) of their respective datasets can be cached in memory on `Config-SSD-V100`. DNNs spend 10 –70% of their epoch time on blocking I/O, despite pipelining and prefetching, simply because the compute rate is higher than fetch rate.

OS Page Cache is inefficient for DNN training. DNN training platforms like PyTorch, TensorFlow and libraries like DALI, rely on the operating system's Page Cache to cache raw training data in memory. Unfortunately, the OS Page Cache leads to thrashing as it is not efficient for DNN training. *If 35% of the data can be cached, then an effective cache should provide 35% hits; instead, the Page Cache provides a lower hit rate.* For a 146 GiB data set, each epoch should see only 65% of the dataset, or 95GiB, fetched from storage. Instead, we observe 85% of the dataset fetched from storage every epoch; the 20% difference is due to thrashing. Figure 4 shows the fetch stalls, including those due to thrashing, when using PyTorch with DALI. An effective cache for DNN training must eliminate thrashing to reduce fetch stalls to the minimum shown in Figure 4.

Lack of coordination among caches leads to redundant I/O in distributed training. In distributed training jobs, the data to

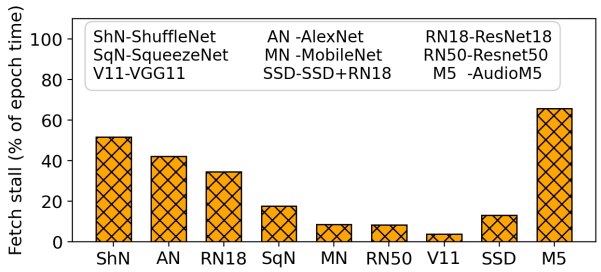


Figure 3: Fetch stalls. Several DNNs experience significant stalls waiting for I/O, when training on `Config-SSD-V100` with 35% of their dataset cached.

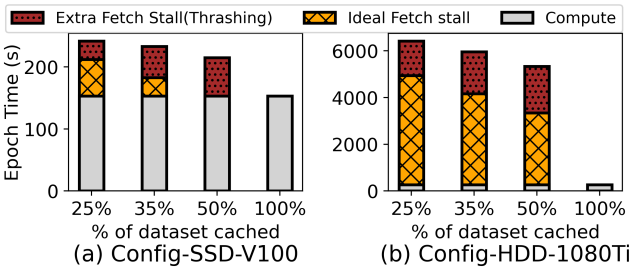


Figure 4: ResNet18 with varying cache. This stacked bar chart splits epoch time into time spent in compute, ideal fetch stalls, and the additional fetch stall due to thrashing.

be fetched and processed is divided randomly among servers, and changes every epoch. As a result, each server often has to fetch data from storage every epoch; this is done even if the required data item is cached in an another server that is a part of the distributed training job. This lack of coordination among caches makes distributed training storage I/O-bound. When training Resnet50 on ImageNet-1K (146GiB) across two servers having a total cache size of 150GiB, each server fetches 45GiB from storage in each epoch (despite the fact that the other server might have this data item in its cache). On `Config-HDD-1080Ti`, this leaves ResNet50 stalled on I/O for 75% of its epoch time.

Lack of coordination in HP search results in redundant I/O. HP search is performed by launching several parallel jobs with different HP on all available GPUs in a server [61]. All HP jobs access the same dataset in a random order in each epoch, resulting in cache thrashing and read amplification. When 8 single-GPU jobs are run in a server (35% cache), there is 7× read amplification per epoch (884 GiB read off storage compared to 125 GiB for one job), which slows down HP search on ResNet18 by 2× on `Config-SSD-V100`.

4.3.3 ***When datasets fit in memory.*** We now analyze the impact of CPU pre-processing on DNN training in the scenario where the entire dataset is cached in memory of a single server, thus eliminating fetch stalls due to storage I/O.

DNNs need 3–24 CPU cores per GPU for pre-processing. Figure 5 shows how DNN training throughput changes as we vary the number of CPU pre-processing threads (per V100 GPU) for four models. For computationally complex models like ResNet50, 3 – 4 CPU cores per GPU is enough to prevent prep stalls; for computationally lighter models like ResNet18 or AlexNet, as many as 12 –

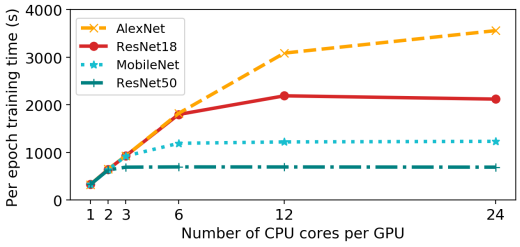


Figure 5: Impact of CPU cores on training

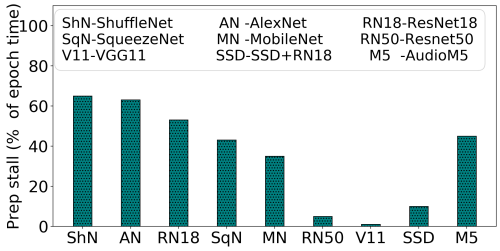


Figure 6: Prep stall across DNNs

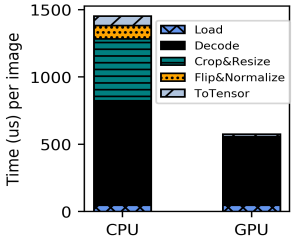
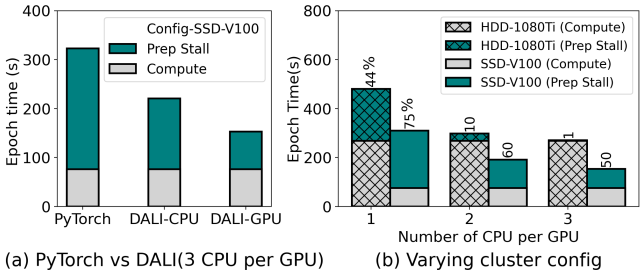


Figure 7: Breakdown prep



(a) PyTorch vs DALI(3 CPU per GPU) (b) Varying cluster config

Figure 8: 8-GPU ResNet18 training. Even with DALI, faster GPUs like V100 have upto 50% prep stalls.

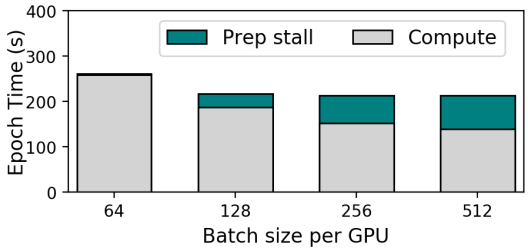
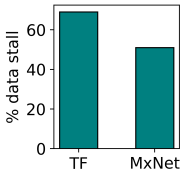


Figure 9: Impact of batch size on prep.



(a) 100% Cache

% cached (Sz:146GB)	8-GPU job Cache Miss	8-job HP I/O (GB)	8-job HP Read amp
50%	91%	860	6.14×
35%	94%	1010	7.21×
25%	97%	1019	7.28×

(b) Varying cache (TensorFlow)

Figure 10: Data stalls across frameworks

24 CPUs per GPU are needed to mask prep stalls. Even on NVIDIA's AI-optimized DGX-2, there are only three CPU cores per GPU; thus, many models have prep stalls on the DGX-2 (§7.6)

DALI is able to reduce, but not eliminate prep stalls. DALI has a GPU-based prep mode to offload a part of pre-processing to the GPU. As shown in Figure 8 (a), when all pre-processing except decoding is offloaded to the GPU for training ResNet18, prep stalls reduce. The effectiveness of DALI depends on the GPU speed; for example, on the slower 1080Ti, DALI is able to eliminate prep stalls using three CPU threads per GPU. On the faster V100 though, DALI still results in 50% prep stalls when using three CPU cores per GPU, and the GPU for pre-processing. Since DALI cannot automatically split pre-processing between CPU and GPU, we empirically find that offloading all operations except decoding to the GPUs offered best performance for the object detection and image classification models except ResNet50 and VGG11. These two models are computationally complex and slow down if pre-processing is performed at the GPU. Furthermore, DALI does not support audio decoding at the GPU. Therefore, pre-processing is entirely done at the CPU for ResNet50, VGG11 and audio M5 models. Figure 6 shows that prep stalls exist across different DNNs when training with eight GPUs each with 3 CPUs when using the best of CPU or GPU-based prep mode of DALI as described above.

Decoding is very expensive! To understand what makes prep expensive, we run a microbenchmark to breakdown the cost of different stages in a typical prep pipeline for image classification tasks. An input image is first loaded into memory, decoded, and then randomly transformed (crop, flip), and finally copied over to the GPU as a tensor that can be processed. Fig 7 shows the time taken for each operation when prep is done on CPU and GPU by DALI. We see that offloading prep to the GPU provides significant speedup at the expense of GPU memory usage (+5GB!). Second, a majority of time during prep is spent in decoding images.

Impact of batch size. The impact of batch size on GPU computational efficiency is well studied [47, 89]; larger batch sizes utilize the massive GPU parallelism better, and also reduce the number of weight updates (inter-GPU communication) per epoch, resulting in faster training. Figure 9 shows the impact of varying the batch size on epoch time and the percentage of epoch time spent on prep stalls for MobileNetv2. As computational efficiency increases with larger batches, training becomes CPU bound due to data prep. Note that, although the required GPU compute time dropped with a larger batch size, per epoch time remained same due to prep stalls. This graph makes an important point; as compute gets faster (either due to large batch sizes, or the GPU getting faster), data stalls squander the benefits due to fast compute.

Redundant pre-processing in HP search results in high prep stalls. During HP search, concurrent jobs process the same data. Currently, there is no coordination; if there are 8 HP jobs, the same data item is processed eight times. This is made worse by the fact that all HP jobs share the same set of CPU threads, leading to fewer CPU threads per GPU, and higher prep stalls. When 8 single-GPU ResNet18 HP jobs run on `Config-SSD-V100`, each job gets 3 CPU for prep and incurs a 50% prep stall as shown in Figure 6. Coordinating these HP search jobs on a single server can potentially eliminate prep stalls, as all available CPU (24 cores) can be used to prep the dataset exactly once per epoch and reused across jobs (Figure 5 shows ResNet18 requires 12 CPUs per GPU to eliminate prep stalls).

4.3.4 ***Data stalls exist across training frameworks.*** To generalize our findings on data stalls across different training platforms and data formats, we analyze the prep and fetch stalls in TensorFlow (TF) using the binary TFRecord format, and MxNet using RecordIO format. Unlike PyTorch, TF does not store training data as small individual raw files. Instead, it shuffles the small random files, serializes it, and stores them as a set of files (100-200MB each) called TFRecords. TFRecords make reads more sequential. MXNet also use a similar serializing technique for data called RecordIO [67].

Figure 10a shows that both native TF and MxNet spend 65% and 50% of the epoch time on prep stall for a 8-GPU ResNet18 training job when the dataset is entirely cached in memory. Next, Table 10b shows the percentage of misses in the Page Cache for a 8-GPU training job and the I/O amplification due to lack of coordination in HP search for varying cache sizes in TF. TFRecord format results in 40% higher cache misses than the ideal because, the sequential access nature of TFRecords (and RecordIO) is at odds with LRU cache replacement policy of the Page Cache, resulting in a pathological case for LRU (this is 20%higher than PyTorch). The lack of co-ordination in HP jobs results in upto 7.2× read amplification; although all jobs read the same 140 GiB dataset, the total disk I/O was 1.1 TB.

4.3.5 ***Analysis of NLP Models.*** In addition to vision and audio models in table 2, we evaluated data stalls on two language models; Bert-Large pretraining [37] on Wikipedia & BookCorpus dataset [96] for language modeling and GNMT [90] on WMT16 [24] (EN-De) dataset for translation, which do not exhibit data stalls. Language models have a distinct pre-processing regime compared to vision-based models. Text-based models pre-process and shuffle the data corpus *once* before training, and then reuse it every epoch. There is no online, random, heavy-weight pre-processing performed in every epoch unlike vision models. Therefore, the language models do not exhibit data stalls in our training environment. However, data stalls may show up in these models if GPUs get faster or the computation requirements for these models gets lower due to compact model representations.

5 DS-ANALYZER: PREDICTIVE ANALYSIS

While all the experiments in §4.3 are run on physical servers, we extend DS-Analyzer to help a user simulate these experiments without having to run all different configurations on physical servers. While there exists prior work that predict the performance of a DNN, they focus on profiling the layer-wise performance of DNN [5, 18], low level perf counters for accelerators [17, 47], or finding optimization opportunities at the neural network layer level [94]. In contrast, DS-Analyzer analyzes the implication of CPU, memory, and storage on the performance of a DNN and answers what-if questions.

This is a powerful means of analyzing whether throwing more hardware at the problem will solve the issue of data stalls. For instance, if training is dominated by fetch stalls (bottlenecked on disk bandwidth), then increasing the number of CPU cores on the machine has no benefit; either DRAM capacity has to be increased, or the disk must be replaced with a higher bandwidth one. Similarly, if the training job is bottlenecked on prep, then increasing DRAM has no effect on training time. DS-Analyzer is useful in scenarios like this, to predict the performance of a model as we scale up CPU, memory, or storage.

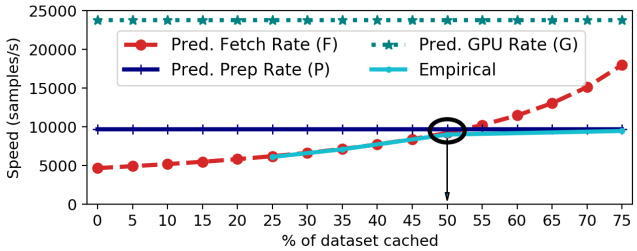


Figure 11: Estimating optimal cache size with DS-Analyzer.

5.1 Estimating data stalls

Consider the different components involved in a typical DNN data pipeline as in Figure 17; data is fetched from cache (and store) with an effective prefetch rate F, pre-processed at the CPU at a rate P and processed at the GPU at a rate G. To perform predictive analysis, DS-Analyzer measures several metrics related to the data pipeline of the model; the maximum ingestion rate at the GPU (G), the rate of CPU prep (P), the rate of cache fetch (C), and the rate of storage fetch (S). Using these metrics, DS-Analyzer models the training iteration to answer what-if questions such as, how much DRAM cache is required for this model to eliminate fetch stalls?; DS-Analyzer collects these metrics for a model as follows.

(i) Measure ingestion rate (G). To find the maximum possible speed at which the DNN can train, DS-Analyzer first runs the job script for a fixed number iterations (default:100) with synthetic data that is pre-populated at the GPUs. It then calculates G as,

$$G = \frac{\text{Total samples processed in (i)}}{\text{Time for (i)}} \tag{1}$$

$$\text{Samples processed} = \#\text{iterations} \times \text{global batch size} \tag{2}$$

(ii) Measure prep rate (P). Next, DS-Analyzer executes the training script with the given dataset by ensuring that the subset of data used is cached in memory, using all available CPU cores. Additionally, the GPU computation is disabled to only run the data loader. This is required because, if $P \geq G$, then we cannot measure P using the knowledge of runs (i) and (ii), as prep will be pipelined with GPU compute. Therefore, DS-Analyzer disables GPU computation and estimates P in the same way as Eq (1).

(iii) Measure storage fetch rate (S). Storage fetch rate is the maximum random read throughput of the storage device. To measure this, DS-Analyzer runs the data loader (with a cold cache, disabling both pre-processing and GPU compute), with all CPU cores.

(iv) Measure cache fetch rate (C). To measure the rate at which data can be fetched from cache, DS-Analyzer uses a microbenchmark to measure memory bandwidth and uses it as an approximation for C. Note that run (ii) actually includes the time to fetch cached items as well; however we see that the cache fetch rate is very high (few tens of GBps), and does not add noise to the measurement of prep rate.

5.2 Example : Predicting optimal cache size

We now describe an example of what-if analysis with DS-Analyzer. We show how DS-Analyzer answers the question : *how much DRAM cache does the DNN need to eliminate fetch stalls?*

To predict the implication of cache size, DS-Analyzer calculates the effective prefectch rate (F) for a given cache size (x % of the dataset). Here, we assume that the cache implements an efficient policy like MinIO; i.e., a cache of size x items has atleast x hits per epoch.

F is computed as follows. Say the size of the dataset is D samples, and cache is x% of the dataset. Therefore, in an epoch, the total time to read the dataset is given by

$$T_f = \frac{D \times x}{C} + \frac{D \times (1-x)}{S} \tag{3}$$

The fetch rate is then calculated as,

$$F = \frac{D}{T_f} = \frac{D}{\frac{D \times x}{C} + \frac{D \times (1-x)}{S}} \tag{4}$$

Since $C >> S$, $F \propto \frac{1}{1-x}$, i.e, the effective fetch rate increases, as the number of uncached items per epoch decreases. Since DS-Analyzer has already estimated values of D, C, and S, given a cache percentage x, DS-Analyzer can predict the fetch rate using Eq (4).

To evaluate how accurately DS-Analyzer can answer this question, we run the actual experiment by varying cache size on a physical server (empirical), and comparing it to the predictions of DS-Analyzer for AlexNet on `Config-SSD-V100` with Imagenet-1K. Figure 11 plots the predicted values of F, P, and G, alongside empirical speed while varying cache size. First, we observe that the predicted training speed ($min(F, P, G)$) is a maximum of 4% off the empirical results. Second, using these predictions, DS-Analyzer can estimate the optimal cache size for the model by comparing it with prep rate (P) and GPU ingestion rate (G). To eliminate fetch stalls, $F > min(P, G)$ as shown by the intersection in Figure 11. At lower cache sizes, training is I/O bound, however, a cache that is 50% of the dataset size is sufficient to eliminate fetch stalls; larger cache (more DRAM) is not beneficial beyond this point, as training becomes CPU-bound. A comprehensive list of data pipeline rates (G, P, F) for several models, datasets, and configurations is in the Appendix (§A.1).

6 MITIGATING DATA STALLS

Based on the insights from our analysis, we explore ways of mitigating data stalls using domain specific techniques that reduces cache misses, and eliminates redundancy in data fetch and prep. We further discuss how to reduce the cost of decoding in future work(§8).

Technique	*Impact*	*Benefits*
MinIO Cache	DNN-aware caching to minimize IO by reducing cache misses per epoch (§6.1)	Single-server training
Partitioned MinIO Cache	Eliminate redundant fetch by coordinating remote MinIO caches (§6.2)	Distributed training
Coordinated Prep	Eliminates redundant fetch and prep across jobs (§6.3)	Single-server training

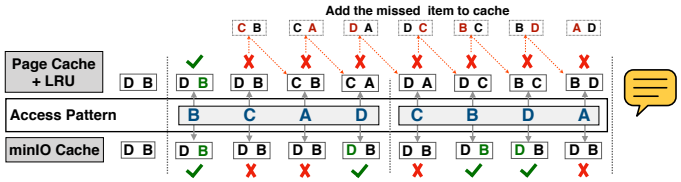


Figure 12: Cache hits with MinIO. Cache activity for two "epochs" of training for page cache and MinIO.

6.1 The MinIO cache

As datasets increase in size, they cannot be cached entirely in the memory of a server during training. In such cases, DNNs suffer from fetch stalls if the rate of data fetch is lower than the rate of compute (despite prefetching and pipelining data fetch with compute) as discussed in (§4.3). Note that, when fetch stalls occur, training is bottlenecked by the bandwidth of storage device, therefore it is crucial to minimize I/O by maximizing cache hits every epoch.

DNN training frameworks today, rely on the OS Page Cache to cache the training dataset. However, we tap on the piercing insight of Stonebraker *et al.*'s pioneering work on database caching [84], that *the abstractions provided by OS can hinder the development of efficient databases*, and validate it in the context of DNN workloads. Therefore, we study the DNN data access pattern to design a domain-specific cache MinIO.

OS Page Cache works as follows; whenever a data item is read from storage, it is cached in the Page Cache to speed up future accesses. When the Page Cache reaches its capacity, a *cache replacement policy* decides which of the existing items to evict to make space for the new one. Linux uses a variant of Least Recently Used (LRU) for cache replacement [39].

However, DNN training has a unique data access pattern: it is *repetitive across epochs* and *random within an epoch*. Training is split into epochs: each epoch accesses all the data items in the dataset exactly once in a random order. We make a key observation about the DNN access pattern that is at odds with such OS cache replacement policies. *All* data items in the dataset have equal probability of access in an epoch. Therefore, it is *not important* which data item is cached. Instead, it is crucial that cached items are not replaced *before* they are used, to minimize storage I/O per epoch.

Therefore, MinIO recommends a simple and unintuitive solution; *items, once cached, are never replaced* in the DNN cache. MinIO works as follows. In the first epoch of the training job, MinIO caches random data items as they are fetched from storage, to populate the cache. Once the cache capacity is reached, MinIO will not evict any items in the cache; instead, the requests to other data items default to storage accesses. The items in the MinIO cache survive across epochs until the end of the training job. Every epoch beyond the first gets exactly as many hits as the number of items in the cache; this reduces the per-epoch disk I/O to the difference in the size of dataset and the cache.

Figure 12 contrasts the caching policy of the OS Page Cache and MinIO. Consider a dataset of size 4 (with items A – D) and a cache of size 2 (50% cache). Let's say after warmup, the cache has two items D and B. Figure 12 shows the state of the cache for two training epochs. MinIO only incurs capacity misses per epoch (here 2); the Page Cache on the other hand, can result in anywhere between 2-4

misses per epoch because of thrashing. For instance, in the first epoch, D is in the cache to begin with, but kicked out to make way for a new item C, and later in the same epoch it is requested again (thrashing). We empirically verified this using large datasets and varying cache sizes (§7) and found that Page Cache results in close to 20% more misses than MinIO due to thrashing.

MinIO's no replacement policy simplifies the design of the cache as we do not need bookkeeping about the time or frequency of access of data items. Moreover, we choose to implement MinIO in user-space and not as a new replacement policy in the kernel, making it flexible to use in scenarios where the user has no root privileges to modify the kernel. The strength of MinIO thus lies in its simplicity and effectiveness.

6.2 Partitioned MinIO Caching

MinIO reduces the amount of disk I/O (fetch stalls) in single-server training. In distributed training, the dataset is partitioned and processed by a group of servers. Each server operates on a random shard of the dataset per epoch, and this partition changes every epoch (§2.1). The MinIO cache alone, is not efficient in this setting. For example, consider a distributed training job across two servers, each of which can cache 50% of the dataset. In every epoch, each server has to process a random 50% partition of the dataset, some of which may be hits in the local MinIO cache but the misses result in storage I/O, which is expensive and results in fetch stalls.

We observe that the cross-node network bandwidth in publicly available cloud GPU instances and our clusters(10-40 Gbps) is upto 4× higher than the read bandwidth of local SATA SSDs (530 MBps). Data transfer over commodity TCP stack is much faster than fetching a data item from its local storage, on a cache miss. Therefore, we can coordinate the remote MinIO caches across all servers.

Partitioned MinIO caching works as follows. In the first epoch, the dataset is sharded across all servers, and each server populates it's local MinIO cache with data items in the shard assigned to it. At the end of the first epoch, we collectively cache a part of the dataset of size equal to the sum of capacities of individual MinIO caches. To route data fetch requests to the appropriate server, we maintain metadata about data items present in each server's cache. Whenever a local cache miss happens in the subsequent epoch at any server, the item is first looked up in the metadata; if present, it is fetched from the respective server over TCP, else from its local storage. If the aggregate memory on the participating servers is large enough to cache the entire dataset, then partitioned caching ensures that there is no storage I/O on any server beyond the first epoch; the entire dataset is fetched exactly once from disk in the duration of distributed training.

6.3 Coordinated Prep

Hyperparameter (HP) search for a model involves running several concurrent training jobs, each with a different value for the HP and picking the best performing one. Our analysis shows that co-locating HP search jobs on the same server results in both fetch and prep stalls (§4) due to lack of coordination in data fetch and prep among these jobs.

We introduce *coordinated prep* to address this issue. The idea behing coordinated prep is simple. Each job in the HP search operates on the same data; hence, instead of accessing data independently for each job, they can be coordinated to fetch and prep the dataset exactly once per epoch. Each epoch is completed in a synchronized fashion by all HP jobs; as a result, pre-processed minibatches created by one job can be reused by all concurrent jobs.

Coordinating HP search jobs must be done carefully to ensure that: *each job processes the entire dataset exactly once per epoch.* A naive way of doing this is to pre-process the dataset once and reuse across all HP jobs and all epochs as suggested by prior work [55, 68]. This approach will not work for two reasons. First, reusing pre-processed data across epochs may result in lower accuracy, as the random transformations are crucial for learning. Second, the pre-processed items are 5–7× larger in size when compared to the raw data items. Caching pre-processed items will overflow the system memory capacity quickly for large datasets. If we store them on storage, we may incur fetch stalls.

Coordinated prep addresses these challenges by staging pre-processed minibatches in memory for a short duration *within an epoch.* Since each job has identical per-minibatch processing time, the minibatch is short lived in the staging area. Coordinated prep works as follows.

Each HP search job on a server receives a random shard of the dataset when they start. Each job fetches and pre-processes the assigned shard, creating minibatches as they would normally do. When ready, these minibatches are exposed to the other jobs in the cross-job staging area. This is a memory region that is accessible to all running jobs on the server. Additionally, each minibatch has a unique ID and an associated atomic counter that tracks how many jobs have used this minibatch so far in the current epoch. When a job needs a minibatch for GPU processing, it retrieves it from the staging area and updates its usage counter. A minibatch is deleted from the staging area when it is used exactly once by all running jobs, as we want to ensure that it is not used across epochs. We empirically show in §7 that the addition of cross-job staging area does not introduce additional memory overhead.

Thus, coordinated prep ensures one sweep over the dataset per epoch for both data fetch and pre-processing, eliminating redundant work. Note that coordinated prep allows addition or removal of jobs only at epoch boundaries; this is not an issue because popular HP search algorithms evaluate the objective function (for e.g., accuracy), and make decisions on terminating or continuing the job at epoch boundaries [48, 60]

To handle job failures in HP search, we implement a failure detection module to monitor the status of running jobs. Every prepared minibatch fetched from the staging area has an associated timeout. If any job times out waiting for a minibatch, it notifies the driver process of a possible failure. All the jobs can deterministically identify which job failed to populate the batch it is waiting on. If the job indeed failed, a new process is spawned to resume data loading for the failed shard.

6.4 Tying it all together with CoorDL

We implement the three techniques discussed thus far as a part of a user-space data loading library, CoorDL. We build CoorDL on top of DALI to take advantage of the GPU-accelerated data pre-processing

operations. CoorDL can be used as a drop-in replacement for the default PyTorch dataloader.

The overall architecture of CoorDL is as follows. The training dataset resides on a local storage device like SSD/HDD. If the data resides on a remote storage service, it is cached in local storage when first accessed [3]. For all later epochs, the data is fetched from local storage. In each training iteration, a minibatch of data must be fetched from disk (or cache), pre-processed to apply random transformations and collated to a tensor that can be copied over to the GPU for computation. CoorDL manages its own MinIO cache of the raw data items (before any stochastic pre-processing transformations are applied). The data sampling and randomization is unmodified; in each epoch, every minibatch is sampled randomly from the dataset. Every data item is then subjected to the random pre-processing pipeline specified in the training workload. The prepared minibatch is then placed in a cross-job staging area for consumption by the GPU. If a single data-parallel job is running across multiple GPUs in a server, then the minibatches in the staging are used exactly once per epoch and discarded; if there are concurrent HP jobs on a server, then the staging area retains minibatches until each concurrent job has used it exactly once in the current epoch. Any minibatch that satisfies this criteria is evicted from the staging area to make way for newer batches.

7 EVALUATION

We now evaluate the efficacy of CoorDL on three different aspects of the training process: multi-GPU training on a single server, distributed training across multiple servers, and hyperparameter tuning. We evaluate our techniques on **nine** models, performing **three** different ML tasks (image classification, object detection and audio classification) on four different datasets, each over 500GB as shown in Table 2. Since DALI strictly outperforms PyTorch DL, we use DALI as the baseline in our experiments. For each model, we run both CPU-based (all pre-processing on CPU) and GPU-based (part of decoding and all other transformations on GPU) mode of DALI, and present the best of the two results.

Experimental setup. We evaluate CoorDL on two representative server configurations from Microsoft clusters (Tbl 3) each with 500 GiB DRAM, 24 CPU cores, 40 Gbps Ethernet, eight GPUs, and 1.8 TiB of storage space. `Config-SSD-V100` uses V100 GPUs and a SATA SSD, while `Config-HDD-1080Ti` uses 1080Ti GPUs and a magnetic hard drive. We use the same training methodology we used for analysis (§4.1). We seek to answer the following questions:

How does the MinIO cache affect multi-GPU training on a server?	§7.1
How does partitioned caching improve training time for jobs distributed across multiple servers?	§7.2
How does coordinated prep benefit HP search?	§7.3
Does CoorDL affect DNN training accuracy?	§7.4
Does CoorDL enable better resource utilization compared to DALI?	§7.5
Does CoorDL accelerate training on ML servers like the DGX-2?	§7.6

7.1 Single-server Multi-GPU training

CoorDL speeds up a single-server training job by reducing cache misses using the MinIO cache. Figure 13 (a) plots the relative speedup with respect to DALI while training the image classification and object detection models on the OpenImages dataset, and audio classification on FMA dataset. We evaluate MinIO against two modes of DALI. DALI's default mode is *DALI-seq*, where it reads data sequentially off storage and shuffles them in memory [6]. *DALI-shuffle* accesses the dataset in a randomized order (doing random reads, similar to the native dataloader of PyTorch).

MinIO results in upto 1.8× higher training speed compared to DALI-seq by eliminating thrashing on `Config-SSD-V100`. When the image classification models are trained with ImageNet-22k dataset, CoorDL results in up to 1.5× speedup. On `Config-HDD-1080Ti`, CoorDL accelerates ResNet50 training on OpenImages by 2.1× compared to DALI-seq and 1.53× compared to DALI-shuffle respectively.

Reduction in cache misses. We measure the disk I/O and number of cache misses when training ShuffleNet on OpenImages dataset on `Config-SSD-V100`. This server can cache 65% of the dataset. CoorDL reduces misses to the minimum number of 35%, resulting in 225 GB of I/O. In contrast, DALI-Seq results in 66% cache misses, increasing I/O by 87% to 422 GB; DALI-shuffle results in 53% cache misses, increasing I/O by 50% compared to CoorDL to 340 GB.

Note that, when the whole dataset does not fit in memory, DALI-shuffle performs better than DALI-seq (because sequential access is a pathological case for the Linux LRU page cache). Therefore, our evaluation in the rest of this section compares CoorDL to the stronger baseline, DALI-shuffle.

7.2 Multi-Server Distributed Training

We now evaluate CoorDL on a distributed training scenario. The lack of cache co-ordination between the participating servers results in fetch misses that lead to disk I/O. CoorDL uses partitioned caching to avoid redundant I/O.

Figure 13(b) shows that CoorDL improves the throughput of distributed training jobs by upto 15× (AlexNet on OpenImages) when trained across two `Config-HDD-1080Ti` servers (16 GPUs). On `Config-HDD-1080Ti` servers, 65% of the OpenImages dataset can be cached on a single server; and it can be fully cached in the aggregated memory of two servers. Therefore, CoorDL moves the training job from being I/O bound to GPU bound.

When trained across two servers on `Config-SSD-V100`, CoorDL accelerates ShuffleNet on ImageNet-22k by 1.3×, and Audio-M5 on FMA by 2.9×. The relative gains are lower on `Config-SSD-V100` because the cost of a fetch miss is lower on SSDs due to its high random read throughput, as compared to HDDs on `Config-HDD-1080Ti`.

7.3 Hyperparameter Search

Figure 13 (d) plots the relative increase in throughput of individual jobs across several models when eight concurrent HP search jobs are run on a `Config-SSD-V100` server. On less computationally complex models like AlexNet and ShuffleNet, CoorDL increases training speed by 3×, because these models are originally CPU bound due to prep. For the audio model, CoorDL increases the training speed by 5.6×. CoorDL reduced the total disk I/O from 3.5TB to 550GB. Similarly, on `Config-HDD-1080Ti`, CoorDL results in 5.3× faster training on the audio model, and 4.5× faster training on ResNet50. On `Config-HDD-1080Ti`, CoorDL results in 5.3× faster training on the audio model, and 4.5× faster training on ResNet50 by coordinating data fetch and prep.

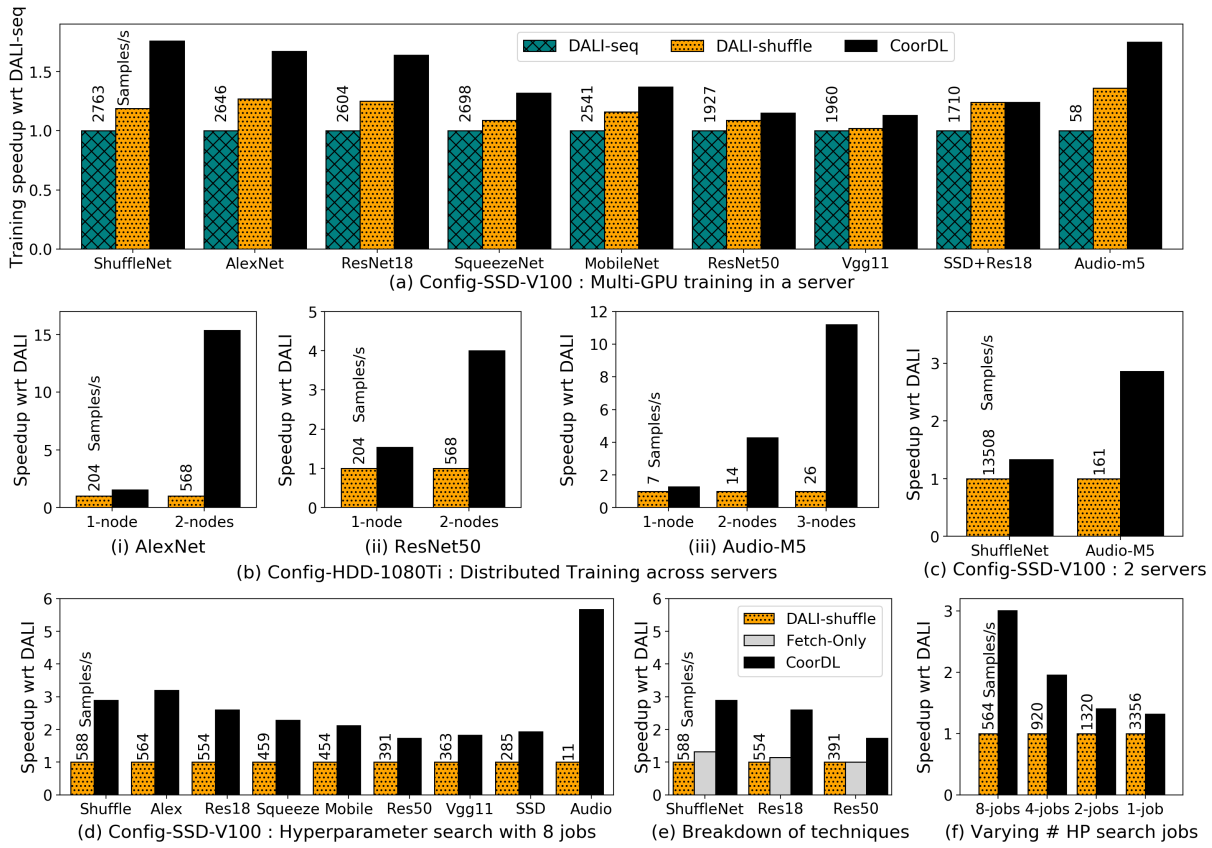


Figure 13: This graph compares DALI against CoorDL for a variety of training scenarios; single server, multi-server and HP search, across 2 clusters and 9 models. CoorDL significantly accelerates training by eliminating redundant data fetch and prep.

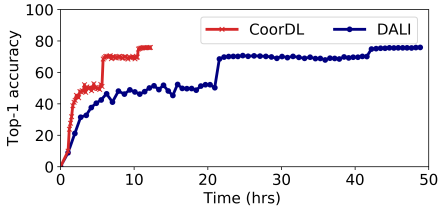


Figure 14: Training to accuracy

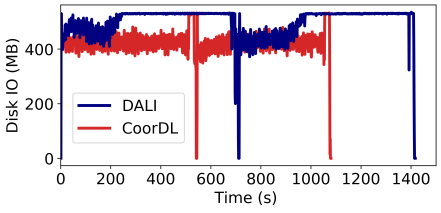


Figure 15: Disk I/O profile

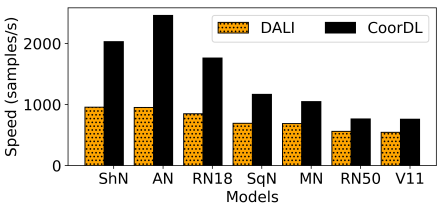


Figure 16: Evaluation on DGX-2

Split of coordinated prep benefits. Next, we show the breakdown of speedup due to coordination of data fetch and prep during HP search. When fetch is coordinated, concurrent jobs use data fetched by other jobs; but each job performs its own data prep. CoorDL coordinates both; eliminating redundant fetch and prep. Figure 13 (e) plots the results on `Config-SSD-V100`. In this case, data stall is dominated by prep, which CoorDL mitigates unlike prior work like Quiver [58] that only coordinates fetch.

Multi-GPU HP search jobs. Figure 13 (f) shows training with different configs of HP search jobs on a machine; 8 1-GPU jobs, 4 2-GPU jobs, 2 4-GPU jobs, or 1 8-GPU job for AlexNet on OpenImages. For a single job case, the benefit is due to the MinIO cache; in other configs, it is due to coordinated prep. More the number of concurrent jobs, higher the benefit with coordination.

7.4 Training to Accuracy with CoorDL

CoorDL does not change the randomness of data pre-processing involved. Its techniques do not affect the learning algorithm. To demonstrate this, we train ResNet50 to accuracy on ImageNet-1K using 16 GPUs across two `Config-HDD-1080Ti` servers, where each server is capable of caching 50% of the dataset. Figure 14 shows that CoorDL reduces the time to target accuracy (75.9%) from two days to just 12 hours (4× better), due to partitioned caching.

7.5 Resource Utilization

MinIO results in lower disk I/O and better CPU utilization. Figure 15 shows the I/O for two epochs of training ResNet18 on OpenImages on `Config-SSD-V100`. The I/O behavior is similar across models and server configurations. DALI observes cache hits

at the beginning of the epoch, but soon becomes I/O bound (disk bandwidth: 530 MB/s). Since MinIO is caching a random subset of the dataset, cache hits are uniformly distributed across the epoch in CoorDL. This results in a predictable I/O access pattern and faster training (epochs end earlier in Figure 15). Profiling the CPU during training shows that the pre-processing threads in DALI are often stalled waiting for I/O. Since MinIO reduces the total disk I/O, CoorDL is able to better utilize the CPU for pre-processing. The combination of lower disk I/O and better CPU utilization leads to shorter training times when using CoorDL.

CoorDL uses a fraction of available network bandwidth. CoorDL shards the dataset equally among all servers in distributed training to ensure load balancing. We track the network activity during the distributed training for ResNet50 on OpenImages across two, three, and four servers with DALI and CoorDL. CoorDL used 5.7 Gbps per server of network bandwidth (14% of the 40 Gbps available). DALI used 1.18 Gbps of network bandwidth per server. CoorDL used 4.8× higher network bandwidth to train 4.3× faster.

Co-ordinated prep has low memory overhead. By design, co-ordinated prep has the same memory requirements as DALI. To experimentally validate this, we track the memory utilization of running hyperparameter search on AlexNet on OpenImages on a `Config-SSD-V100` server using eight concurrent jobs. CoorDL uses 5 GB of extra process memory to store prepared mini-batches in memory until all hyperparameter jobs consume it. We reduce the cache space given to CoorDL by 5 GB (keeping the total memory consumption same for CoorDL and DALI). Despite the lower cache space, CoorDL still accelerated training by 2.9×.

7.6 CoorDL on DGX-2

We now compare CoorDL against DALI on the bleeding-edge ML optimized server DGX-2 while performing HP search across 16 GPUs using OpenImages dataset. Since this dataset can be fully cached in the memory of DGX-2 (1.5TB DRAM), we observe no stalls due to data fetch beyond the first epoch. However, the imbalance in the ratio of CPU-GPU results in prep stalls which CoorDL mitigates by coordinating pre-processing. CoorDL accelerates HP search by 1.5×– 2.5× over DALI by eliminating redundant data prep, enabling efficient usage of CPU to mask prep stalls.

8 DISCUSSION

Our analysis of data stalls reveals several key insights to utilize the computational capabilities of GPUs by minimizing data stalls. While we explore ways of mitigating data stalls in a user-space library CoorDL, we believe there is more to be done.

Decoded cache to reduce pre-processing overhead. CoorDL does not address the high cost of decoding/decompressing raw images, which our analysis identifies to be the most expensive operation during data prep. A future direction is to evaluate the benefits of caching decoded data items instead of the current approach of caching raw encoded items. Since decoding is deterministic, it is possible to cache it across epochs. However, this is non trivial; decoding increases the dataset size by $5 - 7\times$. It is an interesting future direction to enable decoded caching without incurring the high space overhead, possibly using serialized data formats.

Automatic prep offload to GPUs. Data pipelining frameworks like DALI have the ability to perform certain image and audio based pre-processing (prep) such as crop, flip, and other transformations on GPU accelerators. However, there is a memory-performance tradeoff in deciding how many steps of prep are offloaded to the GPU for two reasons. (1) Performing prep at the GPU takes up a part of the already scarce GPU memory which may result in training with lower batch sizes, thereby affecting training efficiency. (2) Prep at the GPU may interfere with the computations performed by the learning algorithm; this adversely affects the overall throughput of training for computationally expensive and deeper networks. Therefore, the split of prep operations must be carefully chosen considering the model's architecture, batch size, and data stalls. While this split is determined manually by trial-and-error today, automating it with a careful eye on GPU and CPU utilization is an interesting future direction.

Minibatch as a service. Our analysis shows that the imbalance in CPU cores per GPU in ML optimized servers result in data stalls for several models. In such cases where single-host capacities are maxed out, a viable approach is to offload data prep to other idle host machines in a cluster. This is especially useful in production clusters with high-bandwidth ethernet, where several jobs use the same dataset and similar pre-processing pipelines; a dedicated set of servers can be used to centrally pre-process minibatches of data, while the training jobs can request minibatch as a service, thereby entirely disaggregating learning from data management.

Cost-peformance tradeoff of upgrading hardware. Our analysis finds that data stalls squander away the improved performance of faster, expensive GPUs, resulting in lower value/$ spent as shown in Figure 8 (b). Therefore, it may be economical to train some models on slower, less expensive GPUs with no data stalls, rather on underutilizing the accelerator capabilities due to stalls on faster, expensive GPUs. In practice, such techniques may improve the overall efficiency in multi-tenant clusters by assigning jobs to accelerators in such a way that they maximize GPU utilization.

Data stalls in inference. This work addresses data stalls in the training pipeline which have three distinct features from inference. (1) Training requires a large volume of data samples, (2) performs a larger set of data prep for every batch, and (3) requires backpropagation during the learning phase. While inference jobs require fewer prep steps per sample or batch, it also performs lesser GPU computation compared to training. Moreover, the limited memory and compute availability at edge devices may also introduce data stalls in inference. We hope our analysis encourages similar research and possibly unique optimizations in inference land.

Trade-off between convergence rate and epoch time for other SGD variants. This work focuses on the most common case of mini-batch SGD with a random shuffling of the data in every epoch which is the default for the models we analyzed. A future direction is to understand the impact of relaxing the ETL requirements assumed in this work (such as random prep and shuffling every epoch) on epoch time and model convergence. Although relaxing these constraints may reduce data stalls and hence epoch time, it may prolong convergence, or affect the accuracy of some models. It is worth investigating this behavior theoretically and empirically.

9 RELATED WORK

To the best of our knowledge, this paper presents the first comprehensive analysis of data stalls in DNN training. We place our work in the context of prior work.

Optimizing remote storage via local caching. Quiver [58] uses local SSD caches to eliminate the impact of slow reads from remote storage. The best case for Quiver is when the dataset is completely cached on local storage; our system starts from this baseline and further improves performance. Quiver does not consider or optimize prep stalls, only handling fetch stalls.

Partitioned caching. Cerebro [68] introduces a new parallel SGD strategy for model selection tasks. It partitions the dataset across the servers in a cluster and hops the models from one server to other, instead of shuffling data. Cerebro does not improve performance for DNN training on a single server, while CoorDL optimizes performance in this scenario. Furthermore, Cerebro is designed for a specifc scenario - distributed model search; on the contrary, our analysis and CoorDL have a broader scope. DeepIO [95] uses a partitioned caching technique for distributed training with remote data, but relies on specialized hardware like RDMA. In contrast, our work shows that it is possible to mask fetch stalls using commodity TCP stacks.

Redundancy in DNN training. Prior work like Model Batching [70] addressed redundancy in model search when running multiple DNNs together on a single GPU, by sharing GPU computation across jobs. Our analysis looks at the setting where GPUs are not shared between jobs. OneAccess [55] is a preliminary study that makes a strong case for storing pre-processed data across epochs to reduce prep stalls; however such an approach precludes commonly used online data pre-processing techniques and this can affect model convergence. In contrast, CoorDL carefully eliminates redundancy while preserving accuracy and providing significant speedups.

Hardware solutions to fetch stalls. New hardware like NVIDIA's Magnum IO [13], and PureStorage's AIRI [19] provide high throughput storage solutions to address fetch stalls. While these fast hardware may mask fetch stalls in some models, they may not help if the model is bottlenecked on prep stalls. Our work mitigates data stalls with existing commodity servers as opposed to relying on expensive hardware solutions, by efficiently using available hardware.

Optimizing DNN training time. A number of solutions have been proposed to reduce the training time for DNNs including specialized hardware [4, 9, 51, 54, 72, 73, 76, 82], parallel training [30, 34, 45, 53, 56, 57, 69], GPU memory optimizations [29, 49, 77], lowering communication overhead [12, 43, 50, 63, 89, 92], , and operator optimizations [28, 52, 87]. This paper presents a new point in this spectrum, *data stalls*.

Domain specific caching. The idea of designing a caching policy that is aware of application semantics is not new. Stonebraker *et al.* highlighted the importance of domain-aware caching for databases [84]. Tomkins *et al.* show that informed prefetching and caching in file systems can reduce the execution time of I/O-intensive applications [86]. Our work draws parallels to such techniques by first understanding DNN access pattern and then devising a caching policy based on these observations.

10 CONCLUSION

We present the first detailed study of data stalls in several DNNs, and show that it accounts for up to 65% of the training time. The insights from our study, guide the design of a coordinated caching and preprocessing library, CoorDL, that can accelerate DNN training by mitigating data stalls. CoorDL accelerates training by up to 15× for distributed training across two servers, and 5.3× for HP search (on the audio model), by coordinating data fetch and prep across jobs. The techniques behind CoorDL are simple and intuitive, easing adoption into production systems.

ACKNOWLEDGEMENTS

This work was done during an internship at Microsoft Research as part of Project Fiddle. We thank all the anonymous VLDB reviewers, members of the UT SaSLab, Jorgen Thelin, Jack Kosian, Deepak Narayanan, Keshav Santhanam and many of our MSR colleagues for their invaluable feedback that made this work better. We sincerely thank MSR Labs for their generous, unwavering support in procuring the many resources required for this work.

REFERENCES

[1] 2020. AWS Instance Types. https://aws.amazon.com/ec2/instance-types/#p3.
[2] 2020. AWS Instance Types. https://aws.amazon.com/ec2/instance-types/#p2.
[3] 2020. Blobfuse. https://github.com/Azure/azure-storage-fuse.
[4] 2020. Cerebras Wafer Scale Engine. https://www.cerebras.net/.
[5] 2020. Cloud TPU Tools. https://cloud.google.com/tpu/docs/cloud-tpu-tools.
[6] 2020. DALI: Supported Operations. https://docs.nvidia.com/deeplearning/dali/user-guide/docs/supported_ops.html#nvidia.dali.ops.FileReader.
[7] 2020. EBS Volume types. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-volume-types.html.
[8] 2020. Fast AI Data Preprocessing with NVIDIA DALI. https://devblogs.nvidia.com/fast-ai-data-preprocessing-with-nvidia-dali/.
[9] 2020. GraphCore Intelligence Processing Unit. https://www.graphcore.ai/.
[10] 2020. ImageNet-22k. http://www.image-net.org/releases.
[11] 2020. Microsoft Philly Traces. https://github.com/msr-fiddle/philly-traces.
[12] 2020. NCCL. https://developer.nvidia.com/nccl.
[13] 2020. NVIDIA : Magnum-IO. https://www.nvidia.com/en-us/data-center/magnum-io/.
[14] 2020. NVIDIA DGX-2: Enterprise AI Research System. https://www.nvidia.com/en-us/data-center/dgx-2/.
[15] 2020. NVIDIA nvJPEG Library. https://docs.nvidia.com/cuda/nvjpeg/index.html.
[16] 2020. NVIDIA Object Detection. https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/Detection/SSD.
[17] 2020. NVIDIA Profiler. https://docs.nvidia.com/cuda/profiler-users-guide/index.html.
[18] 2020. Profiling MXNet models. https://mxnet.apache.org/api/python/docs/tutorials/performance/backend/profiler.html.
[19] 2020. PureStorage : AIRI. https://www.purestorage.com/products/flashblade/ai-infrastructure.html.
[20] 2020. PyTorch: DDP vs DP. https://pytorch.org/tutorials/intermediate/ddp_tutorial.html.
[21] 2020. TorchAudio classifier. https://pytorch.org/tutorials/beginner/audio_classifier_tutorial.html?highlight=audio.
[22] 2020. TorchVision models. https://pytorch.org/docs/stable/torchvision/models.html.
[23] 2020. Training a Champion: Building Deep Neural Nets for Big Data Analytics. https://www.kdnuggets.com/training-a-champion-building-deep-neural-nets-for-big-data-analytics.html/.
[24] 2020. WMT16. http://www.statmt.org/wmt16/.
[25] Sami Abu-El-Haija, Nisarg Kothari, Joonseok Lee, Paul Natsev, George Toderici, Balakrishnan Varadarajan, and Sudheendra Vijayanarasimhan. 2016. Youtube-8m:

A large-scale video classification benchmark. *arXiv preprint arXiv:1609.08675* (2016).

[26] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of machine learning research* 13, Feb (2012), 281–305.

[27] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. 2020. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*. ACM, 1:1–1:16. https://doi.org/10.1145/3342195.3387555

[28] 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 (OSDI 18)*. 578–594.

[29] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. *arXiv preprint arXiv:1604.06174* (2016).

[30] Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System.. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, Vol. 14. 571–582.

[31] Jichan Chung, Kangwook Lee, Ramtin Pedarsani, Dimitris Papailiopoulos, and Kannan Ramchandran. 2017. Ubershuffle: Communication-efficient data shuffling for sgd via coding theory. *Advances in Neural Information Processing Systems (NIPS)* (2017).

[32] Alex Clark. 2015. Pillow (PIL Fork) Documentation.

[33] Wei Dai, Chia Dai, Shuhui Qu, Juncheng Li, and Samarjit Das. 2017. Very deep convolutional neural networks for raw waveforms. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 421–425.

[34] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems*. 1223–1231.

[35] Michaël Defferrard, Kirell Benzi, Pierre Vandergheynst, and Xavier Bresson. 2016. Fma: A dataset for music analysis. *arXiv preprint arXiv:1612.01840* (2016).

[36] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[37] 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, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/v1/n19-1423

[38] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. 2017. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 1487–1495.

[39] Mel Gorman. 2020. Understanding the Linux Virtual Memory Manager. https://www.kernel.org/doc/gorman/html/understand/understand013.html.

[40] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

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

[42] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Harry Liu, and Chuanxiong Guo. 2019. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*. USENIX Association, 485–500. https://www.usenix.org/conference/nsdi19/presentation/gu

[43] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H. Campbell. 2019. TicTac: Accelerating Distributed Deep Learning with Communication Scheduling. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*. mlsys.org.

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

[45] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*. 103–112. http://papers.nips.cc/paper/8305-gpipe-efficient-training-of-giant-neural-networks-using-pipeline-parallelism

[46] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[47] Kumar Iyer and Jeffrey Kiel. 2016. GPU Debugging and Profiling with NVIDIA Parallel Nsight. In *Game Development Tools*. AK Peters/CRC Press, 303–324.

[48] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, et al. 2017. Population based training of neural networks. *arXiv preprint arXiv:1711.09846* (2017).

[49] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. 2018. Gist: Efficient Data Encoding for Deep Neural Network Training. In *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA '18)*.

[50] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. 2019. Priority-based Parameter Propagation for Distributed DNN Training. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*. mlsys.org.

[51] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 947–960.

[52] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: Optimizing Deep Learning Computation with Automated Generation of Graph Substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. ACM.

[53] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of the 2nd SysML Conference, SysML '19*. Palo Alto, CA, USA.

[54] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA 2017)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

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

[56] Alex Krizhevsky. 2014. One Weird Trick for Parallelizing Convolutional Neural Networks. *arXiv preprint arXiv:1404.5997* (2014).

[57] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*. 1097–1105.

[58] Abhishek Vijaya Kumar and Muthian Sivathanu. 2020. Quiver: An Informed Storage Cache for Deep Learning. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*. USENIX Association, 283–296.

[59] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Tom Duerig, et al. 2018. The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale. *arXiv preprint arXiv:1811.00982* (2018).

[60] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2017. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *J. Mach. Learn. Res.* 18 (2017), 185:1–185:52. http://jmlr.org/papers/v18/16-558.html

[61] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. 2018. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118* (2018).

[62] Tao Lin, Sebastian U. Stich, Kumar Kshitij Patel, and Martin Jaggi. 2020. Don't Use Large Mini-batches, Use Local SGD. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=B1eyO1BFPr

[63] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2017. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887* (2017).

[64] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. 2016. Ssd: Single shot multibox detector.

In *European conference on computer vision*. Springer, 21–37.

[65] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and Efficient GPU Cluster Scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*. USENIX Association, 289–304. https://www.usenix.org/conference/nsdi20/presentation/mahajan

[66] Qi Meng, Wei Chen, Yue Wang, Zhi-Ming Ma, and Tie-Yan Liu. 2019. Convergence analysis of distributed stochastic gradient descent with shuffling. *Neurocomputing* 337 (2019), 46–57. https://doi.org/10.1016/j.neucom.2019.01.037

[67] Apache Mesos. 2020. RecordIO Data Format. https://mesos.apache.org/documentation/latest/recordio/.

[68] Supun Nakandala, Yuhao Zhang, and Arun Kumar. 2020. Cerebro: A Data System for Optimized Deep Learning Model Selection. *Proc. VLDB Endow.* 13, 11 (2020), 2159–2173. http://www.vldb.org/pvldb/vol13/p2159-nakandala.pdf

[69] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. ACM, 1–15.

[70] Deepak Narayanan, Keshav Santhanam, Amar Phanishayee, and Matei Zaharia. 2018. Accelerating deep learning workloads through efficient multi-model execution. In *NIPS Workshop on Systems for Machine Learning (December 2018)*.

[71] Andrew NG. 2020. Data and DNNs. https://www.wired.com/brandlab/2015/05/andrew-ng-deep-learning-mandate-humans-not-just-machines/.

[72] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, et al. 2017. Can FPGAs beat GPUs in accelerating next-generation deep neural networks?. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 5–14.

[73] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. 2015. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper* 2, 11 (2015), 1–4.

[74] Luis Perez and Jason Wang. 2017. The effectiveness of data augmentation in image classification using deep learning. *arXiv preprint arXiv:1712.04621* (2017).

[75] Philipp Probst, Anne-Laure Boulesteix, and Bernd Bischl. 2019. Tunability: Importance of Hyperparameters of Machine Learning Algorithms. *J. Mach. Learn. Res.* 20 (2019), 53:1–53:32. http://jmlr.org/papers/v20/18-444.html

[76] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Eric Peterson, Aaron Smith, Jason Thong, Phillip Yi Xiao, Doug Burger, Jim Larus, Gopi Prashanth Gopal, and Simon Pope. 2014. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA) (ISCA 2014)*. 13–24.

[77] 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 *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO-49)*. Piscataway, NJ, USA, 18:1–18:13.

[78] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. Imagenet large scale visual recognition challenge. *International journal of computer vision* 115, 3 (2015), 211–252.

[79] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. Imagenet large scale visual recognition challenge. *International journal of computer vision* 115, 3 (2015), 211–252.

[80] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4510–4520.

[81] Supheakmungkol Sarin, Knot Pipatsrisawat, Khiem Pham, Anurag Batra, and Luís Valente. 2019. Crowdsource by Google: A Platform for Collecting Inclusive and Representative Machine Learning Data.

[82] Kaz Sato. 2020. Google's first Tensor Processing Unit. https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu.

[83] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[84] Michael Stonebraker. 1981. Operating System Support for Database Management. *Commun. ACM* 24, 7 (1981), 412–418. https://doi.org/10.1145/358699.358703

[85] Mustafa Suleyman. 2020. Using AI to give doctors a 48-hour head start on life-threatening illness. https://deepmind.com/blog/article/predicting-patient-deterioration.

[86] Andrew Tomkins, R Hugo Patterson, and Garth Gibson. 1997. Informed multi-process prefetching and caching. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 25. ACM, 100–114.

[87] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[88] Subhashini Venugopalan, Marcus Rohrbach, Jeffrey Donahue, Raymond Mooney, Trevor Darrell, and Kate Saenko. 2015. Sequence to sequence-video to text. In *Proceedings of the IEEE International Conference on Computer Vision*. 4534–4542.

[89] Guanhua Wang, Amar Phanishayee, Shivaram Venkataraman, and Ion Stoica. 2020. Blink: A fast NVLink-based collective communication library. In *Proceedings of the 3rd Conference on Machine Learning and Systems, MLSys '20*. Austin, TX, USA.

[90] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *arXiv preprint arXiv:1609.08144* (2016).

[91] 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 *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. USENIX Association, 595–610. https://www.usenix.org/conference/osdi18/presentation/xiao

[92] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. 2017. Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 181–193.

[93] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 6848–6856.

[94] Hongyu Zhu, Amar Phanishayee, and Gennady Pekhimenko. 2020. Daydream: Accurately Estimating the Efficacy of Optimizations for DNN Training. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*. USENIX Association, 337–352.

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

[96] Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. 2015. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*. 19–27.

APPENDIX

This document contains supplementary material, describing experiments that were omitted in the paper for brevity.

A ANALYSIS OF DATA STALLS

Our paper shows the analysis of data stalls in DNN training across various models, datasets, and hardware configurations. Here, we provide additional analysis of prep stalls such as increasing the number of CPU cores per GPU beyond 3, and the data pipeline rates for different models, datasets and number of GPUs.

A.1 Data pipeline rates

Using DS-Analyzer, we measure the rate of different components in the data pipeline as shown in Figure 17. G indicates the GPU rate, P_g represents the pre-processing rate when using the GPU-mode of DALI, P_c is the prep rate for the CPU-mode of DALI, and F is the effective fetch rate. We show these quantities on `Config-SSD-V100` for 7 different models and varying GPU count for the OpenImages dataset in Figure 18, ImageNet-22k in Figure 19 and ImageNet-1K in Figure 20. The configurations and cache size for each dataset is shown in Table 4 When the GPU count is reduced, all other resources (CPU, and memory) are also proportionately reduced to maintain the SKU.

$P = max(P_g, P_c)$, the best of CPU or GPU-based prep

The speed of the data pipeline is given by : $min(P, F)$

Data stall exists in a given configuration if $G > min(P, F)$

We see that across several models and configurations, data stalls are prominent. The fetch rate shown here, assumes an efficient in-memory cache like MinIO that provides x hits per epoch if the cache has x items. If we rely on OS Page Cache, fetch rates observed are 20% lower, resulting in higher fetch stalls.

As we decrease the number of GPUs, the ingestion rate at the GPU G decreases, as they process data slower due to decrease in parallelism. In all configurations, the bandwidth of the storage device remains constant; therefore as we reduce the number of GPUs, F, catches up with G, and the bottleneck in the training pipeline shifts to pre-processing.

There following are the key takeaways from the rate graphs.

- Several models experience data stalls due to fetch or prep across a range of GPU configurations and datasets.
- Rich datasets like OpenImages (higher per-image size) result in higher data stalls as shown in Figure 18
- Even computationally expensive models like ResNet50 and VGG experience data stalls due to the costly pre-processing, despite using state-of-the-art data pipelines.
- GPU based pre-processing hurts the performance of some models like ResNet50 and VGG due to intereference with GPU computation.

A.2 Training on servers with high CPU count

Typically, servers optimized for ML training (for e.g., NVIDIA DGX-2) have 3 CPU cores per GPU [14]. However, some cloud providers like AWS have servers with 8 GPUs and 32 CPU cores (64 vCPUs), that results in 4 cores (or 8 vCPUs) per GPU. We analyze prep stalls in one such server with 8 V100 GPUs, 64 vCPUs, and 500GiB DRAM.

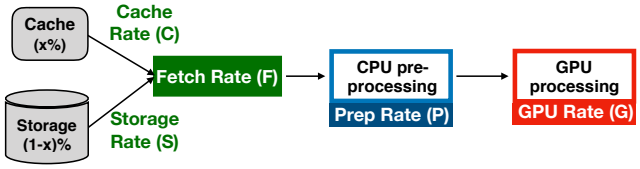


Figure 17: Data Pipeline in DNN training. This figure shows the different hardware components involved in DNN training and the throughput of each component.

Table 4: The table shows the number of GPUs, CPUs, and the percentage of dataset cached in memory for different configurations on `Config-SSD-V100`.

Num GPU	Num CPU	Mem (GB)	% dataset cached OpenImages	ImageNet-22K	ImageNet-1K
8	24	500	65%	35%	100%
4	12	250	32%	17%	100%
2	6	125	16%	9%	70%
1	3	62.5	8%	5%	35%

Figure 21 shows the training speed for a Resnet18 training job as we vary the number of vCPUs per GPU for both CPU-based and GPU-based prep pipeline with DALI. Note that the dataset is fully cached in memory and there are no fetch stalls in this experiment.

Resnet18 has 50% prep stall for 3 CPU cores per GPU, when GPU-based prep is used (shown by the GPU ingestion rate in Figure 21). With 8 vCPUs per GPU, prep stalls reduced to 37%, but did not vanish. Note that, pre-processing with CPU scales linearly upto the number of cores (here 4 per GPU), beyond that, hyperthreading does not result in linear gain in performance. Increasing the number of pre-processing threads in the server from 32 to 64 increased pre-processing speed only by 30%. In this experiment, we did not increase pre-processing threads per GPU beyond 5 in the GPU-prep mode of DALI, as it resulted in higher GPU memory consumption for prep and hence OOM. All the experiments presented in our main submission used 3 physical CPU cores per GPU (with GPU-prep of DALI where beneficial). This is only 25% slower than using 8 vCPUs per GPU (as shown in Figure 21). Additionally, the prep stall shown here is for ImageNet-1K; with richer datasets like OpenImages (higher per-image size), prep stalls increase further.

A.3 Comparing PyTorch DL with DALI

PyTorch has two different native modes for data parallel training; DataParallel (DP) and DistributedDataParallel (DDP). DP is usually slower than DDP even on a single server due to the Global Interpreter Lock (GIL) contention across threads, and additional overhead introduced by scattering inputs and gathering outputs across GPUs [20]. Figure 22 shows the epoch time for 7 different image classification models using the ImageNet-1K dataset(fully cached

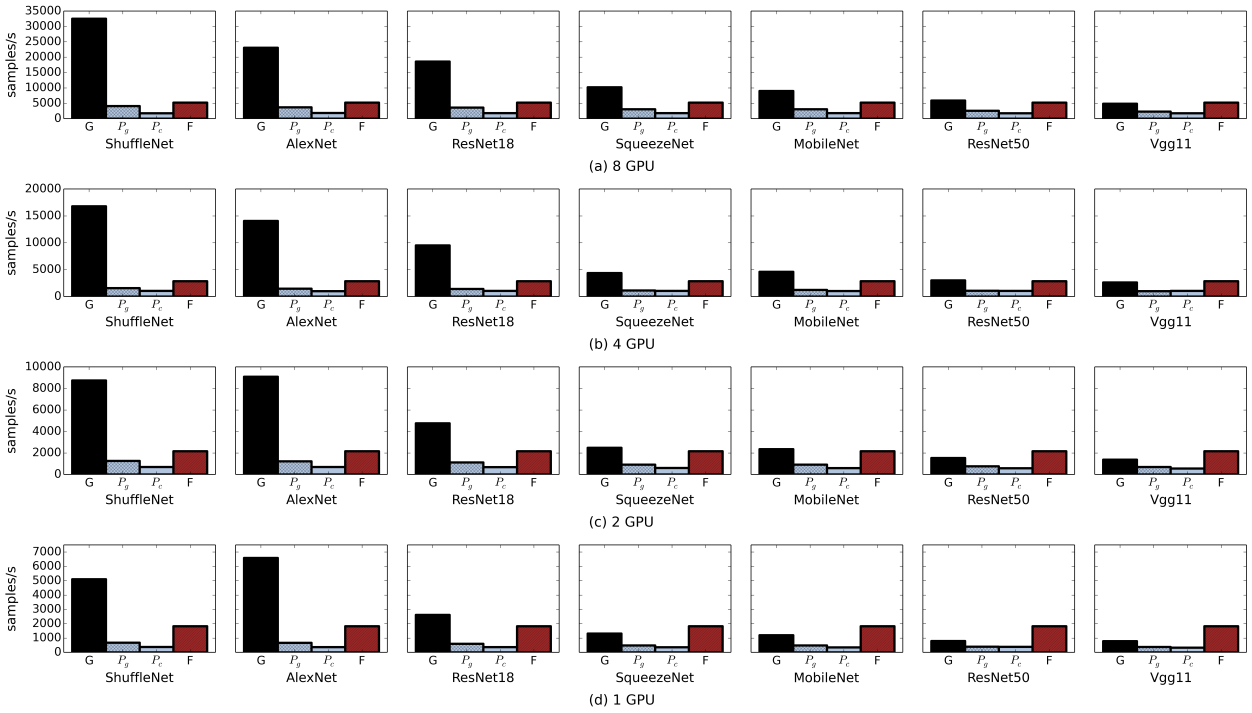


Figure 18: Data pipeline rates with OpenImages. This graph shows that several models across various configurations have data stalls; i.e., the black bar (G) is greater than the minimum of blue(P) and red(F) bars.

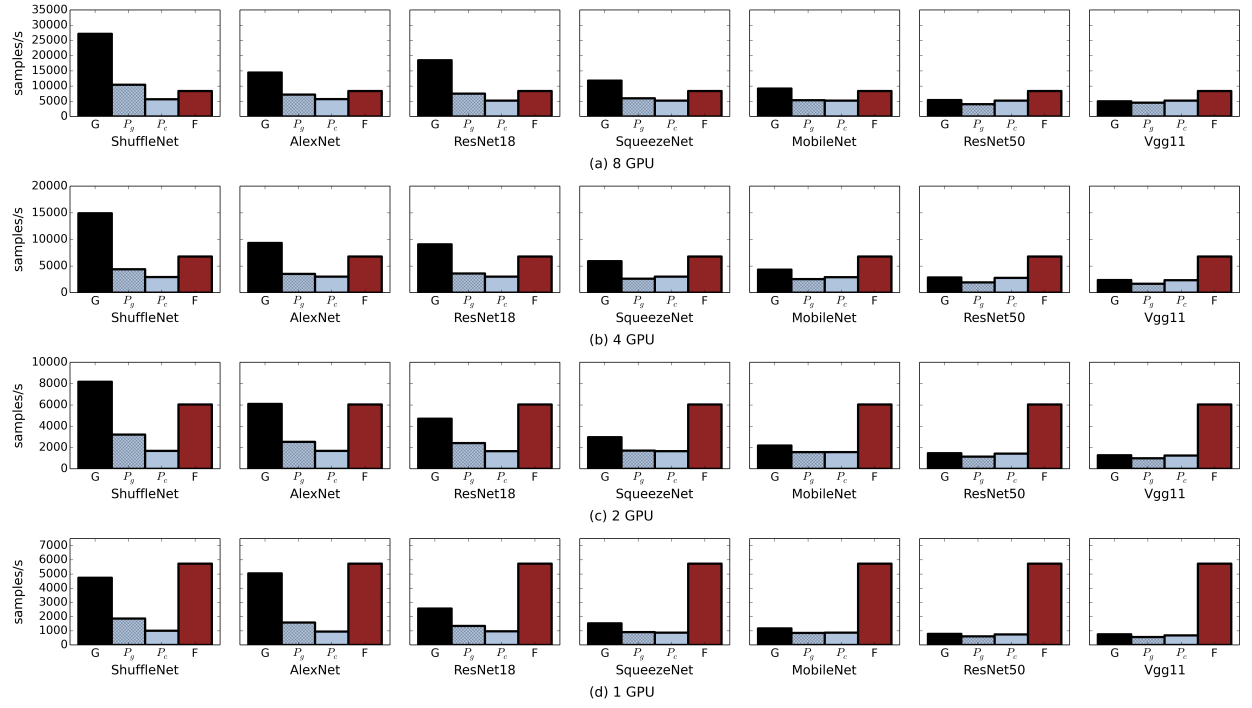


Figure 19: Data pipeline rates with ImageNet-22K. This graph shows that several models across various configurations have data stalls due to fetch and prep; i.e., the black bar (G) is greater than the minimum of blue(P) and red(F) bars.

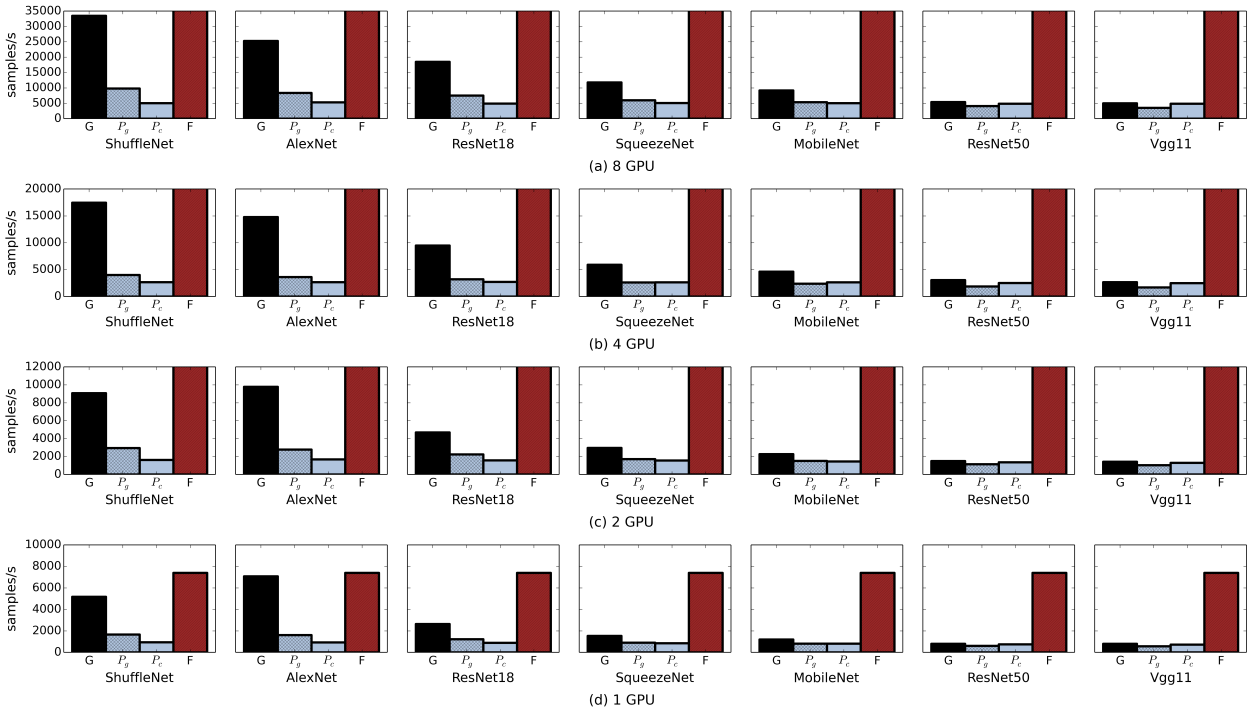


Figure 20: Data pipeline rates with ImageNet-1K. This graph shows that several models across various configurations have prep stalls; i.e., the black bar (G) is greater than the blue(P) bars. Since the dataset fits in memory in most configurations, F is high.

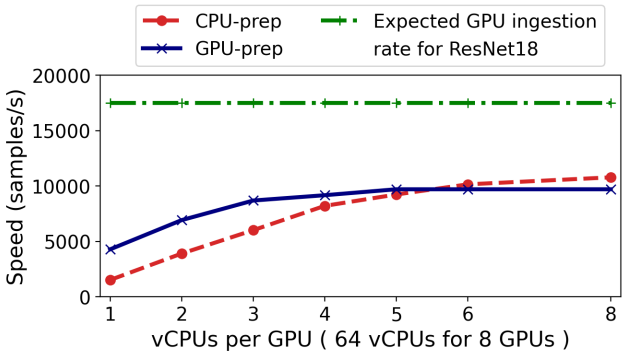


Figure 21: Impact of CPU on prep. This graph plots the epoch time for ResNet18 on `Config-SSD-V100` as we vary the number of vCPUs per GPU. Although epoch time decreases with higher vCPUs, at 8vCPUs per GPU, ResNet18 has 37% prep stalls. With the GPU-prep of DALI, we do not increase threads beyond 5 per GPU as it results in GPU OOM.

in memory) using native PyTorch DL with the faster DDP mode and DALI. PyTorch DL uses the Pillow library [32] and TorchVision [22] for image decoding and pre-processing while DALI uses the optimized nvJPEG library [15], therefore resulting in faster pre-processing even when using only CPU. When the GPU based DALI pipeline is used, training time further drops due to reduction in prep stalls. However, note that there are two downsides to using DALI's GPU based prep. First, it takes up 2-5GB of additional GPU memory for pre-processing, the luxury of which may not be available for all models and GPUs, as GPU memory is limited. Second, scheduling pre-processing on the GPU hurts models like ResNet50, as they are already heavy on GPU computation. In all our analysis and evaluation presented in the paper, we run with both GPU and CPU based DALI pipeline and present the best of the two results (CPU-based prep was faster on ResNet-50 and VGG11).

B EVALUATION OF COORDL AGAINST DALI

Our paper evaluates CoorDL against DALI in various training scenarios. This document provides a more detailed evaluation of some aspects of CoorDL.

B.1 Evaluation with ImageNet22k

ImageNet-22k is the extended version of the popular ImageNet-1K dataset, and contains about 14 million images that belong to 21841 different categories [36]. The average size of an image in this dataset is about 90KB, much smaller than the average image size in OpenImages dataset (300KB), as well as ImageNet-1K (150KB).

When we train the image classification models with ImageNet-22k on `Config-SSD-V100`, MinIO results in 20% higher cache hits than DALI-shuffle that resulted in 1.5× faster training on ShuffleNet, and 1.4× faster on AlexNet and ResNet18.

Next, when we perform distributed training on these models on `Config-SSD-V100` with 2 servers, AlexNet trained 1.3× faster, Shufflenet trained 1.33× faster and ResNet18 achieved 1.12× speedup. The fetch stalls with ImageNet-22k was lower than a more complex dataset like OpenImages because of the low per-image size that

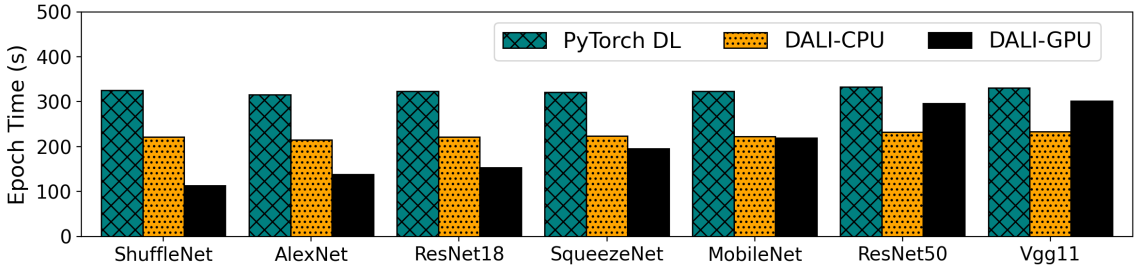


Figure 22: Epoch time with PyTorch and DALI. This graph plots the epoch time for various image classification models with native PyTorch DL and DALI (CPU-prep and GPU-prep) on `Config-SSD-V100`. DALI provides significant speedup over PyTorch even in its CPU mode due to the optimized nvJPEG decoding library. For compute heavy models like ResNet50, GPU based pipeline hurts performance because there is no idle time at the GPU that can be used for pre-processing, and thus interferes with GPU computation.

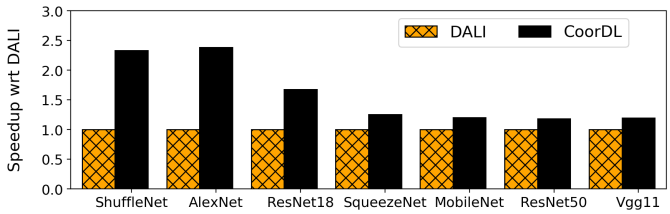


Figure 23: HP search with ImageNet-22k dataset. This plot shows the normalized training speed wrt DALI, when training 8 concurrent HP search jobs on `Config-SSD-V100`.

Table 5: Impact on fetch misses and disk IO. When training ResNet18 on OpenImages (645GB), CoorDL reduces cache misses from 66% to 35%. `Config-SSD-V100` caches 65% of the dataset, so this is the minimum miss rate.

	DALI-seq	DALI-shuffle	CoorDL
Cache miss	66%	53%	35%
Disk IO (GB)	422	340	225

increased the the number of samples the storage can deliver per second.

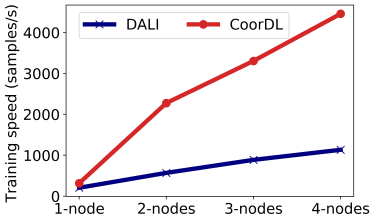
Finally, we perform HP search with eight concurrent jobs on `Config-SSD-V100` on the seven image classification models. As shown in Figure 23, CoorDL results in upto 2.5× speedup.

B.2 Cache misses with CoorDL

CoorDL's MinIO cache is designed to minimize the amount of storage I/O per epoch, by efficiently utilizing the all the items in cache. Table 5 enumerates the fetch misses and total disk I/O for DALI-seq, DALI-shuffle and CoorDL when training ShuffleNetv2 on OpenImages dataset on `Config-SSD-V100`. This server can cache 65% of the dataset. CoorDL is able to reduce disk I/O by 47% compared to DALI-seq and 33% compared to DALI-shuffle, by reducing thrashing by 47% and 33% respectively. MinIO cache is able to reduce the cache misses down to capacity misses.

B.3 Scalability of partitioned caching

Our paper shows that when training is distributed across just enough servers that can cache the entire dataset in memory, partitioned caching can speed up training jobs by upto 15×. However, when we distribute training to more servers, such that their aggregate memory is higher than the total dataset size, CoorDL continues to outperform DALI as shown in Figure 24a. In this experiment, we train ResNet50 on OpenImages on `Config-HDD-1080Ti`, where each server can cache 65% of the dataset. When training extends to 24 GPUs(3servers), or 32 GPUs(4 servers), the throughput with CoorDL increases because, training is not bottlenecked on I/O and more GPUs for training naturally results in faster training due to increase in GPU parallelism. With DALI, although the throughput increases, it is still I/O bound; the increase in throughput is due to the reduced disk I/O per server when training is distributed as shown in Table 24b. Although the I/O per server decreases with DALI as we distribute training across more servers, note that the GPU parallelism is also proportionately increasing; the GPU compute rate (G) and prefetch rate (F), are proportionately increasing, leaving the performance gap the same.



(a) Distributed training

Nodes	Disk IO(GB)
1	342
2	119
3	70
4	50

(b) Disk IO

Figure 24: Distributed training with CoorDL. The plot compares DALI with CoorDL when training ResNet50 across upto 4 nodes. Even when each node can cache 65% of the dataset, DALI results in I/O bound training due to disk fetch, while CoorDL results in zero disk accesses beyond first epoch.

Table 6: On `Config-SSD-V100`, when training with the small ImageNet-1k dataset that fits in memory, CoorDL provides upto 1.87× speedup by eliminating redundant prep

	Per job speed (Samples/s)	
Model	DALI	CoorDL
ShuffleNet	1441	1.81×
AlexNet	1399	1.87×
ResNet18	1056	1.53×
SqueezeNet	835	1.50×
MobileNet	752	1.35×
ResNet50	569	1.21×
VGG11	552	1.22×

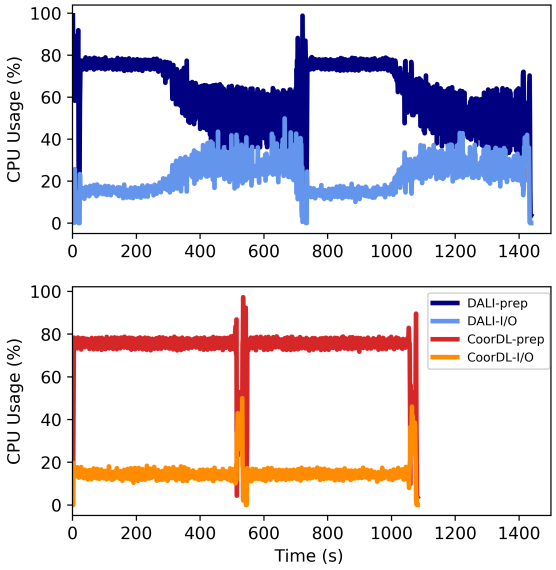


Figure 25: This plot shows the CPU utilization over time for DALI and CoorDL when training ResNet18 on OpenImages. CoorDL uses cache effectively to reduce disk I/O, therefore utilizing CPU on useful pre-processing rather than waiting on I/O

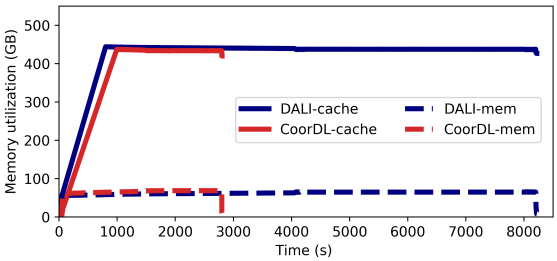


Figure 26: This plot shows the memory utilization for two epochs of HP search using AlexNet on OpenImages, with 8 concurrent jobs. CoorDL uses 5GB of extra process memory; resulting in 5GB lower cache space.

B.4 HP search with fully cached dataset

The core of CoorDL's ability to speed up HP search jobs comes from coordinating pre-processing to overcome the imbalance in the ratio of CPU cores to GPU. We perform HP search with 8 jobs on `Config-SSD-V100` with ImageNet-1k dataset that fits entirely in memory. As shown in Table 6, CoorDL sped up HP search by 1.9× on AlexNet and and 1.2× on ResNet50 by eliminating redundant pre-processing operations.

B.5 Resource utilization with CoorDL

CPU utilization with CoorDL. The paper showed how MinIO reduces the amount of data fetched from storage in each epoch and regularizes the data access pattern. Profiling the CPU during training of ResNet18 on OpenImages shows that the pre-processing threads in DALI are often stalled waiting for I/O as in Figure 25. Since MinIO reduces the total disk I/O, CoorDL is able to better utilize the CPU threads for pre-processing. The combination of lower disk I/O and better CPU utilization leads to shorter training times when using CoorDL.

Low memory overhead of co-ordinated prep. By design, co-ordinated prep has the same memory requirements as DALI. To experimentally validate this, we track the memory utilization of running hyperparameter search on AlexNet on OpenImages on a `Config-SSD-V100` server using eight concurrent jobs. Figure 26 plots the memory utilization over time for both the process working memory, and the cache. CoorDL uses 5 GB of extra process memory to store prepared mini-batches in memory until all hyperparameter jobs consume it. We reduce the cache space given to CoorDL by 5 GB (keeping the total memory consumption same for CoorDL and DALI). Despite the lower cache space, CoorDL still accelerated training by 2.9×.

C BUILDING PY-COORDL IN NATIVE PYTORCH

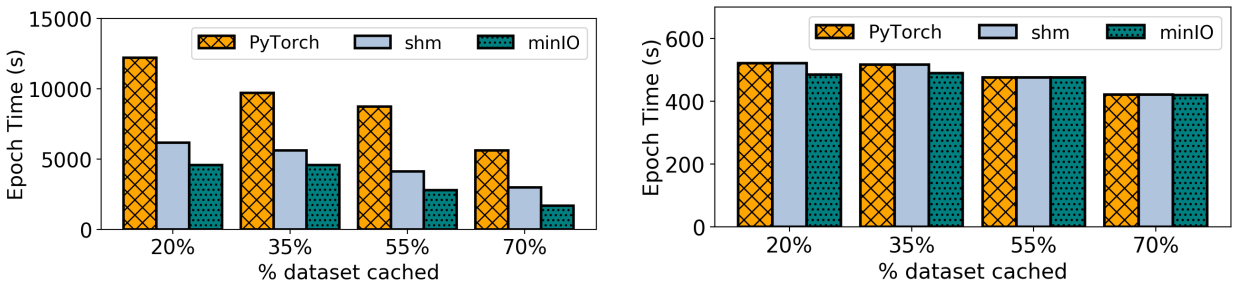
As a proof of concept, we implemented two of the techniques behind CoorDL, MinIO and *coordinated prep* as a pluggable module to the native PyTorch DL (without DALI). This section briefly describes the implementation and presents the evaluation of Py-CoorDL against the native PyTorch DL.

C.1 Implementation

Py-CoorDL is implemented as a pluggable DataLoader module for PyTorch with minimal changes to its current DataLoader API. Py-CoorDL is implemented in 650 lines of Python code. Py-CoorDL is implemented using Python's shared memory abstraction because PyTorch spawns multiple processes instead of threads to parallelize data fetch and prep (due to Python's Global Interpreter Lock limiting concurrency of threads).

C.2 Evaluation

We evaluate Py-CoorDL on a server with 8 `V100` GPUs, each with 16GB of GPU DRAM. Our server is 2 socket, 14-core Intel Xeon E5-2690@2.6GHz, with 500GB DRAM, and 2 different storage devices (SSD and HDD). We evaluate Py-CoorDL on five image classification models; AlexNet [57], ResNet18 [44], ShuffleNetv2 [93],



(a) HDD **(b) SSD**

Figure 27: Evaluation of MinIO caching policy. The graphs compare the native PyTorch DataLoader with Py-CoorDL's MinIO caching policy, and shows the speedup due to the two components in MinIO; sequential access in shared memory(shm), and increased cache hits(MinIO). On HDDs the sequential access of image files in shm provides significant speedup. On SSD, benefits with MinIO are marginal because training is bottlenecked on CPU prep.

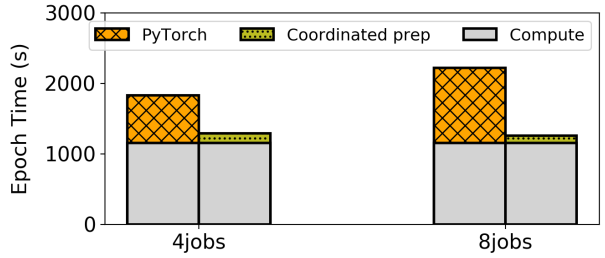


Figure 28: Evaluation of coordinated prep. This graph compares PyTorch against Py-CoorDL's coordinated prep when training 6 or 8 HP search jobs on a server. Coordinated prep is able to reduce prep stalls significantly as compared to PyTorch

SqueezeNet [46] and MobileNetv2 [80]. We set the batch size to the maximum that fits the GPU (512 for Alexnet, Shufflenet and ResNet18, 256 for the others) . We train the model for 5 epochs and report the average epoch time excluding the first warmup epoch. We use the ImageNet 1K dataset of size 146GiB [79] and PyTorch 1.1. To evaluate the benefits of Py-CoorDL, we run our jobs in a Docker container with restricted memory to mimic the scenario where the dataset does not entirely fit in DRAM. This is equivalent to running the full ImageNet dataset (22K classes - 1.3TB) on our server.

C.2.1 ***Multi-GPU training in a server***. We now evaluate the benefit of using Py-CoorDL to perform multi-GPU training in a single server.

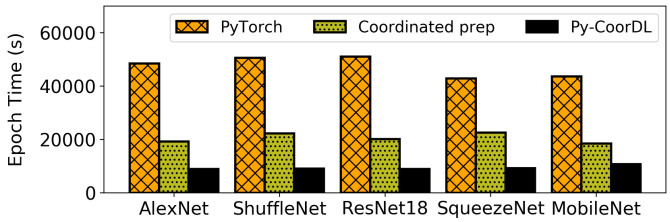
Hard drives. Figure 27a plots the stabilized per epoch time as a function of cache size for ResNet18. In this experiment, DataParallel training is performed on 8 GPUs, each with a batch size of 512 and a total of 24 data workers pre-processing in parallel. Py-CoorDL brings down the per-epoch training time by 2.1×- 3.3×. This is due to two reasons. First, Py-CoorDL increases the sequentiality of reading data items from the disk by indexing the entire data item instead of individual pages. Each data item in the ImageNet dataset is on average 150KB, which spans about 28 pages on disk. The native PyTorch DL fetches the pages of data items on demand, whenever the CPU thread requires to decode the item. As multiple data workers decode images in parallel, the pages from different images were requested randomly. Py-CoorDL reduces this randomness by reading the entire data item into memory, before decoding it. Second, MinIO caching policy results in 20% lower cache misses as compared to the page cache's LRU scheme. Given the low throughput of disks (15MBps), this translates to a high savings in training time.

Solid state drives. Figure 27b shows the variation in training time for different cache sizes, when the dataset is accessed from a fast solid state drive (SSD). The throughput of the SSD is 500MBps. Reduction in cache trashing does not reduce the training time significantly because we are bottlenecked on pre-processing at the CPU (pre-processing throughput is around 327MBps). Therefore, the 20% reduction in store misses translates to a mere 7% lower training time. Note that when an optimized library like DALI is used for pre-processing, the CPU prep rate increases, making storage the bottleneck; this makes MinIO's savings more significant with DALI.

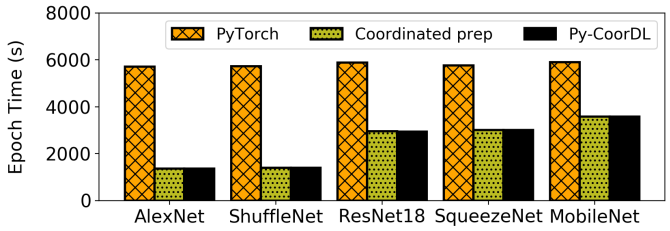
C.2.2 ***HP Search***. To evaluate the benefits of coordinated prep, we construct a microbenchmark where each job trains the ResNet18 model on a single GPU in a server, when the entire dataset is cached in memory. We evaluate two scenarios; 4 jobs, each using 6 data workers for pre-processing (4 GPU and 24 CPU), and 8 jobs with 3 data workers each (8 GPU and 24 CPU). The per-epoch time for these scenarios is shown in Figure 28. As the number of concurrent jobs increase, the data stall time increases because each job gets fewer CPU cores for pre-processing. Py-CoorDL reduces the data stall time close to 0 in both cases. It does so by launching a unified data loading process that pre-processes the dataset exactly once per epoch using all 24 CPUs, and shares the prepared batches across all the jobs. This technique results in 1.8× lower training time when 8 jobs are run concurrently on a single server.

C.2.3 ***End-to-end benefit of Py-CoorDL***. We now evaluate the end-to-end benefit of CoorDL using a macrobenchmark; HP search using Ray Tune [61] when dataset does not fit entirely in memory.

Ray Tune. Ray Tune [61] is a HP optimization framework that provides the flexibility of using various search algorithms such as Population Based Training (PBT), Median Stopping Rule, and HyperBand. Ray Tune uses one of these algorithms to pick a unique



(a) End-to-end workload on HDD



(b) End-to-end workload on SSD

Figure 29: End-to-end evaluation. The graphs compare the total search time for HP optimization on Ray Tune using the baseline PyTorch DL and Py-CoorDL on hard disks and solid state drives. It also shows the contribution of individual components; when just coordinated prep is used without MinIO (indicated as coordinated prep) and when both techniques are used (shown as Py-CoorDL). On SSDs, MinIO does not help accelerate training as much as it does on HDDs, because the fetch rate is higher than the CPU prep rate on SSD.

value for the HP, and launches a training job on one of the available GPUs. We modified Ray Tune's job executor to use Py-CoorDL and launch training jobs one on each available GPU in a server. We used the Hyperband search algorithm to sample 16 values of (learning rate, momentum) pairs and set the stopping criteria to be the completion of one epoch for brevity. The trends remain the same if the stopping criteria is set to a target accuracy.

Experiment setting. We run this macrobenchmark on a machine with 8 GPUs (8 samples are trained in parallel). For the PyTorch DL, we set the number of data workers to 3 for each job and to evaluate Py-CoorDL, we set data workers to 24. Note the total number of data workers in the system is the same in both cases. We set the cache size set to 110GB (≈ 75% of the dataset). We record the total reduction in search time compared to the baseline, and the contribution of each of our techniques, *coordinated prep* alone, and when coordinated prep is combined with MinIO caching. We evaluate the benefits of Py-CoorDL in two scenarios; when the dataset resides on a slower storage media like hard drive and when it is on a relatively faster media like solid state drive.

Dataset resides on hard drive. As shown in Figure 29a, coordinated prep alone results in upto 2.5× speedup in total search time by reducing the total disk accesses by 2.5×. The savings in time comes directly from the reduced disk accesses because the DataLoader in this case is bottlenecked on I/O rather than pre-processing. When MinIO caching policy is enabled, the effective speedup is close to 5.5× due to reduced storage miss and reduction in random accesses.

Dataset resides on solid state drive. When the dataset is on a faster medium like SSD, whose throughput is higher than that of pre-processing, the bottleneck in the DataLoader shifts to CPU. In this scenario, as shown in Figure 29b, coordinated prep reduces the overhead of pre-processing and speeds up search time by reusing prepared minibatches across jobs. With the addition of MinIO policy, the search does not speed up significantly due to cheap IO.

C.3 Summary

Py-CoorDL speeds up DNN training jobs by 2×- 5.7× by enabling efficient reuse of both raw data items and pre-processed batches. Although Py-CoorDL has marginal gains when dataset resides on SSD, the reason was the slow pre-processing rate of data transformation operations used by PyTorch DL. If prep rate goes up, fetch stalls become prominent, and MinIO comes to the rescue, which is the case when using DALI for pre-processing [8].