Efficient Strong Scaling Through Burst Parallel Training

Seo Jin Park [1]   Joshua Fried [1]   Sunghyun Kim [1]   Mohammad Alizadeh [1]   Adam Belay [1]

Abstract

As emerging deep neural network (DNN) models continue to grow in size, using large GPU clusters to train DNNs is becoming an essential requirement to achieving acceptable training times. In this paper, we consider the case where future increases in cluster size will cause the global batch size that can be used to train models to reach a fundamental limit: beyond a certain point, larger global batch sizes cause sample efficiency to degrade, increasing overall time to accuracy. As a result, to achieve further improvements in training performance, we must instead consider "strong scaling" strategies that hold the global batch size constant and allocate smaller batches to each GPU. Unfortunately, this makes it significantly more difficult to use cluster resources efficiently. We present DeepPool, a system that addresses this efficiency challenge through two key ideas. First, *burst parallelism* allocates large numbers of GPUs to foreground jobs *in bursts* to exploit the unevenness in parallelism across layers. Second, *GPU multiplexing* prioritizes throughput for foreground training jobs, while packing in background training jobs to reclaim underutilized GPU resources, thereby improving cluster-wide utilization. Together, these two ideas enable DeepPool to deliver a $1.2 - 2.3\times$ improvement in total cluster throughput over standard data parallelism with a single task when the cluster scale is large.

1 Introduction

The size of deep learning models has been growing rapidly. For example, Microsoft announced that it successfully trained a model for machine translation with 17 billion parameters (Rosset, 2020). To train such large models quickly, it is necessary to use increasingly large training clusters. Current, state-of-the-art training systems already make use of hundreds of GPUs (Goyal et al., 2017; Shoeybi et al., 2019; Kumar et al., 2021).

One of the most difficult challenges these systems face is maintaining efficiency while scaling. The conventional wisdom is to use data parallelism to distribute work across GPUs and to increase global batch sizes proportionally with the size of the cluster. This keeps the iteration time relatively constant, but allows for more samples to be processed per iteration, increasing training throughput with cluster size.

Unfortunately, a higher sample processing throughput does not necessarily translate to a faster time to accuracy: beyond a certain point, training algorithms become less effective with larger batch sizes because they experience a loss in sample efficiency (Shallue et al., 2018; Qiao et al., 2021). Ultimately, this places a fundamental limit on the maximum global batch size that can be beneficial. Therefore, as the

number of GPUs used in training continues to increase, each GPU will inevitably be forced to process smaller batches at a time.

In this paper, we investigate how to balance cluster utilization with time to accuracy in the regime where batch sizes are limited by sample efficiency. The key difficulty is that smaller per-GPU batch sizes expose the uneven parallelism inherent to many DNN models: some layers have sufficient parallelism to keep GPUs busy, while others leave GPUs significantly underutilized. As a result, a conventional data parallel approach is insufficient to make efficient use of GPU hardware.

To address this challenge, we propose **DeepPool**, a system that realizes two key ideas. First, we introduce *burst parallel training* to dynamically adjust the number of GPUs allocated to each layer, so that layers with less parallelism can use fewer GPUs. This increases overall cluster efficiency because it frees up underutilized GPUs for use by other training tasks. Second, we propose a new collection of *GPU multiplexing* techniques that allow multiple models to be trained on each GPU simultaneously. Intuitively, this allows us to further improve efficiency because GPUs have massive internal parallelism that is underutilized by smaller per-GPU batches (SMs, tensor cores, memory bandwidth, etc.). Thus, collocating training tasks can yield better GPU throughput overall.

Throughout this paper, we distinguish between foreground training tasks, such as the training of production models, and

---

background training tasks, like testing new model designs at a smaller scale. DeepPool tailors both of its mechanisms to minimize time to accuracy for foreground tasks, while improving overall cluster throughput by packing background tasks on to the same GPUs.

Our evaluation shows that DeepPool can achieve 1.2–2.3x improvements in total cluster throughput over standard data parallelism with a single task. When compared to statically partitioning a cluster into two groups (data-parallel foreground task training and individual background task training), DeepPool allows 11–38% higher foreground speedup for the same high level of cluster throughput.

DeepPool has some limitations. First, it focuses only on dynamic scaling on sample dimensions, but a more general system would explore more flexible forms of model parallelism. Second, it currently limits background jobs to a single GPU, but our burst parallel planner could be extended in the future to scale background jobs across multiple GPUs. Finally, we discovered that today's GPU hardware has shortcomings that limit its multiplexing efficiency; we discuss our workarounds for this problem in §5.

2 CHALLENGES FOR SCALING DNN TRAINING EFFICIENTLY

There is significant interest in speeding up DNN training on large GPU clusters (Dean et al., 2012; Abadi et al., 2016; Jia et al., 2018; Dryden et al., 2019; Ben-Nun & Hoefler, 2019; Kumar et al., 2021). A primary metric for many organizations is the *time to accuracy*, i.e. how long it takes to train a model to a desired level of accuracy. The ML design process is often iterative, with researchers repeatedly training models and refining the architecture, training data, or hyperparameters based on outcomes. Improving time to accuracy for such jobs, which we will henceforth refer to as "foreground" jobs, can therefore significantly impact researcher productivity.

Unfortunately, parallelizing a DNN training job on a large scale often makes inefficient use of GPU hardware. The conventional approach is to use *weak scaling*: as we scale a job to a large number of GPUs, we increase the global batch size correspondingly, such that the per-GPU batch size is kept constant. Weak scaling improves training *throughput* (the number of data samples processed per second) linearly with cluster size. However, it usually does not result in a linear reduction in time to accuracy. The reason is that increasing the global batch size beyond a certain point can degrade the sample efficiency of the learning algorithm (i.e., we require more data samples to reach the same level of accuracy). This drop in sample efficiency results in a diminishing return in terms of wall-clock training time (Shallue et al., 2018). At large scale, such as 128 GPUs or greater,
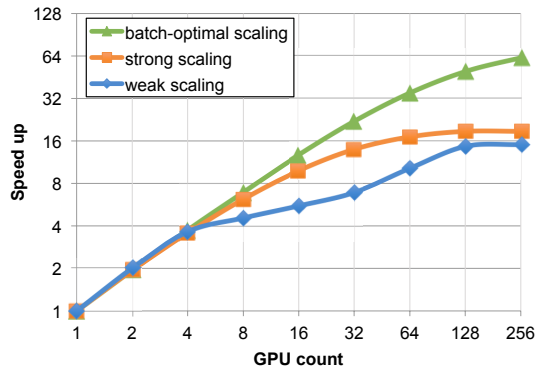

Figure 1. The estimated speedups for training VGG-11 to error = 0.35 with different scaling strategies. Each GPU has 1 Tbps full bi-section networking. Weak scaling uses 256 samples per GPU at each iteration. Strong scaling splits 256 samples across GPUs.

the training time may entirely cease to improve with the addition of more GPUs.

An alternative approach for accelerating DNN training is *strong scaling*: given more GPUs, keep the global batch size fixed, but reduce the amount of computation done by each GPU. For example, in data-parallel training, each GPU processes a smaller batch of samples in each iteration as we scale to more nodes. Unlike weak scaling, this approach does not degrade sample efficiency as it does not change the global batch size. However, it introduces two forms of inefficiency that again prevent a linear speedup with cluster size. First, since strong scaling reduces the amount of computation on each GPU in an iteration, the training job eventually gets bottlenecked by communication bandwidth between GPUs (e.g., for parameter synchronization in data-parallel training). Second, GPUs have massive internal parallelism, which is hard to utilize fully when the GPU's workload becomes too small (e.g., small batches or operations).

An obvious improvement to these two extremes is to optimize the global batch size for each scale. This approach, which we will refer to as "batch-optimal scaling", seeks to find the sweet spot in training throughput and sample efficiency that provides the best time to accuracy.

Figure 1 compares the estimated speedup achieved by the above three approaches for training the VGG-11 model. To obtain these estimates, we refer to the study of Shallue et al. (Shallue et al., 2018) to determine the number of iterations required to train the model to desired accuracy at different global batch sizes. Next, we profile the computation time for different layers of VGG-11 on an NVIDIA A100 GPU at different batch sizes and use a simple network model to estimate the communication time for data-parallel training (see "modeling communication cost" in §4 for details), which together give an estimate of the time per iteration for each approach. As the figure shows, all approaches
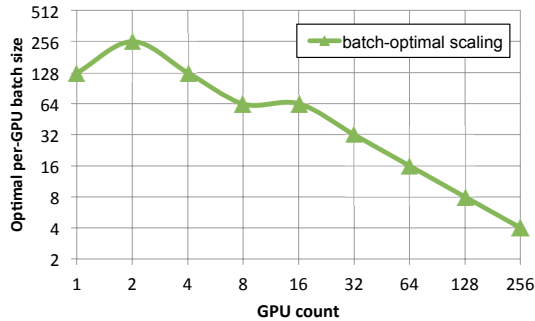
Figure 2. The per-GPU batch size for training VGG-11 to error = 0.35 chosen by the batch-optimal scaling strategy at each scale. Each GPU (NVIDIA A100) has 4.8 Tbps bi-directional networking. As the job scales, the per-GPU batch size tends to decrease, implying a lower per-GPU compute load.
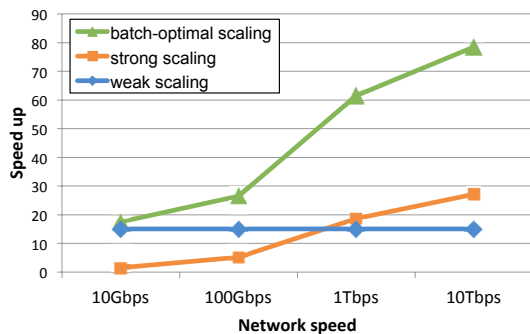

Figure 3. The estimated speedups by using 256 GPUs for training VGG-11 to error = 0.35 at four different network speeds. Weak scaling uses 256 samples per GPU, strong scaling splits 256 samples across GPUs. The benefits of using higher network speeds are clear in the strategies that employ strong scaling.

provide linear speedup up to 4 GPUs. Beyond this point, the marginal benefits of adding more GPUs diminish for weak scaling while strong and batch-optimal scaling provide better speedups.

This simple experiment reveals several interesting findings.

1. Optimizing time-to-accuracy requires small per-GPU batches at large scale. Figure 2 shows the per-GPU batch size chosen by the batch-optimal scaling strategy at each scale. As we can see, at small scale, the best strategy is to use a large per-GPU batch size. However, as the job scales, the optimal per-GPU batch size gets smaller, implying that the amount of compute performed by a GPU in an iteration must decrease.

2. Strong scaling and small per-GPU batches are more effective with fast networks. Figure 3 compares the speedup of each approach at different network speeds at a scale of 256 GPUs. When the network is slow (e.g., 10 Gbps), weak scaling is preferrable since it amortizes the communica-
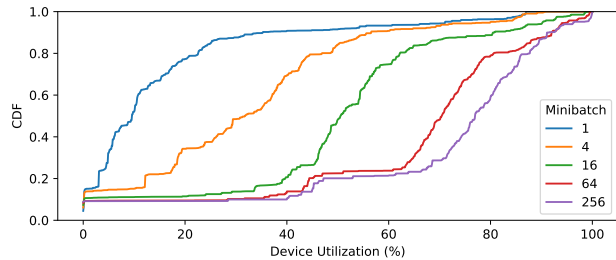

Figure 4. GPU utilization of Resnet-50.

tion cost of parameter synchronization over a larger amount of computation in each iteration. However, with today's cutting edge networks (e.g., 2 Tbps NVSwitch, 200 Gbps ConnectX-6) and even faster network technologies on the horizon (Khani et al., 2021), strong scaling will become more attractive.

3. None of the approaches achieve perfect linear scaling. As shown in Figure 1, even the batch-optimal approach still suffers from diminishing and sub-linear speedup at large scale. As mentioned earlier, this is due to both communication overheads and GPU underutilization as we scale the cluster. Figure 4 shows the device utilization achieved at different batch sizes. For small batch sizes, GPU utilization degrades, suggesting that even with very fast networks, we cannot achieve a linear speedup.

These findings suggest that operators of today's GPU clusters have to make an unfortunate choice. For an important foreground training job, they must either limit the scale, therefore completing the job slower than theoretically possible on the cluster. Alternatively, they can run the job at large scale and suffer low GPU utilization. However, since large GPU clusters are shared by multiple jobs, a natural question is: *is it possible to achieve speedups near the theoretical limits for foreground jobs while simultaneously achieving high GPU utilization by reclaiming unused cycles for less-time-critical background jobs?*

In this paper, we present a system, DeepPool, to achieve precisely this — strong scaling of foreground jobs with efficient GPU multiplexing between foreground and background jobs.

3 APPROACH AND OVERVIEW

3.1 Approach

Our main tool to achieve efficient strong scaling is collocating more than one job on a single GPU. Large GPU clusters are typically shared by many users and training jobs (e.g., within an institution), and there are many small training jobs that don't require large-scale training (usually for quick test-
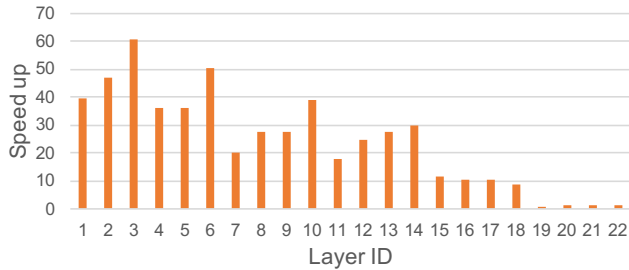
Figure 5. Heterogeneous scalability of layers in VGG16. Y-axis shows the speedup of each layer when it is strong scaled from 128 samples per iteration to 2 samples per iteration using 64 GPUs.

ing of new model architecture with a small dataset or when low cost matters more than speed). By scheduling those small training jobs as a low-priority job, we can reclaim the spare compute power of GPUs during strong scaling. GPU's limited memory is often an obstacle to scheduling multiple jobs on a GPU. Fortunately, strong scaling reduces memory footprints as well as compute load, reserving enough memory space for a small background job.

With the ability to reclaim underutilized GPUs with background jobs, we further improve efficiency with **burst parallel training**. One source of inefficiency in strong scaling is the uneven parallelism across stages within an iteration. As an example, Figure 5 shows the scalability of layers in VGG16; some layers can achieve near-linear speedup with more GPUs, but some other layers don't get any faster. By bursting the number of GPUs only for training stages that can benefit from more GPUs, we can drive up the efficiency further. When some GPUs are idling for the foreground job (for stages using a smaller number of GPUs), the background job can fully utilize the GPU. Alternatively, we may carefully place another large-scale distributed job during the idle gaps.

3.2 System overview

We built DeepPool, a prototype training system that achieves efficient strong scaling by implementing burst parallel training and GPU multiplexing.

A user may submit a training job to DeepPool by providing a PyTorch-like model implementation, dataset, and *inefficiency tolerance*. With the limit on scaling inefficiency, the **burst parallel training planner** decides the scaling of each layer so that it won't use too many GPUs and cause inefficiency beyond the allowance. For estimating speedup and inefficiency, the planner initially profiles each layer with different batch sizes. For the planner to work properly, DeepPool requires the input model's execution graph to be static.

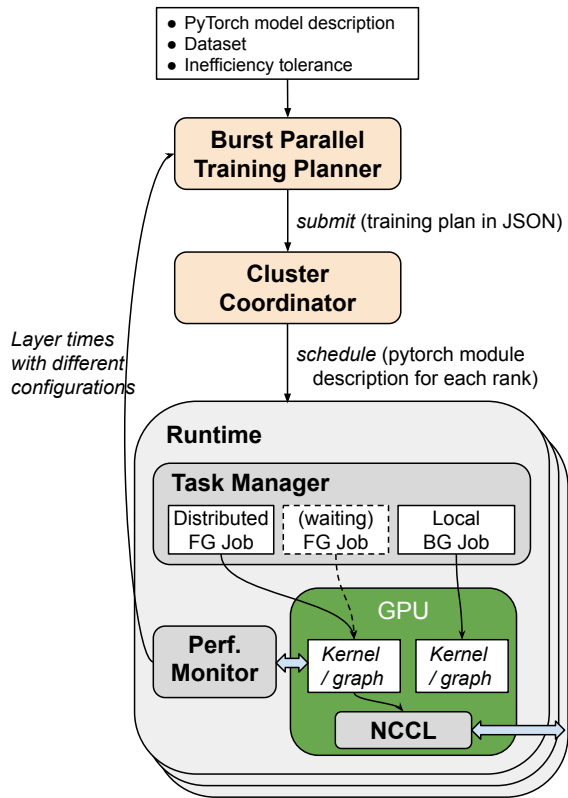The generated burst parallel training plan is then handed to

Figure 6. System architecture.

the cluster coordinator, which manages all GPU runtimes and places a new training job on a subset of GPUs, as specified in the training plan. Currently, the cluster coordinator doesn't support complex placement such as aligning gaps of one burst parallel job with another burst parallel job.

Each GPU in the cluster has a separate host-run DeepPool *runtime*. The *task manager* in each runtime manages all training jobs including the background job. At any moment, a task manager schedules one distributed foreground (burst parallel) job and one local low-priority background job, and only the foreground job communicates with other GPUs via NVIDIA Collective Communications Library (NCCL). The runtime uses the C++ frontend of PyTorch (LibTorch) as the execution engine. While running training jobs, the performance monitor module profiles the runtime of each layer, which may be fed back to the planner to optimize the training plan. In our prototype, this feedback loop happens manually. Additionally, the profiling data is used to bound any slowdowns that may be caused by poor collocation choices.

In the following sections, we discuss how the burst-parallel training planner optimizes the parallelization plan (§4) and how we minimize the interference between foreground jobs and background jobs (§5).

4 BURST PARALLEL SCHEDULING

DeepPool's burst parallel training planner takes as input the total number of available GPUs and the global batch size, which may be optimized by statistical efficiency analysis as seen in §2. Given the two parameters, the planner finds the best strong scaling strategy by choosing the level of scaling (the number of GPUs) for every layer in the model. It minimizes iteration time while limiting the scale of each stage up to a user-given efficiency limit.

Scaling efficiency is defined by *GPU-sec* amplification, where *GPU-sec* is the aggregate active GPU time per iteration, like man-hour or watt-hour. Amplification is defined as $\langle GPU\text{-}sec\ when\ scaled\rangle / \langle single\ GPU\ iteration\ time\rangle$. The planner considers the GPU-sec amplification of each layer individually.

4.1 Iteration time components

DeepPool's planner estimates iteration time by summing up each layer's computation and communication costs.

Computation cost. To compute the iteration time, the planner profiles the computation costs of each layer with every possible degree of scaling. We measure each layer's time to compute forward and backward passes with different per-GPU batch sizes, and the sum is used. In formal notation,

- $\text{comp}_{(i,g)}$: the sum of forward and backward compute time of i-th layer when scaled to g GPUs.

Communication cost. We use a simple networking model for estimating communication costs during training. We assume full bi-section networking (as in NVSwitch networking) and profile the networking performance in terms of per-GPU bandwidth and minimum propagation delay. In the planner, we simply divide the payload size by the bandwidth and add the propagation delay.

The planner must also account for the communication involved when scaling the number of GPUs up or down between layers. When this happens, samples must be transferred across GPUs before running the layer (as do gradients during backward passes). In formal notation,

- $\text{comm}_{(i,g)\rightarrow(j,h)}$: communication time for activation and back-propagation between i-th layer and j-th layer when scaled to g GPUs and h GPUs respectively.

Another communication overhead considered is gradient/parameter synchronization after a backward pass. For simplicity, we assume that this synchronization does not overlap with the backward pass. In formal notation,

- $\text{sync}_{(i,g)}$: time for synchronizing gradients of i-th layer when scaled to g GPUs.

Algorithm 1 Search Linear

Initialize $S[0][g] = 0$ for all $g \in [1, G]$
for $i = 1$ **to** L **do**
    for g **in** $[1, G]$ **do**
        Initialize bestAmp = ∞.
        Initialize bestS = ∞.
        Initialize bestT = ∞.
        for h **in** $[1, G]$ **do**
            if $Amp(i-1, h) \leq \max(\text{bestAmp, AmpLimit})$ **and**
            $S[i-1][h] + \text{comm}_{(i-1,h)\rightarrow(i,g)} \leq \text{bestS}$ **then**
                bestS $\leftarrow S[i-1][h] + \text{comm}_{(i-1,h)\rightarrow(i,g)}$
                bestT $\leftarrow \text{comm}_{(i-1,h)\rightarrow(i,g)}$
                bestAmp $\leftarrow \min(\text{bestAmp}, Amp(i-1, h))$
            end if
        end for
        $S[i][g] = \text{bestS} + \text{comp}_{(i,g)} + \text{sync}_{(i,g)}$
        $T[i][g] = \text{bestT} + \text{comp}_{(i,g)} + \text{sync}_{(i,g)}$
    end for
end for

4.2 Search algorithm

Naively bursting individual layers to the amplification limit won't give the best iteration time since frequent changes of scaling lead to higher communication costs. Thus, DeepPool needs a search algorithm to find the optimal parallelization plan.

The parallelization strategy search is composed of two parts. We use a dynamic programming algorithm that finds the optimal plan for a single chain of DNN layers, and we perform a graph reduction algorithm for DNN graphs that branch out to multiple parallel chains of layers.

Linear search on a single chain of layers. Our dynamic programming for a chain of layers is similar to shortest path search except that we consider the user-given limit on GPU-sec amplification. As input for search, we are given a chain of L layers indexed by $[1, L]$ and the number of available GPUs, G.

With dynamic programming, we compute two tables:

$$S[i][g] = \text{ shortest time to complete } L_1, \ldots, L_i$$
$$\text{while using } g \text{ GPUs for } L_i.$$

$$T[i][g] = \text{time spent on } L_i \text{ while minimizing } S[i][g]$$

The dynamic programming proceeds as shown in Algorithm 1. In the algorithm description, $Amp(i, g)$ denotes the GPU-sec amplification for L_i when scaled to g GPUs.

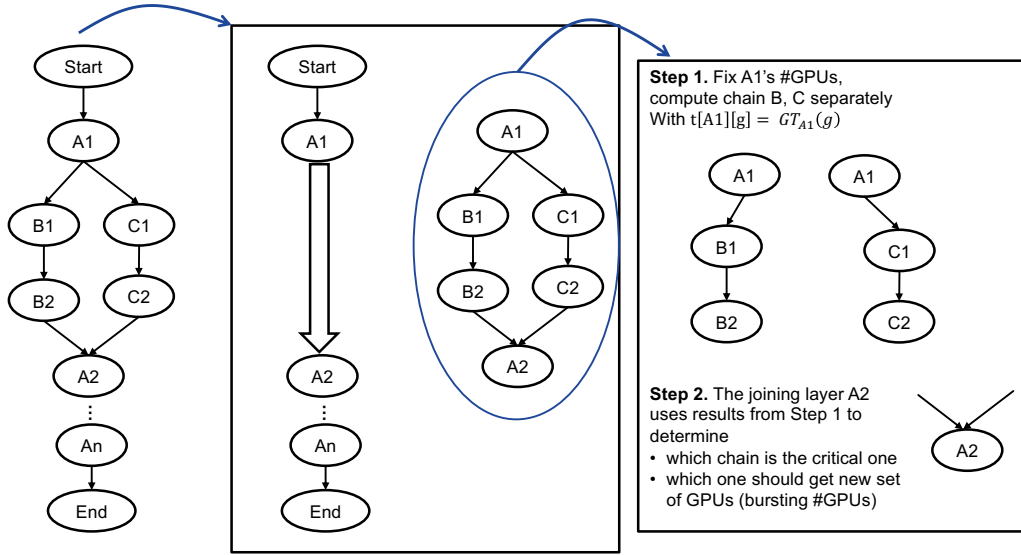$$Amp(i, g) = \frac{T[i][g] \times g}{\text{comp}_{(i,1)}}$$

Figure 7. The graph reduction process of multi-chain search problem into a single chain search problem.

Note that $T[i][g]$ includes communication overheads (`comm` and `sync`), so the amplification limit also counts the communication in. After populating S, we can backtrace from L_L to L_1 and find the number of GPUs used by each layer that minimize the total processing time while obeying the amplification limit.

Multi-chain graph reduction. We extend the single chain algorithm to an algorithm for general DNN computation graphs with branches and joins. The algorithm identifies portions of DNN graphs from the branching layer to the joining layer, and multiple chains inside each block are reduced to a single edge. Figure 7 shows how we reduce a complex graph into a chain of blocks, where each block branches at the beginning and joins at the end. When the linear search algorithm hits a branching layer (A1 in the example), it finds the corresponding joining layer (A2). The linear search algorithm consider the whole block as a single edge cost.

- `tr`$_{(i,g)\to(j,h)}$: time elapsed to transit from L_i to L_j when scaled to g and h GPUs respectively. This is same as `comm` if L_i is directly followed by L_j.

Given that we can reduce branch-join blocks as the transition cost, the linear search algorithm is identical to the single-chain search algorithm.

To compute the transition time between the branching layer and the joining layer, we utilize linear searches on branches. First, we fix the number of GPUs for the branching layer with one of the possible numbers, g. Then, we perform linear searches on all branches while restricting A1 must be scaled to g. After the individual linear search on each

branch, we merge the shortest time to reach the joining layer. The joining layer figures out the critical branch which takes the longest time and decides whether or not to run each other non-critical branch in parallel with the critical branch. For that consideration, we use communication overheads to move samples to and from a new set of GPUs and make sure it won't increase the total training time or overshoot the amplification limit. After merging the shortest times of all branches for all possible GPU counts for the joining layer, we repeat the process with other GPU counts for the branching layer.

5 MULTIPLEXING

DeepPool aims to make use of any unused compute resources left on each GPU in the cluster by concurrently executing a low-priority background job on the GPU alongside the main distributed training task. The goal is to maximize the throughput of the background job, raising the overall throughput and efficiency of the cluster, while ensuring that foreground jobs have nearly the same quality-of-service (QoS) that they would have if executing in isolation. Because it is one of the most common platforms used for training DNNs, DeepPool targets Nvidia's datacenter-class Tesla GPUs. While Nvidia's CUDA implementation provides some functionality for multiplexing tasks with different priorities, we find that the mechanisms are somewhat lacking. As a result, DeepPool must at times intentionally under-utilize the GPUs in order to protect QoS for the foreground job. In this section, we discuss the shortcomings of current GPU hardware for multiplexing, and the mechanisms that DeepPool leverages in order to use multiplexing
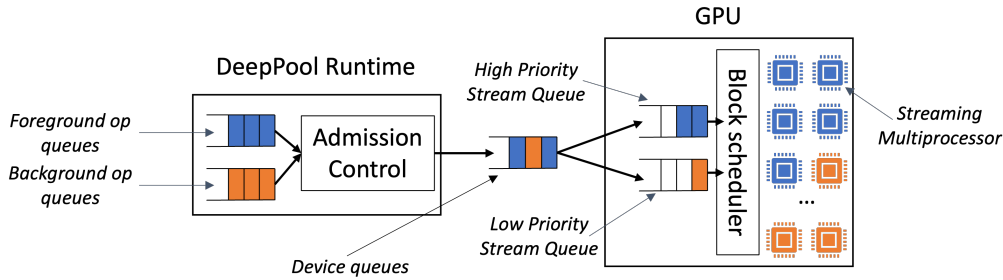
Figure 8. Overview of queues in DeepPool. DeepPool queues work from high and low priority tasks, and uses CUDA stream priorities to provide QoS for high-priority tasks. Because some queues inside the CUDA driver are shared by high and low-priority work, DeepPool keeps queueing at the device to a minimum.

to improve cluster utilization and efficiency, despite these shortcomings.

We start with some background on the execution model for CUDA devices. CUDA allows concurrent execution of tasks on the GPU using a mechanism called *CUDA streams*. CUDA streams allow a program to express dependencies between various operations that are launched on the device. Operations (e.g. compute kernels and memory operations) that are enqueued into a stream are executed on the GPU in FIFO order, while operations placed into different streams may execute concurrently, assuming sufficient compute resources are available on the GPU's streaming multiprocessors (SMs). CUDA streams can be assigned an integer priority at creation time, which allows the on-device scheduler to favor scheduling operations from certain streams over others. Unfortunately, the on-device scheduler on Nvidia GPUs is non-preemptive: while it may favor running kernels from high-priority streams, once it assigns a block of computation to one of its SMs, it allows it to run to completion without interrupting it. The limitation impacts our ability to extract higher utilization from the device when multiplexing high-priority and low-priority jobs: in order to reduce delays for the foreground job, DeepPool runs background DNN jobs with small batch sizes in order to decrease their execution latencies. Unfortunately, this also reduces the total possible utilization for the background job.

In addition to the non-preemptive scheduler, we found several other idiosyncrasies in Nvidia's implementation of stream priorities. We found that work can queue in a number of places that do not differentiate among the respective priorities of the queued work. For example, we found that allowing an unbounded number of kernel or graph launches from each task leads quickly to loss of QoS for the foreground job. We speculate that this is because launch requests from different priority streams are placed into shared transmission queues between the CUDA driver and the device, potentially starving the device of high-priority requests. DeepPool limits the number of outstanding queued requests to limit the amount of head-of-line blocking possibly ex-

perienced by foreground jobs. Figure 8 shows the various hardware queues in the system and illustrates our approach to managing them. We find that overall, several improvements could be made to Nvidia's software and hardware stack to improve efficiency, the most impactful being the addition of an on-device preemptive scheduler.

DeepPool's runtime contains a lightweight execution engine that mediates kernel launches on the GPU from among the set of jobs that are currently assigned to that GPU. The engine maintains a virtual execution queue for each DNN task along with an associated CUDA stream, and limits the number of outstanding launches from each queue onto the GPU to avoid interference in the device's execution queues. The engine leverages *CUDA graphs*, a recent CUDA feature addition that allows the program to group multiple consecutive operations together into a single launch operation, amortizing the costs of communicating with the device. Traditionally, CUDA programs submit each operation to the GPU using calls such as `cudaLaunchKernel`, which asynchronously enqueue the operation onto the device and return within several microseconds. When a computation contains small kernels with short execution times, the host can easily underutilize the device when the kernel runtime is shorter than the amount of time it takes for the host to prepare and enqueue the next operation. Many DNN models benefit heavily from CUDA graphs, particularly when the per-GPU minibatch size is low, and DeepPool's execution engine enables it for all jobs that it runs.

In addition to improving task throughput and GPU utilization, CUDA graphs play an important role in maintaining good QoS for foreground jobs in DeepPool. Because the GPU scheduler is non-preemptive, if the host falls behind in supplying high-priority operations to the device to execute, the device will schedule a lower-priority operation leading to a potentially long delay once the host does enqueue the next operation. CUDA graphs reduce interference by reducing opportunities for low-priority tasks to use device cycles that could be more immediately used for high-priority jobs. While CUDA graphs largely improve the performance

of DeepPool, we observe that the graph execution engine on the device is also vulnerable to head-of-line blocking for high-priority tasks when low-priority tasks have large graphs. To ensure that low-priority tasks with more kernels do not starve high-priority tasks, we split large CUDA graphs launches into groups of smaller graphs, and limit the number of outstanding graph launches.

We also observed that a few training operations were particularly sensitive to interference from low priority tasks. For example, NCCL's all-reduce operation, which is used to synchronize gradients during the backwards pass, more than doubles in execution time when another task is run on the same GPU. To handle such cases, the execution engine monitors the runtimes of each operation, and pauses collocation when a foreground job runs an operator that has been observed to suffer large slowdowns.

6 IMPLEMENTATION

We implement DeepPool on the C++ frontend of a recent release of PyTorch (v1.11.0 (LibTorch)) using CUDA (v11.6 (CUDA)) and NVIDIA's Collective Communications Library (NCCL, v2.11.4 (NCCL)). Recent additions to both PyTorch and NCCL allow us to easily create CUDA graphs containing full training iterations using CUDA's stream capture function. This function records all operations enqueued to the device during some observation period and enables the host to relaunch all the same operations with a single graph launch.

7 EVALUATION

Our evaluation aims to answer the following questions:

1. Can we improve the training throughput of each GPU while strong scaling a foreground job?

2. Does burst parallel training offer better combinations of total cluster throughput and foreground speedup than statically partitioning a cluster?

3. How do the individual techniques of DeepPool enable low-interference collocation?

4. When can we collocate background jobs without interfering with foreground performance?

5. How fast does DeepPool's burst parallel training planner find the optimized training plan?

To answer the questions above, we measured performance with three vision models: VGG-16 (Simonyan & Zisserman, 2015), WideResNet-101-2 (Zagoruyko & Komodakis, 2017), and Inception-V3 (Szegedy et al., 2015). Table 1 shows the number of operators, parameter size, input size, and main operator type for each network.

Table 1. Workload Characteristics

Model	Params	Layers	Input Size	Structure
VGG-16	132 M	21	3 x 224 x 224	Conv, Dense
WideResNet-101-2	127 M	105	3 x 400 x 400	Intense Conv
Inception-v3	24 M	119	3 x 299 x 299	Light Conv

Table 2. Hardware configuration.

GPU	8 × NVIDIA A100-SXM4-40GB
Interconnect	NVSwitch (600 GB/s for each GPU)
Driver	CUDA 11.6, cuDNN: v8.3.2, NCCL 2.11.4
CPU	2 × AMD EYPC 64 cores @ 1.5 GHz
RAM	988 GB
OS	Ubuntu 20.04 (Linux 5.13.0-28-generic)

We compare the performance of three different scenarios. "DP" is our baseline which runs only one data-parallel foreground task by evenly splitting the given global batch across all GPUs. "BP" uses burst parallel scheduling (§4) to scale down stages with low scalability in the foreground task. "BP+Col" collocates a low priority background task with the burst-parallel foreground job. Foreground and background tasks use the same model for ease of understanding GPU throughput. We optimized the baseline "DP" performance by removing kernel launch overheads with CUDA graphs. This optimization improved the baseline performance up to 2× for some models, and made it more challenging to collocate a background job because graphs enable better utilization of the GPU. To our knowledge, no prior work that has focused on collocating multiple DNN training tasks has made use of this feature.

Table 2 shows the hardware configurations for our experiments. Although our work is targeting very large scaling with hundreds of GPUs, we use a single NVIDIA DGX A100 server with 8 GPUs. For the purpose of demonstrating improvements on strong scaling efficiency, we use small per-GPU batch sizes which are required for large-scale training (as seen in §2) for our 8-GPU experiments, and enable Automatic Mixed Precision in PyTorch to leverage faster implementations of certain operators.

7.1 Overall performance

To demonstrate that DeepPool can achieve good training throughput while strong scaling a foreground task, Figure 9 compares the cluster-wide training throughput of foreground and background tasks with three different scenarios. For the two scenarios that use burst parallelism, we set the GPU-sec amplification limit and collocation parameters to minimize the impact on the foreground performance while having a reasonable gain on total training throughput. For VGG-16 and WideResNet-101-2, burst parallel scheduling alone (BP) improves foreground throughput compared to DP as it scales down the number of GPUs used for layers with low scalabil-

(a) VGG-16, scaling b=32   (b) WideResNet-101-2, scaling b=16   (c) InceptionV3, scaling b=32
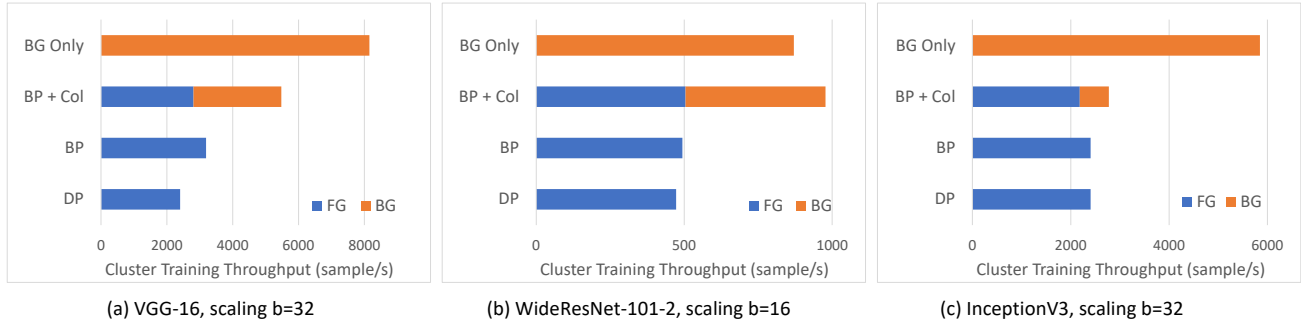
Figure 9. Cluster training throughput while strong scaling with three different scenarios on 8x A100 GPUs. "BG Only" shows the maximum throughput achievable when each GPU just runs the background task used for collocation. (a) VGG16: scaled with global batch size of 32. (b) WideResNet-101-2: scaled with global batch size of 16. (c) Inception-V3: scaled with global batch=32. In all three scenarios, using burst parallel scheduling with collocation dramatically improves cluster throughput with little impact on foreground training time.
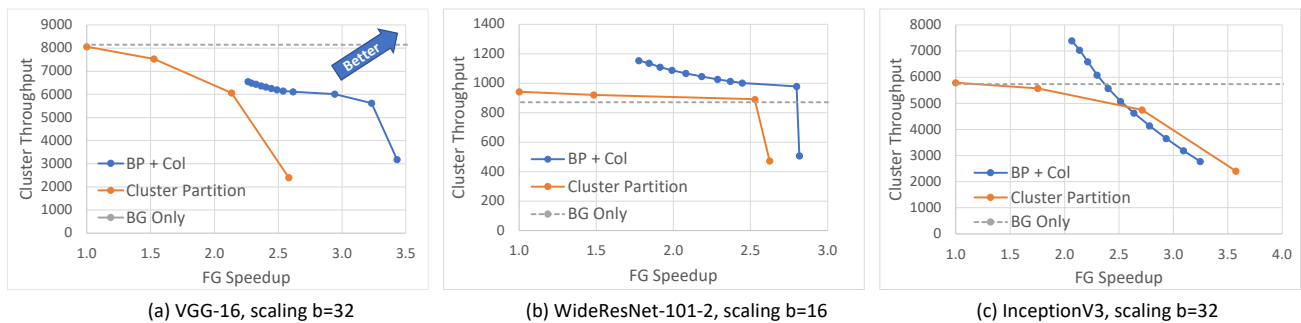

(a) VGG-16, scaling b=32   (b) WideResNet-101-2, scaling b=16   (c) InceptionV3, scaling b=32

Figure 10. Trade-offs between cluster total throughput and foreground job speedup with various operating points of "BP + Col" scenario for the workloads in Figure 9. Speedup is calculated relative to the same job running on a single GPU with the same global batch size. As the baseline, "Cluster Partition" shows four static partition configurations: 1, 2, 4, 8 GPUs for data-parallel foreground task and 7, 6, 4, 0 GPUs for background tasks respectively.

ity, reducing gradient sync overheads. Compared to "BP", "BP + Col" almost doubles the total cluster throughput with less than 18% degradation of foreground throughput, which is still higher than DP with 8 GPUs. For Inception-v3, the burst parallel schedule falls back to standard data-parallel strong scaling since the layers in Inception-v3 have largely even scalability across stages. "BP + Col" only extracts an additional 15% cluster throughput through collocation. This is because Inception-v3 contains a large number of layers with very short kernel execution times on the GPU, which makes it more vulnerable to interference from a background task. We explore this issue in more detail in §7.3. Additionally, without burst parallel scaling, there are fewer resources left for the background job to use.

To demonstrate DeepPool's benefit in depth, Figure 10 compares foreground speedups and cluster-wide training throughput for two different scenarios. It shows multiple operating points of "BP + Col" by varying the GPU-sec amplification limit and collocation parameters. As the baseline, it also shows multiple "Cluster Partition" configurations, which divide the 8 GPUs into data-parallel training groups

for the foreground task and individual local training groups for the background tasks. For VGG-16, "BP + Col" configurations can achieve up to 38% higher foreground speedups than the "Cluster Partition" configurations that provides the same cluster throughput. For WideResNet-101-2, "BP + Col" provides not only better foreground speedup (up to 11%) but also higher cluster throughput (up to 25%). For Inception-V3, "BP + Col" allows up to 37% higher foreground speedup when we want to operate the cluster with higher throughput than 2 foreground GPU cluster partitions.

7.2 Decomposing benefits of each technique

To illustrate the contributions of each mechanism for multiplexing in DeepPool discussed in §5, we iteratively add each mechanism and show its impact on system throughput and QoS for VGG-16 in Figure 11. Adding CUDA graphs yields a 15% gain in throughput. Models with larger numbers of small kernels may benefit even more. For example, Inception-V3 sees a 2.2× speedup when enabling CUDA graphs. Naively running a second lower-priority instance of VGG-16 dramatically reduces the throughput of the main
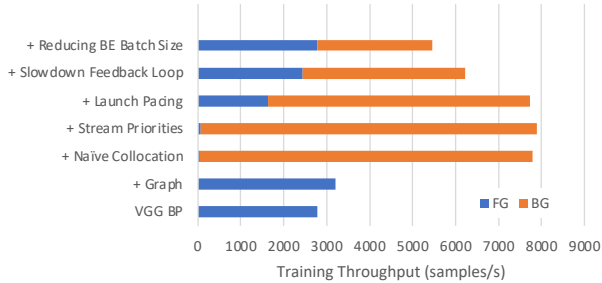
Figure 11. The contribution of each individual multiplexing mechanism towards QoS and throughput when multiplexing VGG-16 on a cluster with 8x A100 GPUs. From the bottom row up, each row illustrates the impact of the addition of each technique.

Figure 12. Pairwise collocation of various synthetic CUDA kernels with varied compute intensity and execution latency using stream priorities. Each box shows the high priority kernel's achieved throughput in each collocation as a percentage of its peak achievable throughput when run in isolation. CUDA stream priorities are largely effective, but the non-preemptive device scheduler struggles to provide good QoS for short kernels when collocated with low-priority long-running kernels.

task, and adding the use of stream priorities to differentiate between the two has little impact. By limiting the number of outstanding operations enqueued to the device by each task ("launch pacing"), DeepPool reduces contention in shared device queues and improves the throughput for the high-priority task, allowing stream priorities to be used by the device to more effectively differentiate between high and low-priority tasks. By monitoring the slowdown of each individual operator and banning collocation for highly sensitive operators ("slowdown feedback loop"), DeepPool further improves the QoS for the high-priority task. Finally, DeepPool reduces the batch size of the background job in order to reduce the impact of having a non-preemptive scheduler on the foreground job.

7.3 Analysis on collocatability

To better understand the limits of multiplexing, we run a microbenchmark to explore the impact of non-preemption in the GPU scheduler. We examine the pairwise collocation of several synthetic kernels with varied compute intensities and execution latencies. Figure 12 shows the throughput degradation for each kernel when run in a high-priority stream when each other kernel is run in a low-priority stream. We observe that streams priorities are largely effective except for in the case where the high-priority tasks have very low execution latency while the low-priority tasks have high execution latency. For this reason, DeepPool limits the batch size used in best-effort tasks, resulting in lower latency kernels that cause less interference. Preemptive scheduling on the device would allow for better QoS for short-running high-priority kernels.

7.4 Performance of burst parallel training plan search

DeepPool's burst parallel training planner uses dynamic programming and graph reduction to find the optimal parallelization plan. To prove that these two search methods are fast enough to be practical, we benchmarked our single-threaded Python implementation of the search algorithm.
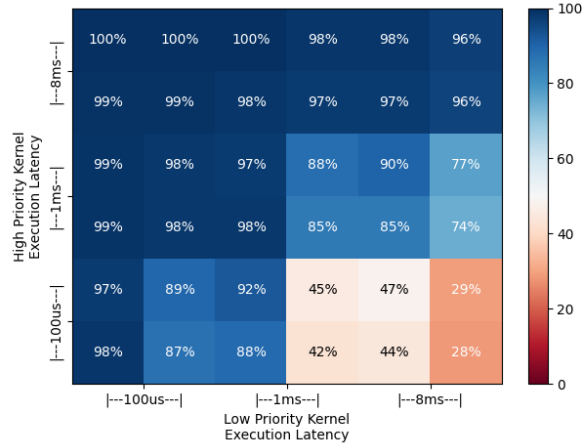
Table 3. Time to search for burst parallel training plans

Scale	8 GPUs	1024 GPUs
VGG-16	0.01 sec	0.05 sec
WideResNet-101-2	0.02 sec	0.11 sec
Inception-v3	0.22 sec	3.23 sec

Table 3 shows the measured search time for two settings: 8 GPUs and 1024 GPUs. For all models, search completes within a few seconds. To limit the growth of the search space, our planner only considers GPU counts that are powers of two. Thanks to this optimization, search time grows only by 5-15× even when we scale from 8 GPUs to 1024 GPUs. The search for Inception-v3 takes longer than other models since it has multiple branches, so its search must perform the graph reduction algorithm.

8 RELATED WORK

Statistical efficiency of training. Increasing the mini-batch size to scale out the number of GPUs often reduces the statistical efficiency of training, resulting in a much longer training time to reach the same accuracy. A wide study (Shallue et al., 2018) showed that the statistical efficiency depends on model architecture and data set. There were efforts to increase the maximum sizes of mini-batches without hurting the statistical efficiency (Goyal et al., 2017; Mikami et al., 2018; You et al., 2017). However, those works were often targeted to specific DNN architectures, so it's not obvious how to apply the same technique to other new archi-

tectures. Statistical efficiency is now considered for cluster schedule optimization as well. Pollux (Qiao et al., 2021) is a cluster scheduler that measures the change of statistical efficiency and uses the effective maximum mini-batch size to optimize training goodput.

Collocating multiple jobs on a GPU. Several prior works have observed low utilization of both compute and memory GPU resources during DNN training and propose concurrent execution of either multiple training jobs or parallel stages within a single training task to improve utilization. Gandiva (Xiao et al., 2018) attempts to pack multiple jobs onto a single GPU when the cluster is overloaded, and avoids colocations that result in a net throughput loss. Wavelet (Wang et al., 2021) and Salus (Yu & Chowdhury, 2020) aim to improve GPU utilization by interleaving or overlapping execution stages that have different levels of memory usage. IOS (Ding et al., 2021) and Rammer (Ma et al., 2020) exploit inter-operator parallelism to maximize job throughput, concurrently running operators from parallel branches of a single training task. While several of these works identify host-side scheduling overheads as a contributing factor to GPU compute underutilization, none leverage CUDA graphs to amortize kernel launch costs. Additionally, none of these works aim to provide performance isolation between concurrently executing foreground and background jobs.

Strong scaling on other dimensions. Another approach to tackle the diminishing speedup from strong scaling is scaling on dimensions beyond samples (i.e., reducing samples per GPU). Recent work (Jia et al., 2018; Dryden et al., 2019) explored other dimensions such as splitting height/width of image samples or splitting parameters. However, those unconventional splits incur communication overhead after every layer to prepare input for the next layer (e.g., halo exchange). Our parallelization strategy planner can explore those other dimensions for strong scaling. However, with the latest hardware and new vendor-provided tools like CUDA graphs, the computation cost of each layer is often reduced below 100 us, so the communication penalty of those unconventional splits now exceeds the improvement on computation time. Thus, we primarily focused on burst scaling on the sample dimension only. Pipeline parallelism (Narayanan et al., 2019; Huang et al., 2019) is often used to scale training throughput. But pipeline parallelism doesn't directly reduce iteration time and sacrifices statistical efficiency, so we don't consider it in this work.

Gradient all-reduce overhead. As we reduce the iteration time through strong scaling, the gradient all-reduce overhead becomes more critical. Along with improvements in networking speed, there were many efforts on reducing the overhead. To reduce the actual number of bytes sent over the network, prior works have investigated various forms of gradient compression, notably, quantization (Seide et al., 2014; Alistarh et al., 2016) and sparsification (Strom, 2015; Dryden et al., 2016; Lin et al., 2017), and also have designed efficient collective operations tailored for inherently sparse data (Renggli et al., 2019; Fei et al., 2021). Leveraging programmable switch hardware capabilities, the idea of in-network aggregation has been revisited in the context of deep neural network training (Sapio et al., 2019; Lao et al., 2021).

Cluster-wide optimization. As the compute demands for deep neural network training have grown steeply, it has become the norm to build a cluster with concentrated compute resources and let users share them. To efficiently serve multiple training jobs being submitted to the shared cluster in real-time, various dynamic scheduling algorithms have been developed targeting a wide range of performance metrics such as average job completion time (Peng et al., 2018; Gu et al., 2019; Hwang et al., 2021), fairness (Mahajan et al., 2020), cluster utilization (Xiao et al., 2018), and throughput-cost efficiency (Or et al., 2020).

9 CONCLUSION

We have introduced two techniques to improve the efficiency of strong scaling: burst parallel training and GPU multiplexing. By bursting the number of GPUs only to stages that can benefit from the additional GPUs, we save GPU resources. We showed that the saved resources can be reclaimed by a collocated background job with only a small impact to the foreground job. We also show that further improvements to hardware design can yield efficiency gains. These two techniques significantly improve the overall cluster throughput while strong scaling a time-critical foreground job.

ACKNOWLEDGMENTS

A ARTIFACT

DeepPool's source code along with instructions for building and running DeepPool are available at https://github.com/DeepPoolML/DeepPool.

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pp. 265–283, 2016.

Alistarh, D., Li, J., Tomioka, R., and Vojnovic, M. Qsgd: Randomized quantization for communication-optimal stochastic gradient descent. *arXiv preprint arXiv:1610.02132*, 1, 2016.

Ben-Nun, T. and Hoefler, T. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43, 2019.

CUDA. NVIDIA CUDA toolkit documentation. https://docs.nvidia.com/cuda/index.html/. Accessed: 2021-10-14.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pp. 1223–1231, 2012.

Ding, Y., Zhu, L., Jia, Z., Pekhimenko, G., and Han, S. Ios: Inter-operator scheduler for cnn acceleration. In Smola, A., Dimakis, A., and Stoica, I. (eds.), *Proceedings of Machine Learning and Systems*, volume 3, pp. 167–180, 2021. URL https://proceedings.mlsys.org/paper/2021/file/38b3eff8baf56627478ec76a704e9b52-Paper.pdf.

Dryden, N., Moon, T., Jacobs, S. A., and Van Essen, B. Communication quantization for data-parallel training of deep neural networks. In *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*, pp. 1–8. IEEE, 2016.

Dryden, N., Maruyama, N., Benson, T., Moon, T., Snir, M., and Van Essen, B. Improving strong-scaling of cnn training by exploiting finer-grained parallelism. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 210–220, 2019. doi: 10.1109/IPDPS.2019.00031.

Fei, J., Ho, C.-Y., Sahu, A. N., Canini, M., and Sapio, A. Efficient sparse collective communication and its application to accelerate distributed deep learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pp. 676–691, 2021.

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

Gu, J., Chowdhury, M., Shin, K. G., Zhu, Y., Jeon, M., Qian, J., Liu, H., and Guo, C. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pp. 485–500, 2019.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., and Chen, z. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems 32*, pp. 103–112. Curran Associates, Inc., 2019. URL http://papers.nips.cc/paper/8305-gpipe-efficient-training-of-giant-neural-networks-using-pipeline-parallelism.pdf.

Hwang, C., Kim, T., Kim, S., Shin, J., and Park, K. Elastic resource sharing for distributed deep learning. 2021.

Jia, Z., Zaharia, M., and Aiken, A. Beyond data and model parallelism for deep neural networks, 2018.

Khani, M., Ghobadi, M., Alizadeh, M., Zhu, Z., Glick, M., Bergman, K., Vahdat, A., Klenk, B., and Ebrahimi, E. Sip-ml: High-bandwidth optical network interconnects for machine learning training. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, pp. 657–675, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383837. doi: 10.1145/3452296.3472900. URL https://doi.org/10.1145/3452296.3472900.

Kumar, S., Wang, Y., Young, C., Bradbury, J., Kumar, N., Chen, D., and Swing, A. Exploring the limits of concurrency in ml training on google tpus. In Smola, A., Dimakis, A., and Stoica, I. (eds.), *Proceedings of Machine Learning and Systems*, volume 3, pp. 81–92, 2021. URL https://proceedings.mlsys.org/paper/2021/file/28dd2c7955ce926456240b2ff0100bde-Paper.pdf.

Lao, C., Le, Y., Mahajan, K., Chen, Y., Wu, W., Akella, A., and Swift, M. M. Atp: In-network aggregation for multi-tenant learning. In *NSDI*, pp. 741–761, 2021.

LibTorch. The C++ frontend of PyTorch (LibTorch). https://pytorch.org/cppdocs/frontend.html/. Accessed: 2021-10-14.

Lin, Y., Han, S., Mao, H., Wang, Y., and Dally, W. J. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

Ma, L., Xie, Z., Yang, Z., Xue, J., Miao, Y., Cui, W., Hu, W., Yang, F., Zhang, L., and Zhou, L. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 881–897. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/ma.

Mahajan, K., Balasubramanian, A., Singhvi, A., Venkataraman, S., Akella, A., Phanishayee, A., and Chawla, S. Themis: Fair and efficient {GPU} cluster scheduling. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pp. 289–304, 2020.

Mikami, H., Suganuma, H., Tanaka, Y., Kageyama, Y., et al. Massively distributed sgd: Imagenet/resnet-50 training in a flash. *arXiv preprint arXiv:1811.05233*, 2018.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pp. 1–15, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359646. URL https://doi.org/10.1145/3341301.3359646.

NCCL. NVIDIA deep learning NCCL documentation. https://docs.nvidia.com/deeplearning/nccl/index.html/. Accessed: 2021-10-14.

Or, A., Zhang, H., and Freedman, M. Resource elasticity in distributed deep learning. In Dhillon, I., Papailiopoulos, D., and Sze, V. (eds.), *Proceedings of Machine Learning and Systems*, volume 2, pp. 400–411, 2020. URL https://proceedings.mlsys.org/paper/2020/file/006f52e9102a8d3be2fe5614f42ba989-Paper.pdf.

Peng, Y., Bao, Y., Chen, Y., Wu, C., and Guo, C. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, pp. 1–14, 2018.

Qiao, A., Choe, S. K., Subramanya, S. J., Neiswanger, W., Ho, Q., Zhang, H., Ganger, G. R., and Xing, E. P. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pp. 1–18. USENIX Association, July 2021. ISBN 978-1-939133-22-9. URL https://www.usenix.org/conference/osdi21/presentation/qiao.

Renggli, C., Ashkboos, S., Aghagolzadeh, M., Alistarh, D., and Hoefler, T. Sparcml: High-performance sparse communication for machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2019.

Rosset, C. Microsoft research blog, Feb 2020. URL https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/.

Sapio, A., Canini, M., Ho, C.-Y., Nelson, J., Kalnis, P., Kim, C., Krishnamurthy, A., Moshref, M., Ports, D. R., and Richtárik, P. Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701*, 2019.

Seide, F., Fu, H., Droppo, J., Li, G., and Yu, D. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*. Citeseer, 2014.

Shallue, C. J., Lee, J., Antognini, J., Sohl-Dickstein, J., Frostig, R., and Dahl, G. E. Measuring the effects of data parallelism on neural network training. *arXiv preprint arXiv:1811.03600*, 2018.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using gpu model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition, 2015.

Strom, N. Scalable distributed dnn training using commodity gpu cloud computing. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision, 2015.

Wang, G., Wang, K., Jiang, K., LI, X., and Stoica, I. Wavelet: Efficient dnn training with tick-tock scheduling. In Smola, A., Dimakis, A., and Stoica, I. (eds.), *Proceedings of Machine Learning and Systems*, volume 3, pp. 696–710, 2021. URL https://proceedings.mlsys.org/paper/2021/file/c81e728d9d4c2f636f067f89cc14862c-Paper.pdf.

Xiao, W., Bhardwaj, R., Ramjee, R., Sivathanu, M., Kwatra, N., Han, Z., Patel, P., Peng, X., Zhao, H., Zhang, Q., et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pp. 595–610, 2018.

You, Y., Gitman, I., and Ginsburg, B. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*, 2017.

Yu, P. and Chowdhury, M. Fine-grained gpu sharing primitives for deep learning applications. In Dhillon, I., Papailiopoulos, D., and Sze, V. (eds.), *Proceedings of Machine Learning and Systems*, volume 2, pp. 98–111, 2020. URL https://proceedings.mlsys.org/paper/2020/file/f7177163c833dff4b38fc8d2872f1ec6-Paper.pdf.

Zagoruyko, S. and Komodakis, N. Wide residual networks, 2017.