Stash: A comprehensive stall-centric characterization of public cloud VMs for distributed deep learning

Aakash Sharma, Vivek M. Bhasi, Sonali Singh, Rishabh Jain,
Jashwant Raj Gunasekaran†, Subrata Mitra†, Mahmut Taylan Kandemir, George Kesidis, Chita R. Das
Computer Science and Engineering, Pennsylvania State University
Adobe Research†
{abs5688, vmbhasi, sms821, rishabh, mtk2, gik2, cxd12}@psu.edu, {jgunasekaran, subrata.mitra}@adobe.com

***Abstract*—Deep neural networks (DNNs) are increasingly popular owing to their ability to solve complex problems such as image recognition, autonomous driving, and natural language processing. Their growing complexity coupled with the use of larger volumes of training data (to achieve acceptable accuracy) has warranted the use of GPUs and other accelerators. Such accelerators are typically expensive, with users having to pay a high upfront cost to acquire them. For infrequent use, users can, instead, leverage the public cloud to mitigate the high acquisition cost. However, with the wide diversity of hardware instances (particularly GPU instances) available in public cloud, it becomes challenging for a user to make an appropriate choice from a cost/performance standpoint.**

In this work, we try to address this problem by (i) introducing a comprehensive distributed deep learning (DDL) profiler Stash, which determines the various execution stalls that DDL suffers from, and (ii) using Stash to extensively characterize various public cloud GPU instances by running popular DNN models on them. Specifically, it estimates two types of communication stalls, namely, interconnect and network stalls, that play a dominant role in DDL execution time. Stash is implemented on top of prior work, DS-analyzer, that computes only the CPU and disk stalls. Using our detailed stall characterization, we list the advantages and shortcomings of public cloud GPU instances for users to help them make an informed decision(s). Our characterization results indicate that the more expensive GPU instances may not be the most performant for all DNN models and that AWS can sometimes sub-optimally allocate hardware interconnect resources. Specifically, the intra-machine interconnect can introduce communication overheads of up to 90% of DNN training time and the network-connected instances can suffer from up to 5× slowdown compared to training on a single instance. Furthermore, (iii) we also model the impact of DNN macroscopic features such as the number of layers and the number of gradients on communication stalls, and finally, (iv) we briefly discuss a cost comparison with existing work.

I. Introduction

The continual growth of Deep Learning (DL) has fuelled many facets of Artificial Intelligence such as machine vision [37], natural language processing [9], neuromorphic computing [44], [45] etc. The advancements in DL have mainly been driven by the availability of large amounts of training data as well as powerful compute platforms such as CPU or GPU clusters, TPUs, NPUs and other accelerators that can handle increasingly complex/heavy Deep Neural Network (DNN) computations. However, the ever-growing DNN-model and training data sizes accompanied by the increasing ubiquity of DNNs place a higher demand on compute resources for faster processing speeds and shorter overall training time. Although current accelerators enable faster training, they are typically expensive to maintain, owing to their power-hungry nature. This potentially renders them cost-ineffective, especially in intermittent training scenarios. To avoid the prohibitively high upfront cost of purchasing a GPU machine/cluster, users employ public cloud GPU resources to run their workloads.

Public cloud providers such as AWS, Azure, and GCP provide a gamut of GPU instance offerings. These offerings vary in their hardware configurations and pricing. Cloud providers typically do not allow any flexibility in changing the CPU vCores, memory or GPUs of an instance, thereby limiting users to select from *pre-configured* instances. Note that the choice of instance type(s) drives the total cost of training a model [7] and users may rely on benchmarks such as DawnBench [7], NVIDIA examples [35], etc. or on their intuition to choose the best instance(s) for their needs.

To address this problem, we introduce a Distributed Deep Learning (DDL) profiler Stash, which can measure the various execution stalls (on network, CPU and disk) that a typical DDL pipeline experiences. Using our profiler, we characterize various public cloud GPU instances from both a cost and performance perspective with emphasis on communication-related stalls. This characterization provides novel insights into public cloud GPUs and its network, which can be used by tenants to make an informed decision vis-a-vis choosing the right DDL cluster configuration for their specific model.

Stash is built by extending existing profiler DS-analyzer [31] which characterizes single-node DNN jobs in a private Microsoft cluster with emphasis on the bottlenecks (stalls) caused due to CPU pre-processing and storage I/O latency. However, it has a key omission of not profiling communication-related stalls. Compared to the single-node scenario, where the primary stalls were observed to be CPU and/or disk I/O stalls in the DS-Analyzer work, we observe communication stalls to be the primary bottleneck in both single and multi-node DDL (which is also corroborated by prior works [33], [48]). In fact, storage-related stalls can (at least partially) be eliminated through DRAM caching in early epoch(s) but communication

2575-8411/23/$31.00 ©2023 IEEE
DOI 10.1109/ICDCS57875.2023.00023

stalls hamper every iteration of a typical DDL, thus proving to be a more pressing concern.

Motivated by this, we propose novel techniques to profile the communication-related stalls of DDL and implement it as part of Stash. Next, using the profiler, we extensively characterize public cloud GPU instances for the various stalls they suffer from while executing a typical DDL pipeline. A stall analysis on public cloud (AWS in our experiments) is particularly useful, since instances differ not just in their hardware offering, but also in the QoS they provide, as discussed in later sections. Moreover, a systematic study of the communication overhead of public cloud instances for DDL is lacking, partly due to the lack of publicly-available tools or profiling methodologies to measure such an overhead.

Prior DDL profiling related work such as [28] and [55] only describe methodologies to estimate and simulate communication overheads. While Srifty [28] characterizes some AWS GPU instances, they do not provide an analysis of the various slowdowns instances may experience. Instead, they only provide the DDL throughput offered by instance type (and cost incurred) without explaining possible causes. Further, they do not dive into the hardware characteristics of each instance type, including the interconnect, and the various idiosyncrasies that may degrade performance. Thus, they simply suggest that preferring larger GPU instances is always beneficial and that instance throughput scales near linearly with added GPUs. However, our analysis suggests that this is not always true. Also, though they analyse the variance of the AWS network bandwidth, we note here that network QoS is subject to high temporal (up to months) and spatial (availability zones, regions) variations and is hard to definitively characterize, unlike intra-node hardware. On the other hand, DS-Analyzer [31] studies DDL 'fetch' and 'prep' stalls (see next section), but does *not* study network stalls.

Hence, we conduct an extensive stall-based characterization of various GPU-accelerated instances of a public cloud using Stash. Furthermore, using this profiler, we analyze a number of DNN architectures to understand which architectural properties (such as the number and sizes of layers) drive communication stall behavior. This work attempts to understand and introspect the peculiarities of DL on public cloud VMs to help advance systems research.

Our **main contributions** in this paper can be summarized as follows:

- We introduce Stash, a profiling tool which can measure communication stalls (in addition to CPU and disk stalls) of DDL running on both single and multiple nodes.
- We perform stall-centric characterization of various AWS GPU instances, using a number of popular DNN models. The estimated communication overheads from intra-machine interconnect are found to be up to 90% of the training time and network-connected instances are found to be slowed down by up to 5× compared to a single node instance. Our profiling has led us to some surprising discoveries regarding the communication overhead experienced by AWS GPU instances.
- We identify the limitations of each instance type. Specifically, our results indicate that higher capacity GPU instances do not always lead to better performance and that AWS hardware interconnects may have various shortcomings.
- We identify architectural features in DNN models that influence communication stall behavior, namely, the number of layers as well as the total number of parameters (size of the DNN model).

The rest of this paper is organized as follows. In Section II, we discuss the background pertinent to AWS GPU instances and DS-Analyzer. In Section III, we motivate our problem and discuss related work. Our characterization scheme is described in Section IV. The results from our characterizations are presented in Sections V and VI. And finally, Section VIII summarizes our major observations and findings.

II. Background

In this section, we provide an overview of the GPU instance family of AWS, along with background on prior work.

A. Public Cloud Offerings

Hardware capabilities, both in terms of compute and interconnects, are particularly significant in the context of DDL on the public cloud, as the GPU instances offered by providers (such as AWS) have fixed configurations [3], thereby limiting user choice of a custom single-node training solution. Table I lists the P family GPU instance types offered by AWS along with their hardware specifications and pricing. The P4 instances have the most powerful GPUs (NVIDIA A100 Tensor core GPUs), while the P3 and P2 instances respectively have the less powerful, yet quite capable, NVIDIA V100 and K80 GPUs. The P3 and P2 instances are of particular interest to us, as they offer the most variety in terms of the number of GPUs available per node amongst all GPU instances viable for DNN training.[1]

Apart from the GPUs used, the interconnects and network links available to these instance types also have a significant impact on the end-to-end training time as they dictate data transfer speeds during various training steps. Specifically, interconnect links are utilized during gradient communication among workers (GPUs) present on the same physical node, whereas network links are used when the communication is between workers on different nodes. The interconnect architecture for the AWS p3.16xlarge and p3.24xlarge instances [19] is depicted in Figure 1.

B. DS-Analyzer: Stall Characterization

Among the studies that characterize the private cloud DS-Analyzer [31] is of particular significance to us as it also aims to identify DNN training bottlenecks, specifically with regards to 'fetch' and 'prep' stalls. These stalls refer to the time spent fetching mini-batches of data from the disk (fetch stall) and pre-processing it prior to training with it (prep stall).

[1]P4 is a dedicated offering not considered herein.

Instance type(s)		GPU(s)	VCPUs	Interconnect	GPU Memory (GB)	Main Memory (GB)	Network Bandwidth (Gbps)	Price/hr
P4		8×A100	96	NVSwitch	320	1152	400	$32.7726
P3	p3.2xlarge	1×V100	8	PCIe	16	61	up to 10	$3.06
	p3.8xlarge	4×V100	32	PCIe + NVLink	64	244	10	$12.24
	p3.16xlarge	8×V100	64	PCIe + NVLink	128	488	25	$24.48
	p3.24xlarge	8×V100	96	PCIe + NVLink	256	768	100	$31.218
P2	p2.xlarge	1×K80	4	PCIe	12	61	¡ 10	$0.90
	p2.8xlarge	8×K80	32	PCIe	96	488	10	$7.20
	p2.16xlarge	16×K80	64	PCIe	192	732	25	$14.40

TABLE I: AWS GPU instance types with prices (N. Virginia).

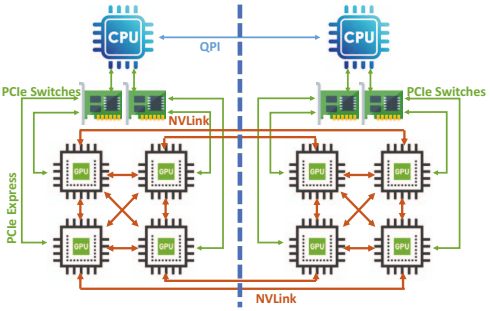


Fig. 1: P3.16xlarge interconnect architecture.

DS-Analyzer uses three steps to calculate prep and fetch stalls (refer to Figure 2). Step ② pre-populates synthetic data in the GPUs and runs training to measure the maximum ingestion rate of the system. This is followed by step ③ which runs training on actual data but with all OS caches cleared[2]. Finally, in step ④, training is run over actual data such that the entire data is cached in main memory (from the previous step). The prep stall is calculated by finding the difference between ④ and ②. This is because there is no disk I/O involved in step ④ and any difference in training time after deducting the time spent in GPU processing of ② will yield the time spent in pre-processing at CPU. After this, the fetch stall is calculated by finding the difference of ③ and ④, since any increase in time over ④ would be due to disk I/O.

III. DDL Stall Analysis in Public Cloud

In this section, we highlight the novelty of this work over prior work and discuss related work. We also motivate the importance of "profiling communication stalls" in DDL, especially on the public cloud.

Are there prior works which analyze communication overhead in DDL?

In [33], the authors measure the communication overhead of training to be 80% of the entire training time. However, they do not specify any general methodology to measure the actual overhead. In comparison, [48] characterizes DDL workloads on Alibaba PAI [6] and observes the communication overhead to be 62% with the use of parameter server (PS) [24] (whose communication performance is strictly less than all-reduce [23], [48]). Moreover, the said work is specific to a private cluster and specific hardware details of the machines running the workload are not available. Although they have released a general profiling methodology using TensorFlow [1] internal tooling and manual feature extraction, it is limited to TensorFlow and the PS communication architecture. They build a general framework for measuring the DDL performance, specifically for transient cloud instances which are frequently revoked. This work, too, does not account for communication stalls. Finally, tools such as nsight [34], nvprof [36] etc. are incapable of measuring communication stalls as this requires instrumenting the model and/or framework.

Why should a communication profiler for DDL be introduced?

A communication stall profiler can potentially help in the end goal of reducing it. In the past, several distributed DNN algorithms have been proposed [8], [11], [13], [24], [26], [53], [54] to reduce communication overhead of DDL. However, there is a lack of a *profiling tool* to measure the real world efficacy with regards to the communication overhead for various algorithms. While [55] aims to predict the efficacy of optimizations introduced in DNNs, it only estimates the communication overhead through simulations based on the available bandwidth and size of the communicated gradients. A similar strategy is employed in [28]. Both these works do not measure the actual communication overhead in the real world, which depends on a complicated asynchronous communication pattern and, consequently, suffers from varying bandwidth availability. Another question that can be asked here is: why not instrument the framework directly to measure communication overhead? The problem with instrumenting the framework is that every layer needs to be instrumented separately since the compute and communication is overlapped. The instrumentation would need to measure when computation ends and communication begins. This requires additional CUDA synchronization calls for each layer, since GPU kernels are launched asynchronously and communication is run concurrently. Such synchronizations might severely alter the execution runtime [55].

Why characterize GPU instances of public clouds?

The gamut of public cloud GPU-based instances available (see Section II-A) makes the task of choosing the most performant configuration a non-trivial one for end-users. This is due to the presence of various stalls in the DDL pipeline as well as the lack of a good scientific study that characterizes these stalls in the various public cloud GPU instances. Prior works on DL characterization such as [2], [14], [17], [20], [28], [31], [32], [48] do not characterize the instances of the public cloud for their QoS; consequently, users cannot use these works to

[2]In the original paper, this step is described to be after the next step but we observed from the DS-analyzer open source code that that is not the case.

choose the appropriate instance type(s). To further complicate matters, cloud providers offer different types of interconnects for their GPU-accelerated instance type(s). Apart from the interconnect type, the user can also "tie" various instances through a computer network. These communication options introduce further complexity to the choice of instance(s) for DDL. Without a good characterization study to identify what slowdowns various hardware of the public cloud induce, existing work on selecting an appropriate cloud configuration may fall well short of an optimal solution.

How do existing cloud configuration management systems fall short?

Prior work such as [28], [51] attempt to find the best VM configuration to run DL while satisfying user constraints. Although the end objective is met in their respective experiments, these works require extensive characterization and profiling to accomplish it. With poor characterization, such works may not predict the "true" throughput of a VM for a given model. For instance, none of them consider contention in intra-node network in their performance estimation models. Moreover, the cost of profiling itself is high and is not considered in the cost savings shown in their experiments. Our work too incurred cost in running the characterization experiments but the users can use the takeaways without running any further experiments. Essentially, the cost of automating a recommendation system for cloud configuration is often not considered. However, our work can be used by users to find an optimal VM configuration (albeit manually) without any extra cost to them.

IV. Methodology

In this work, we use the AWS public cloud to run all experiments. Specifically, we run DDL experiments in the AWS N. Virginia region using P type instances, which are AWS's recommended instance type for DL. All DDL models are run using PyTorch distributed in *synchronized data parallel* setup [25] with *collective allreduce* [10], [38] for gradient exchange. This setup is known to have better reproducability, convergence and performance [28]. We do not focus on the parameter server [24] communication protocol as its performance has been shown to be strictly less than allreduce.

Using our profiler Stash, we characterized various AWS P type instances (instance family recommended for training) with reference to four stall parameters, namely, (i) interconnect stall, (ii) network stall, (iii) CPU stall (prep stall), and (iv) disk stall (fetch stall). While these stalls provide important insights into the hardware characteristics of AWS instances, we also provide a training time and monetary cost comparison of running DDL on various AWS instance types. Our characterization exploits the repetitive nature of DL, and is able to calculate the various stalls from a single epoch. This is possible since the stall characteristics of a single epoch are representative of that of the entire training time (which scales linearly with the number of epochs).

A. Characterization

We conduct two types of characterizations, macro and micro, on AWS instances as explained below.

Macro Characterization: We run DDL using the models listed in Table II to characterize relevant GPU instances. We use two types of DNN models in this work: convolutional (CNNs) and transformer-based. Image classification with CNNs is the most common task used for evaluating ML-system performance [30]. A more recent work [28] uses CNNs for their characterization study. Transformers are growing in popularity and use cases, especially for NLP. We do not characterize much larger models such as DLRM as cheaper VMs from the public cloud are infeasible for them. Such large models may best be run on large dedicated instances such as the AWS P4 equipped with A100 GPUs and NVswitch interconnect. The P4 family has only a single type of instance and hence, a characterization study is not necessary. Large DNN models often do not fit on a single GPU's memory, thereby forcing users to employ techniques such as model and hybrid parallelism to train the model with multiple GPUs. Our profiling tool currently supports only data parallelism as stalls can be fully expressed through it.

Domain	Type	Name	Gradient size	Input Dataset
Vision	Small	AlexNet [21]	9.63M	Imagenet1k [16] (133 GB)
		MobileNet-v2 [40]	3.4M	
		SqueezeNet [15]	0.73M	
		ShuffleNet [29]	1.8M	
		ResNet18 [12]	11.18M	
	Large	ResNet50 [12]	23.59M	
		VGG11 [43]	132.8M	
NLP		BERT-large [9]	345M	SQuAD 2.0 [39] (45 MB)

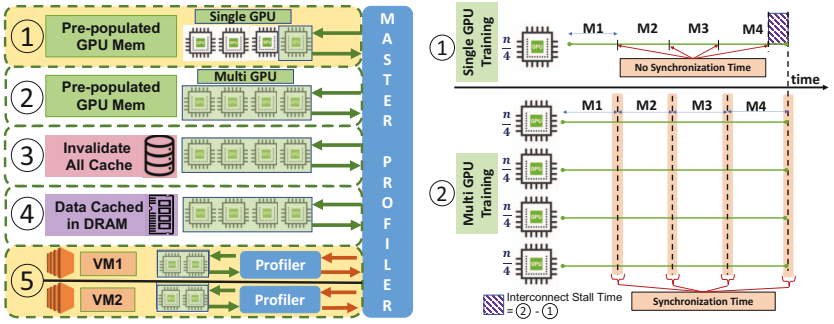
TABLE II: DDL models used.

Micro Characterization: We conduct micro characterization by running synthetic training experiments using two models – ResNet and VGG. As part of this, we study various aspects of the model architecture that influence the communication stalls including the number of layers. We also modulate certain model architecture features (such as "residual" network branches and batch normalization layers) to study their impact on communication stalls.

B. Profiler Design

A schematic view of the Stash profiler is depicted in Figure 2(a). In the figure, step ① and step ⑤ are our contribution and steps ②, ③, and ④ are from the prior work, DS-Analyzer. Note that Stash pre-populates the GPU memory with synthetic training data as part of step ①, ② and ⑤, and runs training over it. Training over synthetically pre-populated data has the advantage of eliminating all stalls (CPU, disk etc.) in the DDL pipeline before the GPU. However, the training still suffers from GPU related stalls such as communication. Communication stalls can be categorized into two – (i) interconnect stall (intra-machine), and (ii) network stall (inter-machine). Below, we describe the methodology of determining the communication stalls using step ①, step ②, and step ⑤.

1) ***Interconnect (I/C) Stall****:* We define an interconnect stall as the *inter-GPU communication overhead of DDL in a single machine that arises due to the communication among the GPUs*. This is a key indicator of the performance of the underlying interconnect and is also indicative of the end-to-end training time which is determined using two steps:



(a) High level view of the Stash profiler

(b) I/C stall (M: Mini-batches, batch size = $\frac{n/4}{4}$)

Fig. 2: Our Stash scheme.

1) Stash pre-populates synthetic data in the memory of a single GPU only such that the number of samples the GPU processes is the same as that in a single GPU in a distributed training setup with multiple GPUs. Here, the batch size for multi-GPU training is kept the same as that of a single GPU. Stash then performs synthetic training on just a single GPU (in a multi-GPU machine) while keeping all other GPUs idle (see step ① in Figure 2(a)). Since this is a single-GPU training, no inter-GPU communication overhead is incurred.
2) Stash then runs distributed training, over all GPUs in the machine, on synthetic data. The number of samples each GPU processes and the per-GPU batch size is kept the same as in step ①.

Note that distributed training adds communication overhead to the end-to-end training time as a consequence of gradient synchronization. As a result, the difference in training time between ② and ① essentially yields the interconnect stall of the model for a particular machine. In essence, we are comparing the throughput of a single GPU with multiple GPUs by adjusting the batch size, but relieving the user of the manual effort of doing so.

We now describe an example of determining interconnect stalls using Figure 2(b). Suppose that, in a four GPU machine, the total DNN training dataset consists of n samples and the training must run over four mini-batches per epoch such that the batch size per GPU is set to be $\frac{n/4}{4}$. Therefore, as part of ①, Stash will pre-populate only one GPU with $n/4$ samples and a training process will be launched for one epoch using that particular GPU, keeping the other GPUs idle. This single-GPU training epoch has no need for gradient synchronization and hence, does not suffer from any communication overhead. After step ①, Stash will pre-populate all other GPUs with $n/4$ samples each as part of step ②, and launch distributed training over n samples (i.e., a DDL epoch). These four training processes will suffer from communication overheads due to the all-reduce (gradient synchronization) step, as depicted. The difference between the elapsed time of training over n samples with 4 GPUs and training over $n/4$ samples with a single GPU is the "communication overhead" (indicated in figure), which is essentially the interconnect stall.

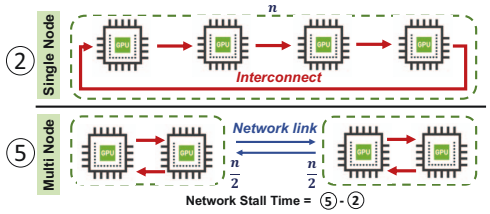


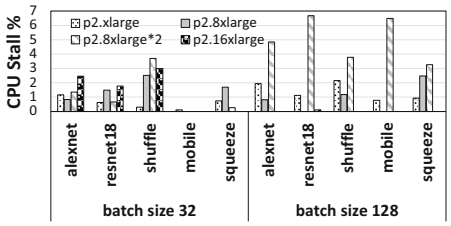
Fig. 3: Network stall calculation.

*2) **Network (N/W) Stall**:* We define a network stall as the *inter-GPU communication overhead of DNN training over multiple machines that arises due to the network link(s) between them*. This type of stall occurs when DDL is performed across multiple machines linked through a network. Since the all-reduce step requires gradients to be sent via both the intra-machine interconnect network as well as the inter-machine computer network, the slowest link becomes a communication bottleneck. Whenever the network link is the slowest link (compared to intra-node interconnect), network stalls occur. Stash determines network stalls as follows. Synthetic distributed training is run over multiple machines connected via network such that the total number of GPUs is the same as in the single machine training of ②. This is step ⑤ in Figure 2. The difference between the training times of ② and ⑤ yields the *network stall* of the model.

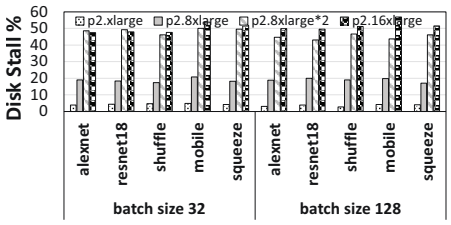
Again, Figure 3 depicts an example of determining network stall. Suppose we run step ② in an instance with 4 GPUs over n data samples, as shown in the figure. As part of ⑤, Stash now runs training over 2 instances with 2 GPUs each but with $n/2$ samples per machine keeping the per GPU batch size constant. When we train on 2 instances with a network link between them, the communication is bottlenecked by the network link if the link is slower than the hardware interconnect (most cases). For most cases where the network link is slower than the hardware interconnect, the network stall is calculated as the time difference between ② and ⑤.

V. Macro Characterization

Our characterization aims to answer a fundamental question, i.e., **which instance type is the most cost-effective and/or**



(a) CPU stall % (CPUs are sufficient for pre-processing)



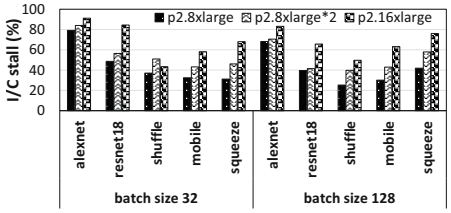
(b) Disk stall % (Scales with #GPUs per instance.)

Fig. 4: CPU and disk stall % of total training time in P2.

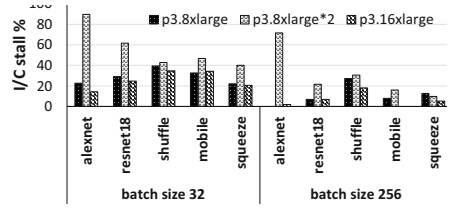
performant? To answer this, we realize that further questions need to be asked and hence, we begin our discussion by asking a simple but specific question: *How much stall does a DDL job experience from spending time on CPU, disk, interconnect, and network?* We investigate this problem by conducting a stall analysis on AWS P type instances with representative DDL workloads, while keeping the GPU as the first class resource. Although we use specific DNN models as example workloads, the techniques used herein can be generalized to all DDL workloads. We run our DDL workloads across four different mini-batch sizes (except BERT-large), with the largest batch size being (approximately) the maximum size that could fit in the GPU memory. For BERT, we only run on batch size 4, as that is the maximum size that allows the resultant data to fit in GPU memory (16 GB). Note that the batch sizes stated in the figures are per GPU and the effective batch size can be obtained as the per-GPU batch size times the number of GPUs. For brevity, we only show the plots of the profiling with the smallest and largest batch sizes used. The stall percentage is calculated as: $I/C\ \ stall\% = (\frac{I/C\ \ stall\ \ time}{single\ \ GPU\ \ time}) \times 100, \quad N/W\ \ stall\% = (\frac{N/W\ \ stall\ \ time}{single\ \ instance\ \ time}) \times 100$, where the I/C and N/W stalls are calculated as described in the previous section.

A. Analysis on AWS P2

AWS P2 instances use the NVIDIA K80 GPU with PCIe third generation interconnects. The P2 instances consist of three instance types – p2.xlarge, p2.8xlarge and p2.16xlarge as discussed in Section II. We profile P2 instances with small models across four mini-batch sizes – 32, 64, 96 and 128. Since K80 GPUs have limited compute and memory resources, they are not very suitable for running large models, i.e. models with a high parameter count. In practice, we observed very high I/C stall and monetary cost of training large models on P2. For e.g., for ResNet50, interconnect stall was observed to be 750% and monetary cost was $41 to train for a single epoch

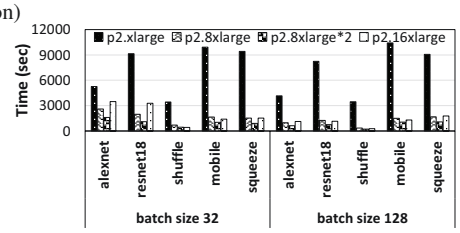


(a) P2 (K80 GPU)

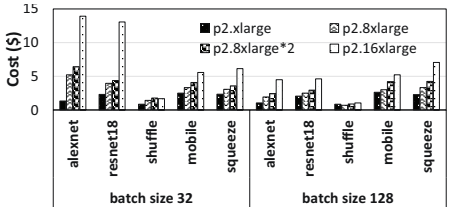


(b) P3 (V100 GPU)

Fig. 5: I/C stall small models (p2.16xlarge has the worst stalls due to PCIe contention, p3.8xlarge suffers from sub-optimal interconnect allocation)



(a) Training time (sec)



(b) Training cost ($)

Fig. 6: Training Time and Cost for P2, Small Models. (16xlarge is the least cost-optimal)

(the latter being 2000% more than P3). Hence, we employ the smaller models to characterize stalls on the K80 GPUs.

*1) **Stall Analysis**:* Figure 4 shows the CPU and disk stalls as a percentage of the total training time for mini-batch sizes 32 and 128. Unlike [31], we notice negligible CPU stalls in AWS, pointing to the fact that vCPUs at AWS are sufficient for most pre-processing needs of DL jobs. We further notice the largest amount of disk stalls for the 16x type machine. This is because there are 16 data loading workers running on the 16x machine to exploit the 16 GPUs of the machine. The 16 workers read from the attached SSD in parallel and create an I/O contention. The AWS general purpose SSD used in our experiments is unable to keep up with this demand and the training spends a significant amount of time performing disk I/O (only when data is not cached in DRAM).

We now discuss the interconnect and network stalls of P2 instances. We observe from Figure 6(a) that the 16x large runs a slower training than two 8xlarge machines that are network-

connected. This is true in all our batch runs. Furthermore, we observe from Figure 5 that 16xlarge has a higher interconnect stall time than both 8xlarge and 8xlarge*2 (two 8xlarge). This begs the question, *what is causing the slowdown in the 16xlarge?*

The slowdown in 16xlarge can be attributed to the limited bandwidth of the PCIe buses of P2 instances used for communication. In case of the 16xlarge, the PCIe bandwidth is shared among 16 workers causing congestion and "slicing" of the limited PCIe bandwidth. We validate this claim by measuring the PCIe bandwidth available per GPU using CUDA in xlarge, 8xlarge and 16xlarge instances. All GPUs are used in parallel when running the bandwidth test and we report the per device bandwidth in Figure 7. The GPUs in 16xlarge instance receive significantly less bandwidth than the GPUs of all other P2 instance types. This bandwidth is lower than the expected network bandwidth and hence the training gets throttled on the interconnect link, rather than on the network. As the network is not the slowest link and the 8xlarge instance has access to higher interconnect bandwidth than the 16xlarge, the 8xlarge*2 configuration performs better than the 16xlarge. This gives us an intuition that the 16xlarge instance is the least cost-optimal and we test this empirically by observing the monetary cost of running the workloads in Figure 6(b).

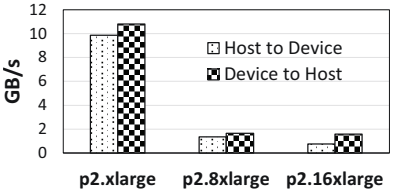
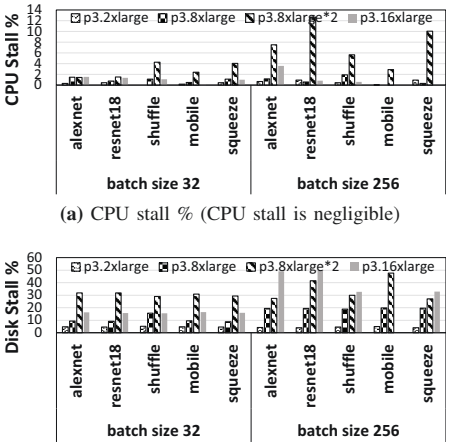


Fig. 7: Per GPU PCIe bandwidth measured in P2

A linear increase in cost is observed as the size of the P2 instance is increased. This confirms the intuition from our study of interconnect stalls that the monetary cost of executing a DDL workload is proportional to the observed interconnect stall of that workload. The lowest cost of running the training is on p2.xlarge, which has a single GPU and hence, has no interconnect stalls. However, the DDL execution time does not always *decrease* linearly from smaller instance to larger instance. From Figure 6(a), we notice that there is no significant improvement in training time on 16xlarge for a $2\times$ increase in cost. In fact, we notice in most cases that the running time in 16xlarge is more than that of 8xlarge, even with the instance having twice the resources as that of 8xlarge. This is because although resources like CPU, GPU and memory are doubled, the PCIe bus bandwidth remains the same (as already demonstrated), thereby causing congestion and significant slowdowns.

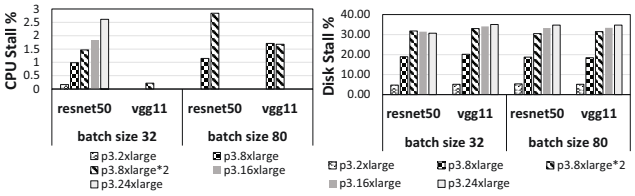
2) ***Recommendation****:* We observe both high interconnect and disk stalls on the 16xlarge instance and accordingly, believe the 16xlarge instance may not be the cost-optimal choice. Even when more GPUs are needed than what the 8xlarge instance can provide, training time and cost are lower when using a combination of 8xlarge instances connected via network compared to using the 16xlarge instance.



(a) CPU stall % (CPU stall is negligible)

(b) Disk stall % (Disk stall highest for 16xlarge)

Fig. 8: CPU and disk stall for P3, small models.



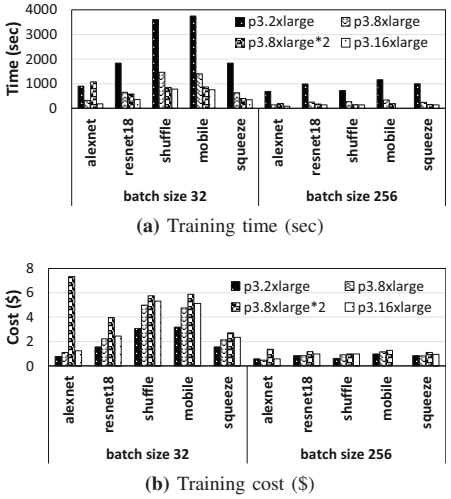
(a) CPU stall % **(b)** Disk stall %

Fig. 9: CPU and disk stall for P3, Large image models. (CPU stall is negligible, disk stall high for experiments with 8 GPUs)

B. Analysis on AWS P3

AWS P3 instances use the NVIDIA V100 GPU with NVLink interconnect as already described in Section II. The P3 instances are high-performing instances capable of training large DNN models in a cost-effective manner. We begin our discussion with the stall analysis of P3 in the sequel.

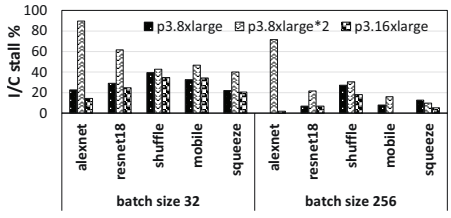
1) ***Stall Analysis****:* We show CPU and disk stalls for small models in Figure 8 and large models in Figure 9. The CPU



(a) Training time (sec)

(b) Training cost ($)

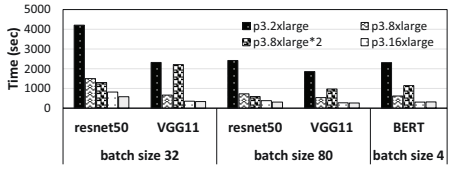
Fig. 10: Training Time and Cost for P3, Small Models. (16xlarge is the most performant)



(a) Small

(b) Large

Fig. 11: I/C stall for P3. (16xlarge has the lowest stall)



(a) Training time (sec)

(b) Training cost ($)

Fig. 12: Training Time and Cost for P3, Large Models. (16xlarge and 24xlarge are equally performant)

and disk stalls follow the same pattern as in P2. The CPU stalls are negligible and the disk stall is high for the 16xlarge instance. Unlike p2.16xlarge, the p3.16xlarge has eight GPUs and hence, eight workers perform I/O on the attached SSD. However, the throughput of training is also high due to the higher compute capacity (of both GPU and CPU) of the instance, thus, leading to higher usage of the SSD and higher disk stalls (data is not cached in DRAM).

The P3 instances use NVLink for communication between the GPUs instead of the PCIe bus. As discussed in Section II, NVLink offers significantly higher bandwidth compared to traditional PCIe-based communication and hence, we expect lower interconnect stalls while using NVLink. We measure and show the actual interconnect stalls for P3 in Figure 11 and notice that they are lower than those of the P2 instances, as expected. However, we also observe the 8xlarge (which has half the number of GPUs as the 16xlarge) to have higher overall interconnect stalls than the 16xlarge, especially for smaller models or while using smaller batch sizes. As the number of GPUs decreases, the volume of gradients to be transferred (as each GPU generates gradients) also decreases, thereby, requiring lesser bandwidth from the underlying interconnect. This should ideally translate into lower interconnect stalls for the 8xlarge. Therefore, we ask the question: *why does the p3.8xlarge instance not have strictly lower interconnect stalls than the 16xlarge?*

The reason for this anomaly is that although AWS provides a highly connected crossbar architecture (refer Figure 1) for communication via the NVLink, this may not be the case for the 8xlarge. Ideally, AWS should split the 16xlarge instance into two 8xlarge instances such that each instance gets an entire crossbar as shown by the dotted line in Figure 1. This would have provided the tenant/user with a highly-connected, high bandwidth GPU interconnect, resulting in lower interconnect stalls. However, we theorize that AWS is not able to ”evenly slice” the physical interconnect so as to give an entire crossbar to the 8xlarge instance. This may be due to multiple single size GPU requests from several tenants occupying GPUs in a crossbar. The 8xlarge instance loses the benefit of the crossbar architecture due to this and ends up being less performant with respect to interconnect stalls. This ”trait” of AWS interconnects is essentially probabilistic in nature and a tenant may indeed end up getting an entire crossbar for their 8xlarge instance, thereby, resulting in lower interconnect stalls.

Next, we compare the performance of p3.16xlarge with that of the p3.24xlarge. The p3.24xlarge is a dedicated instance offering which has the same number and type of GPUs as the 16xlarge but with twice the memory. It also comes with a dedicated local SSD storage along with more vCPUs and DRAM than the 16xlarge (refer Table I). However, from our stall analysis of the 24xlarge, we do not observe any significant decrease in stalls or training time compared to the 16xlarge. This is true even for our BERT large model which is both compute and memory–intensive. We now ask: *why is the performance of 24xlarge not strictly better than the 16xlarge?* The answer to this question, again, lies in its GPU interconnect. From [19] we know that both the 16xlarge and the 24xlarge use the same NVlink interconnect hardware and hence they also suffer from the same types of interconnect stalls. Although the 24xlarge offers a better configuration for each of its hardware components (GPU, DRAM, CPU, SSD etc.), it misses out on improving its NVLink interconnect. The DNN pipeline suffers from the same amount of communication overhead as the 16xlarge and hence, is not able to exploit the better hardware. This further lends credence to the importance of communication overhead in DDL (missed by prior work).

However, there is a caveat to this. The 24xlarge instance has twice the amount of per-GPU memory (32GB) than the 16xlarge. This allows users to run training with larger batch sizes thereby reducing time per epoch. However, we can't conduct a cost analysis between 16xlarge and 24xlarge by increasing the batch size of training on 24xlarge due to two reasons: (i) single epoch with different batch sizes is not representative of the same end-to-end training, and (ii) large batch sizes tend to converge to sharp minimizers which leads

to poor generalization [18]. But for comprehensiveness, we run our BERT model on the 24xlarge after doubling the batch size to 8. This resulted in about 12.8% improvement in training time and costing about $2.37. This is more than the $2.1 cost of running the model on 16xlarge with half the batch size.

Finally, we ask: *what happens when the instances are connected via the network?* To answer this question, we calculate the network stall of two p3.8xlarge instances connected via the network (p3.8xlarge*2) in Figure 13 as part of step ⑤ of Stash and notice network stalls as high as 500%. This is because as soon as the "all-reduce" ring contains a network link, the training gets throttled on this slow network link. Compared to the NVLink interconnect, which has a sufficiently large bandwidth to accommodate fast data transfers, the network link introduces higher slowdowns. This discourages us to run training over network links.

Note that we do observe large models like VGG to have low interconnect stall (but high network stall). The reasoning for this will be discussed in Section VI-A.

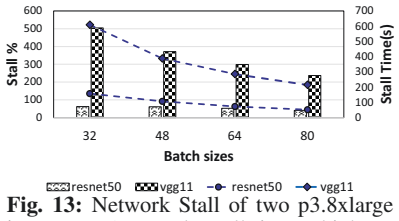
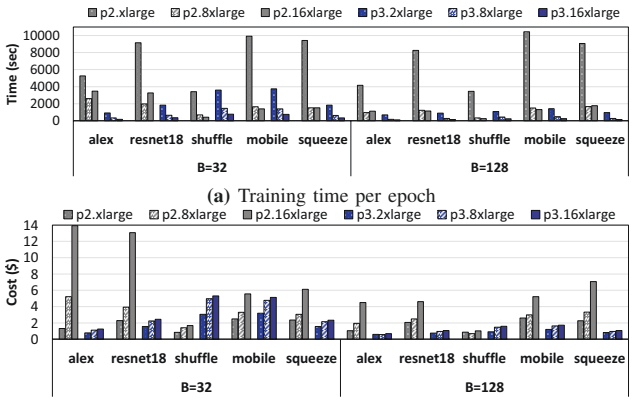


Fig. 13: Network Stall of two p3.8xlarge instances. (Network stall is as high as 500%)

*2) **Cost Analysis**:* We show the cost and time analysis of P3 instances in Figures 10 and 12. The cost analysis follows the same pattern as that in P2 instances but the performance of the instances differs. We find that the smallest P3 instance, the 2xlarge is the most cost optimal followed by the 8xlarge and the 16xlarge. The 24xlarge is the least cost-optimal in most experiments. An immediate question that can be asked here is: *how is 8xlarge more cost optimal than both 16xlarge and the 24xlarge?*

The answer to this question is that although 16x/24xlarge instances have lower interconnect stalls than the 8xlarge, they still suffer from higher disk stalls (due to more number of workers, refer Figures 8(b) and 9(b)) and hence, end up being less cost-effective than the 8xlarge. Note that the actual disk stall suffered is not as high as shown in the disk stall analysis due to caching of data. The disk stall is only high enough to compensate for the small interconnect stall difference between 8xlarge and the 16xlarge. It is mostly the interconnect stall that drives the cost-effectiveness of an instance. We also observe that the network connected instances are the least cost optimal due to high network stalls.

*3) **Recommendation**:* We recommend the single 2xlarge as the most cost-effective instance for training. However, we realize that using a single GPU is not practical to train most models due to time constraints. Hence, tenants must specifically find out the stalls for their models before running an end-to-end training on an 8xlarge or a 16xlarge. Fortunately, Stash is designed to solve this very problem and tenants can use it to find out the various stalls in their model. We do not recommend the use of 24xlarge unless the model requires the high GPU memory offered.



(a) Training time per epoch

(b) Training cost per epoch

Fig. 14: P2 vs P3 train-time/cost comparison. (P3 is generally more cost-optimal except for very small models)

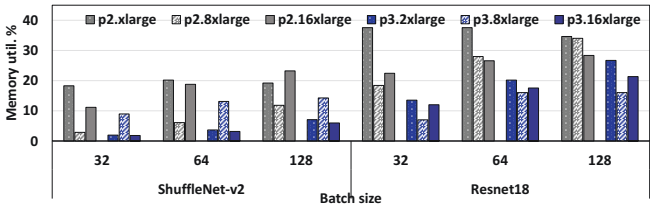


Fig. 15: GPU memory util. of P2 vs P3 for a two model. (Shufflenet has low GPU util. in P3)

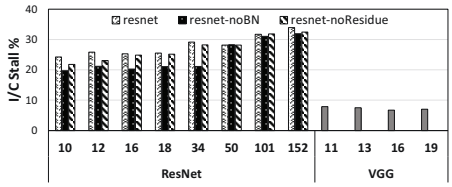
C. Comparison between P2 and P3

We now compare the two GPU instance types – P2 and P3 from a cost-efficiency perspective. From Figure 14, we notice that P3 instances are generally more cost-effective than P2 instances, although P3 instances are about 3.5× costlier per hour than P2 instances. This is because of the lower stalls P3 instances experience compared to their P2 counterparts. However, some smaller models like ShuffleNet are not able to exploit the memory and compute capacity of large V100 GPUs present in the P3 instances, unlike models with many layers like ResNet18 (shown in Figure 15). Hence, such small models incur the least cost when trained on P2 instances. Figure 14 shows the training time/ cost of running DDL on P2 and P3.

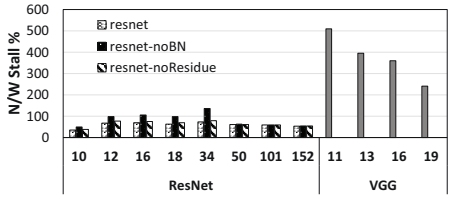
*1) **Recommendation**:* While we recommend using P3 instances whenever possible, we do notice that smaller models such as ShuffleNet can be trained cost-effectively on P2s. We also note that AWS has limited GPU availability and tenants might not always receive the desired number/type of GPUs from AWS. Thus, tenants may be forced to use P2 instances due to unavailability of P3s.

VI. Micro Characterization and Network Stall Analysis

In this section, we analyze the interconnect and network stalls through synthetic DNN models to (i) find characteristics in model architecture that influence interconnect and network stall behavior, and (ii) express the generality of our interconnect and network stall profiler for unseen models. We then discuss a cost comparison with Srifty.



(a) I/C Stall %



(b) N/W Stall %

Fig. 16: VGG has low I/C stall but high N/W stall while ResNet is vice versa

A. Micro Characterization

In order to verify the generality of our profiling technique for new models, we create synthetic models by altering the model architecture of popular DNNs (namely, ResNet and VGG), to highlight features that can affect stall behavior. Note that these changes are not meant to improve the DNN model accuracy or training/convergence time. Rather, they are meant to provide insights that can be leveraged by users to architect DNN models to improve system utilization. We run all experiments on a p3.16xlarge instance with a batch size of 32 per GPU, repeated thrice with the results averaged across the runs. A smaller batch size (32) is chosen so as to maximize all-reduce cycles. This, in turn, exacerbates communication stalls, thereby, facilitating the analysis of its underlying cause(s). We begin our discussion by asking two questions: (i) **Is there a relationship between the number of layers in a model and its communication stalls?**, and (ii) **How does the number of gradients to be transferred in a model affect communication stalls?**

*1) **Relationship between DNN layers, gradients and communication stalls**:* We answer the above questions by observing the communication stalls of ResNet and VGG with varying number of layers (fig 16). We observe that as the number of layers increases (accompanied by a commensurate increase in the number of gradients), both the interconnect stall and network stall time increases. This is expected, as there is more data to be transferred with the increase in number of gradients. However, despite the number of gradients in VGG being far more than that in ResNet (refer table II), *VGG is observed to have lower interconnect stall time than ResNet (Figure 16)* . Moreover, we also notice that *the network stall time of VGG is significantly more than ResNet (Figure 16).* These facts lead us to the next question that arises logically: **Why is the interconnect stall of VGG low and the network stall high when compared to those of ResNet?**

*2) **Explaining VGG and ResNet communication stalls**:* From [25], we know that distributed PyTorch overlaps communication and computation during the backward pass at individual layers. In case of ResNet, there is a large number of layers and relatively fewer gradients to transfer per layer. In comparison, VGG has fewer layers, but a larger number of gradients to transfer per layer. For instance, VGG16 consists of about 134.7 million trainable parameters while ResNet152 consists of only 58.5 million [22]. Therefore, the gradients to transfer per synchronization point is greater in VGG, but the number of times the gradients get transferred is higher in ResNet. We now explain how this characteristic leads to the interconnect stalls observed in the previous subsection.

Suppose VGG has G_{vgg} bytes of gradients and L_{vgg} layers, and ResNet has G_{res} bytes of gradients and L_{res} layers. Also, let us say that NVLink offers B_{nv} bandwidth with τ_{nv} latency. The time to transfer gradients comprises both latency and data transfer time. Define this for VGG and ResNet to respectively be T_{vgg} and T_{res}. Thus, the transfer time using NVlink is given by:

$$\mathrm{T}_{vgg} = \left[\tau_{nv} + \frac{G_{vgg}}{L_{vgg} \times B_{nv}}\right] \times L_{vgg}, T_{res} = \left[\tau_{nv} + \frac{G_{res}}{L_{res} \times B_{nv}}\right] \times L_{res}$$

Since NVLink offers very high bandwidth (more than 100 Gbps [23]), and also because both models have a large number of layers, we can assume: $\frac{G_{vgg}}{L_{vgg} \times B_{nv}} \ll \tau_{nv}$ and $\frac{G_{res}}{L_{res} \times B_{nv}} \ll \tau_{nv}$. Hence, data transfer time over NVLink is $T_{vgg} \approx \tau_{nv} \times L_{vgg}$ and $T_{res} \approx \tau_{nv} \times L_{res}$.

Thus, $\boxed{L_{res} > L_{vgg} \implies T_{res} > T_{vgg}}$

In other words, the training process experiences increased slowdown due to the larger number of layers to transfer in ResNet (or in any other deep model, for that matter). It follows that in the case of VGG, although the data to be transferred is much larger, the time to transfer is nearly zero due to the lower number of layers. The only slowdown we observe here is due to the transfer latency associated with the transfer link/framework.

Now, we explain the high network stall time observed for VGG in the previous subsection. As already explained in Section IV-B2, the collective all-reduce performed across the network-connected instances is throttled by the network link. Hence, we can assume that the data transfer time is a function of the network link only. Suppose the network link offers B_{nw} bandwidth with τ_{nw} latency. Similarly, the data transfer time over network is:

$$\mathrm{T}_{vgg} = \left[\tau_{nw} + \frac{G_{vgg}}{L_{vgg} \times B_{nw}}\right] \times L_{vgg}, T_{res} = \left[\tau_{nw} + \frac{G_{res}}{L_{res} \times B_{nw}}\right] \times L_{res}$$

Since the network link is slow, we can assume: $\tau_{nw} \ll \frac{G_{vgg}}{B_{nw}}$ and $\tau_{nw} \ll \frac{G_{res}}{B_{nw}}$. Hence, the data transfer time over network link is: $T_{vgg} \approx \frac{G_{vgg}}{B_{nw}}$ and $T_{res} \approx \frac{G_{res}}{B_{nw}}$.

Thus, $\boxed{G_{vgg} > G_{res} \implies T_{vgg} > T_{res}}$

In other words, since the network link is slow, the data transfer is throttled on the transfer time rather than the latency. Since a much larger volume of gradients needs to

be transferred in VGG (in total), the network stall is much higher in VGG than in ResNet.

3) ***Impact of model architecture****:* To probe further into the specific aspects of DNN model architecture that impact interconnect stalls, we made two changes to the ResNet model by removing batch normalization (BN) as well as residual networks. These changes are intended to show the extent to which these layer types impact communication stalls. From figure 16 we notice that removing residual networks has minimal impact on communication overhead. This is because residual networks do not introduce any new layers and hence do not impact communication. However, removing BN reduces the number of layers in the model and hence we see lowering in communication stalls as shown in the figure 16.

4) ***Recommendation****:* From our experiments, we observe that models with very deep networks and fewer gradients are unable to fully exploit the fast NVLink interconnect, whereas shallower networks with large gradient transfers can be throttled on the network link. Hence, we recommend running shallow networks with large gradients on instances with the best interconnect possible. However, if the model is very deep with fewer gradients per layer, the models can be run on instances without the best interconnect, such as the p3.8xlarge. The penalty for running such models on network-connected instances is also minimized.

B. Discussion: Comparison with Srifty

Srifty extensively measures network-throughput variance (for it to work) by performing a grid probe of communication by sweeping different buffer sizes, world size, instance types and location. This results in 40K unique measurements [28]. Hence, to use srifty one needs to run these probes again if: (i) they do not gain access to the original measurements, (ii) the network changes (public cloud network is susceptible to infra or tenant changes), or (iii) the target location is not measured by srifty. Moreover, the user is expected to setup new VMs to take the measurements, which involves cold-start delays, along with the effort of setting up large clusters (up to 64 VMs required). Such significant added cost of achieving an automated recommendation system should be accounted against srifty's performance. Note that Stash comes at no such costs to the users.

VII. Related Work

Characterizing Deep Learning. Existing works in the area of DL characterization [2], [14], [17], [27], [31], [32], [48]–[50] do not conduct a stall analysis on *public* cloud GPU instances, which is what we do here.

Cost optimization in the public cloud. Prior works such as [42], [47], [52] explicitly focus on reducing costs of migrating and running "generic" workloads on public cloud (including DNN inference), i.e., not specifically DL. While some prior works [4], [5] may focus on resource management, they make implicit monetary decisions by selecting the serverless platform, which may not be cost-optimal.

VIII. Concluding Remarks

We introduced a DDL stall profiler Stash and presented novel methodologies to measure communication stalls in particular. Using Stash, we extensively characterized public cloud GPU instances for the various stalls they experience when training popular DNN models. We found communication stalls to be the major bottleneck in DDL training and that some AWS instances are heavily impacted by them. The observed interconnect stalls and network stalls were up to 90% of single GPU and 500% of single instance training time respectively due to severe bandwidth contention when using the PCIe bus, sub-optimal resource allocation when using the NVLink and low AWS network bandwidth. Note that these high stalls translate to higher training costs. Further, we analyzed the impact of DNN model architecture features on communication stalls. Finally, we discuss the true cost of cloud recommenders. Stash is open-sourced at [46] with a technical report available at [41] along with additional results from those shown here.

Acknowledgements

We are indebted to our anonymous reviewers for their insightful comments. This research was partially supported by NSF grants #1931531, #1955815, #1763681, #2116962, #2122155, #2028929 and a grant from Adobe. We also thank Chameleon Cloud project CH-819640 for their compute grant. All product names used here are for identification purposes only and may be trademarks of their respective companies.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-scale Machine Learning. In *Proc. USENIX OSDI*, 2016.

[2] Ammar Ahmad Awan, Jereon Bédorf, Ching-Hsiang Chu, Hari Subramoni, and Dhabaleswar K Panda. Scalable distributed DNN training using TensorFlow and CUDA-aware MPI: Characterization, designs, and performance evaluation. In *Proc. IEEE/ACM CCGRID*, 2019.

[3] AWS NVIDIA GPU instances. https://aws.amazon.com/nvidia/, Accessed: 2022.06.08.

[4] Vivek M. Bhasi, Jashwant Raj Gunasekaran, Aakash Sharma, Mahmut Taylan Kandemir, and Chita Das. Cypress: Input size-sensitive container provisioning and request scheduling for serverless platforms. In *Proc.*, SoCC '22. ACM, 2022.

[5] Vivek M. Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. In *Proc.*, SoCC '21. ACM, 2021.

[6] Alibaba PAI. https://github.com/AlibabaPAI, Accessed: 2022.06.15.

[7] DawnBench. https://dawn.cs.stanford.edu/benchmark/, Accessed: 2022-06-08.

[8] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large scale distributed deep networks. In *Proc. Advances in Neural Information Processing Systems*, volume 25, 2012.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proc. NAACL-HLT '19*. ACM, 2019.

[10] K. Fukuda. Technologies behind Distributed Deep Learning: AllReduce. https://tech.preferred.jp/en/blog/technologies-behind-distributed-deep-learning-allreduce/, Accessed: 2022.06.08.

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

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision – ECCV 2016*, 2016.

[13] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Proc. NIPS*, 2013.

[14] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. Characterization and prediction of deep learning workloads in large-scale GPU datacenters. In *Proc. SC '21*.

[15] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. https://arxiv.org/abs/1602.07360, 2016.

[16] ImageNet Large Scale Visual Recognition Challenge 2012 (ILSVRC2012). https://www.image-net.org/challenges/LSVRC/2012/, Accessed: 2022.10.18.

[17] Arpan Jain, Ammar Ahmad Awan, Quentin Anthony, Hari Subramoni, and Dhableswar K DK Panda. Performance characterization of dnn training using tensorflow and pytorch on modern clusters. In *CLUSTER*. IEEE, 2019.

[18] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[19] Rashika Kheria, Purna Sanyal, Sr. James Jeun, and Amr Ragab. Optimizing deep learning on P3 and P3dn with EFA . https://aws.amazon.com/blogs/compute/optimizing-deep-learning-on-p3-and-p3dn-with-efa//, Accessed: 2022.06.08.

[20] Yunyong Ko, Kibong Choi, Jiwon Seo, and Sang-Wook Kim. An in-depth analysis of distributed training of deep neural networks. In *IPDPS*. IEEE, 2021.

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 25, 2012.

[22] Mei Leong, Dilip Prasad, Yong Tsui Lee, and Feng Lin. Semi-cnn architecture for effective spatio-temporal learning in action recognition. *Applied Sciences*, 10:557, 01 2020.

[23] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE TPDS*, 31(1):94–110, 2019.

[24] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Proc. USENIX OSDI*, 2014.

[25] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proc. VLDB Endow.*, 13(12):3005–3018, aug 2020.

[26] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. Asynchronous decentralized parallel stochastic gradient descent. In *ICML '18*, pages 3043–3052. PMLR, 2018.

[27] Jie Liu, Jiawen Liu, Wan Du, and Dong Li. Performance analysis and characterization of training deep learning models on mobile device. In *ICPADS*, pages 506–515. IEEE, 2019.

[28] Liang Luo, Peter West, Pratyush Patel, Arvind Krishnamurthy, and Luis Ceze. Srifty: Swift and thrifty distributed neural network training on the cloud. *MLSys*, 4:833–847, 2022.

[29] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *ECCV*, 2018.

[30] Peter Mattson, Christine Cheng, Gregory Diamos, Cody Coleman, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. Mlperf training benchmark. *MLSys*, 2:336–349, 2020.

[31] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in dnn training. https://arxiv.org/abs/2007.06775, 2021.

[32] Saiful A Mojumder, Marcia S Louis, Yifan Sun, Amir Kavyan Ziabari, José L Abellán, John Kim, David Kaeli, and Ajay Joshi. Profiling DNN workloads on a Volta-based DGX-1 system. In *Proc. IISWC*, pages 122–133. IEEE, 2018.

[33] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proc. SOSP*, 2019.

[34] Nsight. https://developer.nvidia.com/nsight-systems, Accessed: 2023.

[35] NVIDIA Deep Learning Examples for Tensor Cores . https://github.com/NVIDIA/DeepLearningExamples, Accessed: 2022.06.08.

[36] Nvprof. https://developer.nvidia.com/blog/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/, Accessed: 2023.

[37] N.J. Parmar, A. Vaswani, J. Uszkoreit, L. Kaiser, N. Shazeer, A. Ku, and D. Tran. Image Transformer. In *Proc. ICML '18*, 2018.

[38] Rolf Rabenseifner. Optimization of collective reduction operations. In *ICCS*. Springer, 2004.

[39] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text, 2016.

[40] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proc. CVPR '18*, 2018.

[41] Aakash Sharma, Vivek M Bhasi, Sonali Singh, Rishabh Jain, Jashwant Raj Gunasekaran, Subrata Mitra, Mahmut Taylan Kandemir, George Kesidis, and Chita R Das. Analysis of distributed deep learning in the cloud. *arXiv preprint arXiv:2208.14344*, 2022.

[42] Aakash Sharma, Saravanan Dhakshinamurthy, George Kesidis, and Chita R Das. CASH: A Credit Aware Scheduling for Public Cloud Platforms. In *Proc. IEEE/ACM CCGrid*, 2021.

[43] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *Proc. ICLR*, San Diego, CA, May 7-9, 2015.

[44] Sonali Singh, Anup Sarma, Nicholas Jao, Ashutosh Pattnaik, Sen Lu, Kezhou Yang, Abhronil Sengupta, Vijaykrishnan Narayanan, and Chita R. Das. Nebula: A neuromorphic spin-based ultra-low power architecture for snns and anns. In *Proc. 47th ACM/IEEE ISCA*, 2020.

[45] Sonali Singh, Anup Sarma, Sen Lu, Abhronil Sengupta, Mahmut T. Kandemir, Emre Neftci, Vijaykrishnan Narayanan, and Chita R. Das. Skipper: Enabling efficient snn training through activation-checkpointing and time-skipping. In *Proc. 55th IEEE/ACM MICRO*, 2022.

[46] Stash. https://github.com/aakash-sharma/Stash, Accessed: 2023.06.5.

[47] C. Wang, B. Urgaonkar, N. Nasiriani, and G. Kesidis. Using Burstable Instances in the Public Cloud: When and How? In *Proc. ACM SIGMETRICS, Urbana-Champaign, IL*, June 2017.

[48] Mengdi Wang, Chen Meng, Guoping Long, Chuan Wu, Jun Yang, Wei Lin, and Yangqing Jia. Characterizing deep learning training workloads on Alibaba-PAI. In *Proc. IEEE IISWC*, 2019.

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

[50] Chunwei Xia, Jiacheng Zhao, Huimin Cui, and Xiaobing Feng. Characterizing dnn models for edge-cloud computing. In *IEEE IISWC*, 2018.

[51] Jun Yi, Chengliang Zhang, Wei Wang, Cheng Li, and Feng Yan. Not all explorations are equal: Harnessing heterogeneous profiling cost for efficient mlaas training. In *IPDPS*. IEEE, 2020.

[52] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *Proc. USENIX ATC*, Renton, WA, 2019.

[53] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *Proc. USENIX ATC*, 2017.

[54] Sixin Zhang, Anna E Choromanska, and Yann LeCun. Deep learning with elastic averaging SGD. *Proc. Advances in neural information processing systems*, 28, 2015.

[55] Hongyu Zhu, Amar Phanishayee, and Gennady Pekhimenko. Daydream: Accurately estimating the efficacy of optimizations for dnn training. In *USENIX ATC 20*, 2020.