ReCycle: Resilient Training of Large DNNs using Pipeline Adaptation

Swapnil Gandhi
gandhis@stanford.edu
Stanford University

Mark Zhao
myzhao@cs.stanford.edu
Stanford University

Athinagoras Skiadopoulos
askiad@stanford.edu
Stanford University

Christos Kozyrakis
kozyraki@stanford.edu
Stanford University

Abstract

Training large Deep Neural Network (DNN) models requires thousands of GPUs over the course of several days or weeks. At this scale, failures are frequent and can have a big impact on training throughput. Utilizing spare GPU servers to mitigate performance loss becomes increasingly costly as model sizes grow. ReCycle is a system designed for efficient DNN training in the presence of failures, without relying on spare servers. It exploits the inherent *functional redundancy* in distributed training systems – where servers across data-parallel groups store the same model parameters – and *pipeline schedule bubbles* within each data-parallel group. When servers fails, ReCycle dynamically re-routes micro-batches to data-parallel peers, allowing for uninterrupted training despite multiple failures. However, this re-routing can create imbalances across pipeline stages, leading to reduced training throughput. To address this, ReCycle introduces two key optimizations that ensure re-routed micro-batches are processed within the original pipeline schedule's bubbles. First, it decouples the backward pass into two phases: one for computing gradients for the input and another for calculating gradients for the parameters. Second, it avoids synchronization across pipeline stages by staggering the optimizer step. Together, these optimizations enable adaptive pipeline schedules that minimize or even eliminate training throughput degradation during failures. We describe a prototype for ReCycle and show that it achieves high training throughput under multiple failures, outperforming recent proposals for fault-tolerant training such as Oobleck and Bamboo by up to 1.46× and 1.64×, respectively.

SOSP '24, November 4–6, 2024, Austin, TX, USA

ACM ISBN 979-8-4007-1251-7/24/11
https://doi.org/10.1145/3694715.3695960

CCS Concepts: • **Computer systems organization** → **Dependable and fault-tolerant systems and networks**; **Distributed architectures**; **Neural networks**.

Keywords: Fault-tolerant Training, Distributed Training, Hybrid Parallelism, Pipeline Adaptation

ACM Reference Format:
Swapnil Gandhi, Mark Zhao, Athinagoras Skiadopoulos, and Christos Kozyrakis. 2024. ReCycle: Resilient Training of Large DNNs using Pipeline Adaptation. In *ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24), November 4–6, 2024, Austin, TX, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3694715.3695960

1 Introduction

Deep Neural Networks (DNNs) are consistently achieving milestone results in domains such as natural language processing [54], speech recognition [25], and computer vision [39]. Since the rapid growth of DNNs is a major contributor to recent breakthroughs [69], we are now in a global race to develop increasingly-large foundation models [9, 26]. Both proprietary [2, 10, 73] and open-source [14, 68] models have tens to hundreds of billions of parameters. Training such models requires uninterrupted access to thousands of accelerators (e.g., GPUs) for several weeks [33, 53, 65]. DNN training is essentially a large supercomputing job that relies on hybrid parallelism, which concurrently leverages data, tensor, and pipeline parallelism strategies [78].

As the scale and duration of training increases, so does the likelihood of encountering failures [19, 22, 30, 33, 74, 79]. While scale-out workloads, such as distributed databases and analytics, are designed to work around server failures, this is not yet the case for DNN training. The rigid parallelization of DNN training implies that each failure can set off a domino effect. All resources are forced to idle while a failed node is replaced and the job is re-optimized and re-started on the new hardware configuration. For example, the training of OPT-175B included 178,000 GPU-hours of wasted time due to various malfunctions [40].

Fault tolerance in DNN training involves three issues: *fault detection*, *checkpointing*, and *efficient execution in the presence of faults*. Large-scale training systems implement

comprehensive monitoring of the health of hardware and software components, so that faults or stragglers are quickly detected and diagnosed [33]. Recent research has focused on reducing the overhead of periodic checkpoints of the training job state [19, 37, 50]. Efficient execution in the presence of faults has received less attention. A common approach in industry is to maintain a reserve of spare GPU servers that will replace failed servers when needed. While conceptually simple, this approach is expensive at scale. As the frequency of faults increases, so does the number of spares.

The alternative approach is to *continue training with the subset of resources available*. This is similar to how scale-out systems approach resilience to failures [17]. Data parallelism provides one simple implementation as it creates full replicas of the model across groups of GPUs. When one GPU server fails, we take all the servers in its data-parallel group offline and continue training the remaining model replicas. Unfortunately, this approach increases the blast radius of the failure and its impact to performance. When training a 530B GPT model using the hybrid-parallel scheme of Megatron-LM [52], a single node failure would force 280 GPUs to go offline and cause a roughly 11% drop in training throughput.

Two recent projects have exploited *pipeline parallelism* instead. Bamboo [67] relies on redundant computation and executes each pipeline stage on two nodes even in the fault-free case. When a node fails, the alternative nodes for its stages will do the work. Bamboo's drawback is the drop in fault-free training throughput due to the redundant computations and added memory pressure on each GPU node. Oobleck [29] avoids overheads in fault-free execution by pre-computing a number of templates for pipeline parallelism that use a different number of nodes. As nodes fail, Oobleck replaces the original pipeline template with one that uses fewer nodes. The performance challenge for Oobleck is balancing work across heterogeneous pipelines as the pipelines with fewer nodes can become stragglers. Both Bamboo and Oobleck require significant re-configuration as nodes fail or rejoin, which can be a challenge as failure rates increase.

In this paper, we introduce *ReCycle*, a novel scheme for resilient training with hybrid-parallel systems. ReCycle enables efficient execution in the presence of failures, without the need for spares and without any impact on model accuracy. Similar to Bamboo and Oobleck, we exploit pipeline parallelism. But unlike these previous proposals that handle failures within a single pipeline, ReCycle utilizes the inherent *functional redundancy across pipelines* in hybrid-parallel training systems. ReCycle re-routes the micro-batches of failed nodes to *peer nodes* that process the same pipeline stage for the model in the other data-parallel groups. Peer nodes store the same model parameters and can handle the extra work without the need for parameter re-shuffling.

If done naively, ReCycle's *adaptive pipelining* increases training time and memory usage due to the additional work for the peers of failed nodes. ReCycle uses a series of optimizations that exploit unique characteristics of hybrid-parallel training to eliminate all or most of the inefficiencies. ReCycle uses the *bubbles* in the pipeline schedule of peers to execute the additional work at low or no overhead. These bubbles typically appear during the *start-up* and *cool-down* phase of the pipeline schedule, but not during the *steady-state* when most additional work must occur. ReCycle overcomes this challenge by *decoupling back propagation* for micro-batches into two distinct gradient computations, one relative to the input and one relative to the parameters. The two steps are independently scheduled in order to better leverage bubbles in the the *cool-down* portion of the pipeline schedule. The selective application of decoupled back propagation exploits the imbalance in memory usage across stages in hybrid-parallel training while avoiding an increase in peak memory pressure. Finally, ReCycle staggers the execution of the optimizer tasks across pipeline stages in order to leverage bubbles from the *start-up* phase of the pipeline schedule for the next training iteration.

We implemented ReCycle on top of the DeepSpeed training framework [60]. The key component of ReCycle is the *Planner*, which utilizes dynamic programming and mixed-integer linear programming (MILP) to determine adaptive pipeline schedule in presence of multiple failures. Operating offline, the *Planner* precomputes efficient schedules for a predefined number of failures, which are then applied as necessary during training. We evaluated ReCycle using DNNs with billions of parameters like GPT-3 [10], using both real-world experiments and simulations of large training systems. We show that ReCycle supports high throughput training even at high failure counts, e.g. ≥10% of the overall system. ReCycle outperforms Oobleck by 1.46× in training scenarios with realistic GPU failures. ReCycle's advantage stems from its efficient scheduling of re-routed work and its ability to avoid major re-shuffling of model parameters upon failures. ReCycle outperforms Bamboo by 1.64×. It is also able to efficiently train much larger models than Bamboo, which requires large memory overheads for its fault tolerance mechanism.

2 Background and Motivation

2.1 Distributed DNN Training

State-of-the-art *Deep Neural Network (DNN)* models consist of tens to hundreds of billions of parameters and are trained on datasets with many trillions of tokens [2, 10, 20, 40, 41, 68, 76]. Their training requires thousands of GPUs for days or weeks [26, 35, 53, 65]. For example, Llama-3 was trained on 15 trillion tokens, using two clusters of 24K GPUs [3].

DNN training is parallelized using three primary forms of parallelism. *Data parallelism* (DP) processes subsets of the input data in parallel across GPU groups, each of which stores the entire model [21, 62, 75]. Data parallelism requires

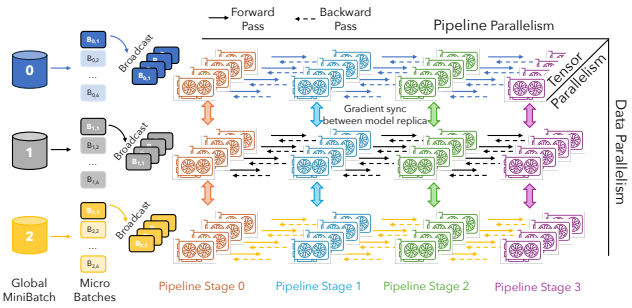


Figure 1. Illustration of hybrid parallelism. Pipeline stages are denoted with different colors. Within each pipeline stage, operators are partitioned through tensor parallelism. The global batch is split into micro-batches across pipelines.

high network bandwidth for the all-reduce operations that reconcile model parameters across all replicas at the end of iteration steps. Tensor and pipeline parallelism are forms of *model parallelism* that facilitate training large models by sharding the model across GPUs. *Tensor parallelism* (TP) partitions the parameters of each layer across GPUs in order to parallelize each linear algebra operation within a layer [64]. It incurs high communication costs that are not easily hidden by computation due to frequent all-reduce operations in both the forward and backward pass. *Pipeline parallelism* (PP) divides the model into sequential groups of layers or *stages*. Micro-batches of data are processed in parallel in a pipelined manner across these stages [27, 36, 43, 46, 51]. Pipeline parallelism requires lower network bandwidth as stages only communicate activations and gradients at layer boundaries. However, it achieves lower compute utilization as the number of stages increases due to pipeline dependencies and bubbles (idle slots) in the pipeline schedule.

Large-scale training systems balance these trade-offs by utilizing all three forms of parallelism as shown in Figure 1 [37, 43, 44, 46, 52, 60, 77, 78]. This is known as *hybrid-parallelism*. In the most common case, systems such as DeepSpeed [60] or Alpa [78] use a combination of tensor parallelism within a multi-GPU server, pipeline parallelism across multi-GPU servers, and data parallelism across pipelines. Hybrid parallelism enables large model training with graceful scaling and reasonable training times (weeks to few months) using optimized clusters with 1000s of densely-connected GPUs.

2.2 Fault Tolerance in Distributed DNN Training

Large training systems include thousands of GPUs, CPUs, memory chips, networking chips, cables of various types, and power conversion and cooling devices. Scale allows for high performance but also leads to frequent faults ranging from software errors to full hardware malfunctions. The Mean Time Between Failure (MTBF) for large training systems can be minutes [22, 24, 33, 74]. For example, large-scale training clusters at Microsoft see a failure every ≈ 45 minutes [30].

In contrast to scale-out frameworks like Map-Reduce [17], which quickly adjust to dynamic changes in resource availability, DNN training operates like a supercomputing job. It relies on fixed sharding and parallelization strategies and gang-scheduled execution. All computational resources must concurrently run uninterrupted, making the system highly susceptible to disruptions from any failure. For example, Meta encountered over 100 hardware failures while training OPT-175B, resulting in the loss of 178,000 GPU-hours [8, 40]. Similar failure rates have been reported by ByteDance [33], Alibaba [24], LAION [7], Microsoft [30] and Google [79]. At this scale, faults are regular occurrences that require systematic optimizations to ensure resilient and efficient training [24, 30, 33, 74].

Enhancing fault tolerance in distributed training requires addressing three critical areas: fast error detection, checkpointing, and efficient execution in presence of faults.

2.2.1 Error Detection. Fast Error Detection is vital for minimizing downtime and preventing the propagation of errors. Effective error detection mechanisms include both hardware and software solutions. Hardware mechanisms, such as error correction codes (ECC), provide immediate detection and correction of data corruption. Software solutions can involve timeouts and heartbeat signals to identify system failures quickly. Additionally, some errors, particularly those that are silent or not immediately apparent, are detected through irregularities in the loss function. These anomalies, often indicative of silent data corruptions, can significantly impact training outcomes if not addressed promptly [5, 7, 15].

2.2.2 Checkpointing. Checkpointing plays a crucial role in fault tolerance by allowing the system to recover from a recent stable state rather than restarting the entire training process. It involves saving the model's state at regular intervals. However, naive checkpointing methods that write data to remote storage can introduce significant pauses in the training process due to latency and bandwidth constraints. To address this, extensive research has focused on optimizing checkpointing techniques to reduce overhead and improve efficiency [19, 33, 50, 73], as detailed in Section 7.

2.2.3 Efficient DNN Training in the Presence of Failures. Our work focuses on ensuring that the system remains operational even when parts of it fail. A common method is to use *warm* spares, preloaded with model parameters to reduce recovery time. While this ensures that training efficiency returns to pre-failure levels, it becomes costly as systems scale and fault frequency rises. Spare servers need similar network bandwidth as the ones they replace, adding networking costs and requiring coarse-grained allocation [34]. For example, training a 530B GPT model requires 280 spare GPUs, provisioned according to the full capacity of a data parallel group, increasing costs by 11% [52]. Use of *cold* spares to run small

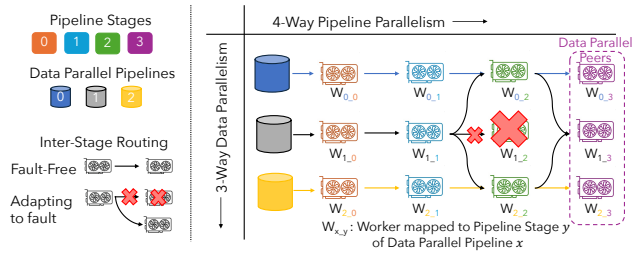


Figure 2. Adaptive pipelining when W_{1_2} fails. The micro-batches from worker W_{1_1}, originally intended for W_{1_2}, are dynamically re-routed to workers W_{0_2} and W_{2_2}, ensuring that the training process continues without interruption.

jobs can offset some of these costs, however eviction delays upon failure can cause significant stalls for large jobs.

An alternative approach is to continue training with the largest subset of available resources. Ideally, if x% of GPU servers fail, training proceeds at no less than $(100 - x)$% of fault-free throughput. A simple method, known as elastic batching, drops an entire data parallel group when a node fails, allowing the remaining groups to continue with the same parallelization [45, 72]. However, this approach has significant drawbacks, as a single node failure can take $PP \times TP$ nodes offline, reducing throughput by $1/DP$ (e.g., 17% for a 1T parameter GPT model [52]). Moreover, this approach requires reducing the batch size or recompiling the program to optimize each data parallel group for a larger batch portion, disrupting the balance and efficiency of the training job.

Two recent projects use pipeline parallelism for fault-tolerant training without spares. Bamboo [67], drawing inspiration from RAID [56], employs redundant computation (RC) where each pipeline stage is replicated on two nodes, even in fault-free cases. When a node fails, its backup handles its and forward and backward passes of failed node. Though RC hides some overhead in pipeline bubbles, it still significantly reduces throughput (see Section 6) and increases GPU memory pressure, limiting scalability. Moreover, Bamboo must restart with a full reconfiguration from a checkpoint for as few as two adjacent node failures.

Oobleck [29] uses pipeline parallelism to ensure resilient execution with no overhead when no faults occur. It utilizes precomputed templates for pipeline parallelism, each differing in stages, micro-batch configurations, and node counts. If a pipeline encounters a failed node, it switches to a template with fewer nodes, which may reduce training throughput. To prevent a slow pipeline from affecting the overall job, Oobleck distributes the global mini-batch based on compute power, leading to increased training time. Pipeline re-configuration when nodes fail or re-join can also become an overhead if failures are frequent. Additionally, Oobleck treats each pipeline as a *black-box* and does not leverage bubbles to mitigate overheads from node failures.

3 ReCycle Techniques

ReCycle aims to support efficient distributed training in the presence of faults. It requires no spare servers and has no impact on model accuracy compared to fault-free training. ReCycle can tolerate multiple hardware failures and maintains training throughput proportional to the number of functional servers available. ReCycle is optimized for fast recovery as failures do not require significant re-shuffling of model parameters between functional nodes.

This section reviews the key ReCycle techniques: *Adaptive Pipelining*, *Decoupled BackProp*, and a *Staggered Optimizer*. Section 4 presents the ReCycle system design.

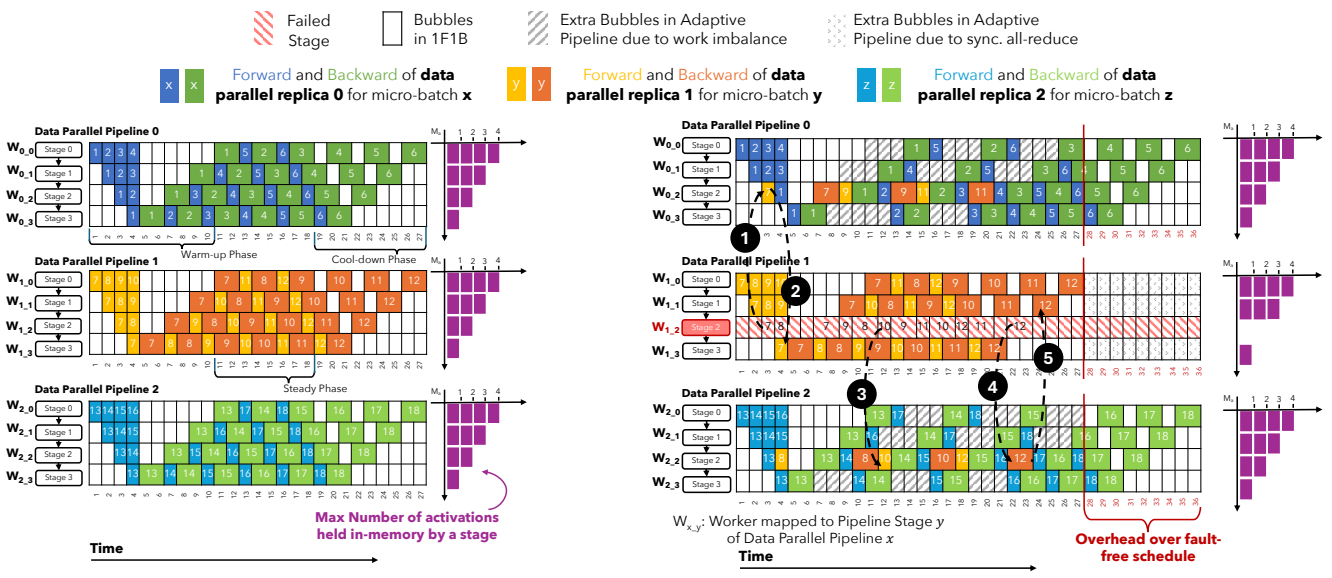
3.1 Adaptive Pipelining: Working Around Failures

ReCycle exploits two key properties in hybrid-parallel training systems. First, there is *functional redundancy* across data-parallel pipelines. The GPUs that process the same pipeline stage hold identical parameters and only differ in the micro-batches they process. In Figure 2, for example, workers W_{0_2}, W_{1_2}, and W_{2_2} are *peers* that hold identical parameters for stage 2 of the 4-stage pipeline. Second, there are *bubbles (idle slots) in the pipeline schedule* for each worker. The 1F1B pipeline schedule [51] in Figure 3a has 9 bubbles in the repeating 27-slot schedule for worker W_{0_2}.

Adaptive Pipelining exploits functional redundancy and bubbles by *dynamically re-routing micro-batches* from a failed worker to its functioning peers in other data-parallel pipelines. We aim to use the bubbles in peers' schedules to process the micro-batches for the failed worker with a low performance penalty. We evenly distribute micro-batches across all functional peers. Since all pipelines perform a synchronized all-reduce at the end of each iteration, load balancing ensures that no single worker (and thus pipeline) is overloaded, delaying the progress of the entire training iteration.

Figure 2 shows an example with three data parallel, 4-stage pipelines. Worker W_{1_2} fails. Upon detecting the failure, *Adaptive Pipelining* will redirect its input to workers W_{0_2} and W_{2_2}. These two peer workers will process micro-batches received from worker W_{1_1} in addition to their regular pipeline load. The output for these additional micro-batches will be sent back to worker W_{1_3}, the original recipient of the output of the failed worker W_{1_2}. All other workers operate as in the fault-free schedule. In essence, *Adaptive Pipelining* repairs the functionality of pipeline 1 by exploiting bubbles in the data parallel peers of failed worker W_{1_2}. Note that *Adaptive Pipelining* requires *no model parameter re-shuffling or re-partitioning across workers* in order to resume after a failure. Hence recovery is fast, unlike schemes like Oobleck that require re-configuring an entire pipeline upon a failure [29].

Figure 3 shows the detailed 1F1B schedule for the system in Figure 2. After the failure, the *forward pass* micro-batch 7 for failed worker W_{1_2} is re-assigned to its functional peer W_{0_2} at time step 3 (❶). The subsequent output is forwarded



(a) Fault-Free 1F1B Schedule

(b) Adaptive Schedule when W_{1_2} fails.

Figure 3. Hybrid-parallel training across 12 workers with 3 data-parallel pipelines, each with 4 pipeline stages, and 6 micro batches. 3a shows a 1F1B fault-free schedule. 3b shows how *Adaptive Pipelining* re-routes micro-batches from failed worker W_{1_2} to its functional peers W_{0_2} and W_{2_2}.

back to worker W_{1_3} at time step 4 (❷). Similarly, the *forward pass* of micro-batch 10 is re-routed to worker W_{2_2} at time step 12 (❸). The *backward pass* operates similarly in the opposite direction. Worker W_{2_2} receives from worker W_{1_3} the gradients for micro-batch 12 at time step 22 (❹). The output gradients are passed to worker W_{1_1} at time step 24 (❺). The overall mathematical computation remains *unchanged* from the fault-free 1F1B schedule, ensuring that *Adaptive Pipelining* does not impact model convergence.

Are sufficient bubbles available? *Adaptive Pipelining* exploits the $3 \times (PP-1) \times DP$ existing but previously unused bubbles in each pipeline to recover from failures. For example, the 405 billion parameter LLaMA-3 is trained with synchronous 1F1B over 8192 GPUs using hybrid parallelism with tensor, pipeline, and data parallelism degree $TP = 8$, $PP = 16$, and $DP = 64$, respectively [3]. ReCycle can leverage $(3 \times (16-1) \times 64) = 2880$ bubbles per iteration to accommodate $\frac{2880}{3} = 960$ rerouted micro-batches per iteration. For a training job with global batch size of 2048, this is sufficient to handle 30 simultaneous failures.

The scheduling challenge: While sufficient bubbles exist, these bubbles are concentrated in the warm-up and cool-down phase of the 1F1B schedule. For example, there are 18 idle slots across data-parallel peers of failed node W_{1_2} in Figure 3a to accommodate its micro-batches. *Adaptive Pipelining* needs to reroute micro-batches to peer workers W_{0_2} and W_{2_2} mostly in the middle of their steady-state schedule, which is optimized to be bubble-free for efficiency [51]. As

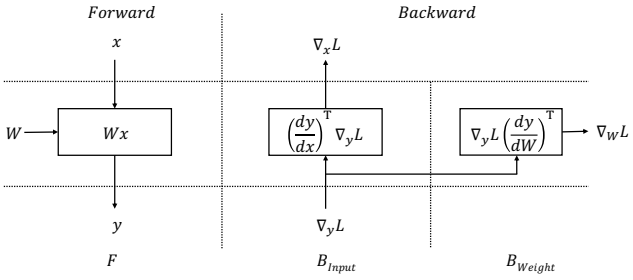


Figure 4. Forward and Backward pass for an operator.

shown in Figure 3b, this has a significant performance impact. The additional micro-batches to peer workers and the dependencies between forward and backward pass computations lead to 9 additional time steps per iteration, for a total of 36. In other words, the failure of 8.3% of workers (1 out of 12) leads to a 33% slowdown in iteration time.

3.2 Decoupled BackProp: Filling Unused Bubbles

Decoupled BackProp addresses the scheduling problem of *Adaptive Pipelining*. It separates the backward pass into two distinct phases, allowing for more flexible scheduling of the extra load in the peers of a failed worker. Figure 4 shows that the backward pass calculates two distinct outputs: the gradients relative to the input (B_{Input}) and the gradients relative to parameters (weights) for that pipeline stage (B_{Weight}). Conventionally, B_{Input} and B_{Weight} are *coupled* as a unified calculation. *This coupling lengthens dependencies between*

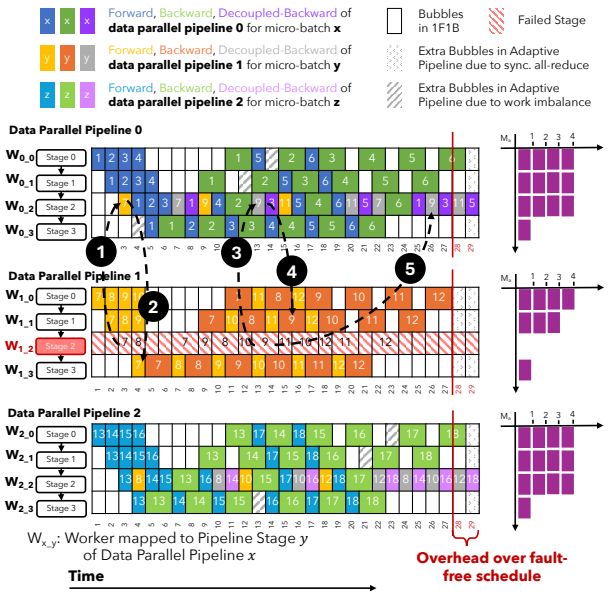


Figure 5. Optimized schedule with *Decoupled BackProp* when worker W_{1_2} fails.

pipeline stages. The backward pass for stage i must wait for the completion of both B_{Input} and B_{Weight} from stage $i+1$, despite stage i requiring only B_{Input} for its backward computations. B_{Weight} *can be deferred* to improve the overall pipeline schedule. Thus, *Decoupled BackProp* splits the backward pass into two distinct tasks.

Figure 5 explains how *Decoupled BackProp* reduces the overheads introduced by *Adaptive Pipelining*. It shows the same example as in Figure 3 with the addition of *Decoupled BackProp*. The forward pass for the micro-batch 7 is unchanged (❶ and ❷). However, instead of waiting on a *coupled* backward pass (both B_{Input} and B_{Weight}) from the previous stage to complete, which causes a ripple effect of dependencies throughout the pipeline, subsequent stages can continue processing new micro-batches using just the B_{Input} from their predecessors. For example, worker W_{0_2} can run the backward pass for micro-batch 9 (❸) and immediately pass the B_{Input} to worker W_{1_3} (❹) without computing B_{Weight}. B_{Weight} is deferred to a later time (❺) to reduce pipeline stalls, bringing the overall overhead down to just two time-steps (7.4% overhead with 8.3% failed workers).

Because B_{Weight} is dependence-free, it can largely be deferred until the end of the training iteration. Hence, we can take advantage of idle slots in the cool-down phase of the 1F1B schedule for B_{Weight} computations, freeing slots in the steady stage for the re-routed work of the failed worker. This enables ReCycle to absorb the additional computational demands imposed by failures with minimal overheads.

The memory challenge. Unfortunately, *Decoupled BackProp* increases memory pressure. Decoupling the computation of B_{Weight} requires that intermediate data is stored for potentially extended periods on each worker. For instance, as shown in Figure 5, decoupling the backward pass for micro batch 1 necessitates retaining its intermediate data until time step 25. To avoid memory exhaustion, ReCycle applies *Decoupled BackProp selectively* only when it can mitigate the overheads of *Adaptive Pipelining*. We also capitalize on the observation that memory imbalance exists among the pipeline stages [36]. To ensure that all pipeline stages are fully utilized, earlier stages need to allocate additional memory to process more forward micro-batches than later stages [51]. By exploiting this available *surplus* memory, ReCycle can effectively offset some of the memory demands of *Decoupled BackProp* without incurring additional costs.

3.3 Staggered Optimizer: Accessing More Bubbles

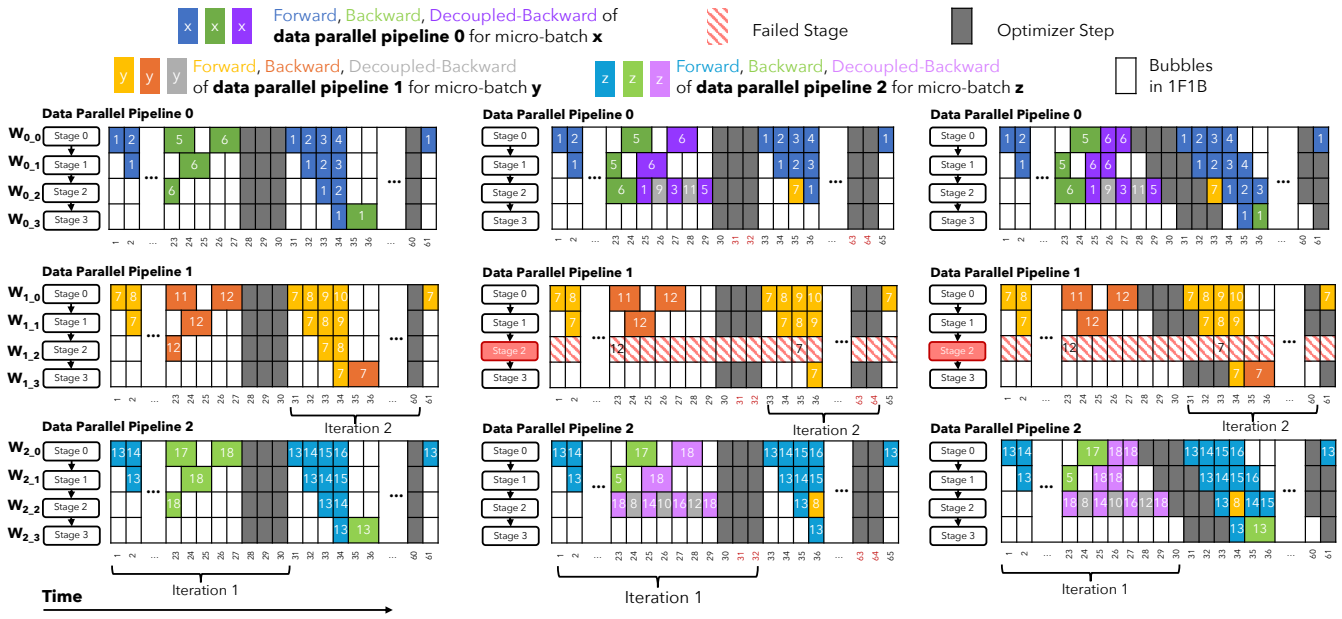
Decoupled BackProp makes efficient use of the cool-down bubbles. However, as seen in Figure 6, the warm-up bubbles remain underutilized due to their placement *after* the synchronous optimizer step of the previous iteration. To exploit warm-up bubbles, we make the critical observation that optimizer steps for different pipeline stages are independent of each other. The *Staggered Optimizer* thus shifts the timing of the optimizer step across pipeline stages to better utilize the bubbles that occur during the warm-up phase of the subsequent training iteration. Effectively, this staggering provides ReCycle *more* bubbles in the cool-down phase to hide the overhead of compensating for failed workers.

Figure 6c shows how combining *Staggered Optimizer* with *Adaptive Pipelining* and *Decoupled BackProp* results in *zero overhead* over the fault-free 1F1B schedule in the running example. By staggering the optimizer step, later pipeline stages can move bubbles from the warm-up phase of the second iteration to the cool-down phase of the first iteration. Thus, peers that need to process additional micro-batches from failed workers within their steady-state schedule (i.e, workers W_{0_2} and W_{2_2}) can further defer their weight gradient computations towards the cool-down phase. Workers for earlier stages (e.g., W_{0_0} for stage 0) can continue with the optimizer step, ensuring that the entire pipeline does not stall. Worker W_{0_0} can start iteration 2 at *exactly the same time step as the fault-free 1F1B schedule*, even with failures.

Put together, these three mechanisms allow ReCycle to flexibly optimize the use of computational resources across pipelines, ensuring that each stage operates near its capacity and maintains consistent utilization throughout training.

3.4 Supporting Multiple Failures and Re-Joins

ReCycle supports multiple failures and rejoins. ReCycle guarantees continued training up to $DP-1$ simultaneous worker failures, as in the worst case DP failures can disable all the data-parallel peers of a pipeline stage. However, ReCycle can probabilistically sustain more failures, as long as there is at least one functional worker per stage across all data parallel pipelines. Figure 7b shows how ReCycle can recover from

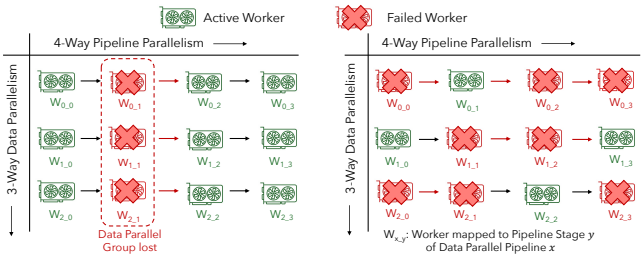


(a) Fault-free 1F1B Schedule

(b) Schedule for Adaptive Pipelining with Decoupled BackProp when worker W_{1_2} fails.

(c) Schedule with the addition of the Staggered Optimizer.

Figure 6. The Staggered Optimizer allows ReCycle to optimize pipeline schedule across training iterations.



(a) The loss of an entire DP group requires checkpoint restoration.

(b) ReCycle can continue training if a functional worker exists per stage.

Figure 7. Example of ReCycle's fault tolerance guarantees.

more than $DP - 1$ failures. Despite 8 simultaneous failures – *2/3 of the GPUs!* – ReCycle ensures continual training.

As workers are repaired, ReCycle can re-insert them back into the adapted pipeline schedule, reducing the probability of high counts of concurrent failures. Unlike unexpected failures, planned worker additions occur at iteration boundaries to overlap any data copying overhead with the previous iteration. Once a worker rejoins, ReCycle ceases rerouting micro-batches and sends all micro-batches from the previous stage to the recovered worker.

In the worst-case scenario, where simultaneous failures impact a single data-parallel group (as shown in Figure 7a), ReCycle falls back on existing resilience strategies: it calculates an efficient hybrid-parallel scheme for the remaining nodes, configures it across all workers by restoring from a recent checkpoint, and resumes training at the speed supported by the new parallelism.

Multi-GPU Servers and Tensor Parallelism. Training systems commonly employ servers equipped with multiple GPUs. A favoured configuration features HGX/DGX servers with 8 GPUs connected in an all-to-all manner using NVLink and NVSwitches [18]. Similar to Oobleck and Bamboo, we treat an entire server as the unit of failure. This approach is pragmatic because servicing even one faulty GPU board necessitates taking the entire server offline. Additionally, issues such as software malfunctions, CPU issues, connectivity disruptions, and power or cooling issues can incapacitate an entire server. Most training systems implement tensor parallelism within a multi-GPU server to leverage the high-bandwidth NVLink connections. However, a tensor parallel group can extend across multiple servers. In such cases, ReCycle, along with Oobleck and Bamboo, classifies the entire group of tensor-parallel servers as a single unit of failure.

4 ReCycle Design

4.1 System Overview

ReCycle consists of three key components, the *Profiler*, the *Executors*, and critically the *Planner*, as shown in Figure 8.

Profiler. When a large training job is first submitted, ReCycle runs a short profiling job to collect key performance

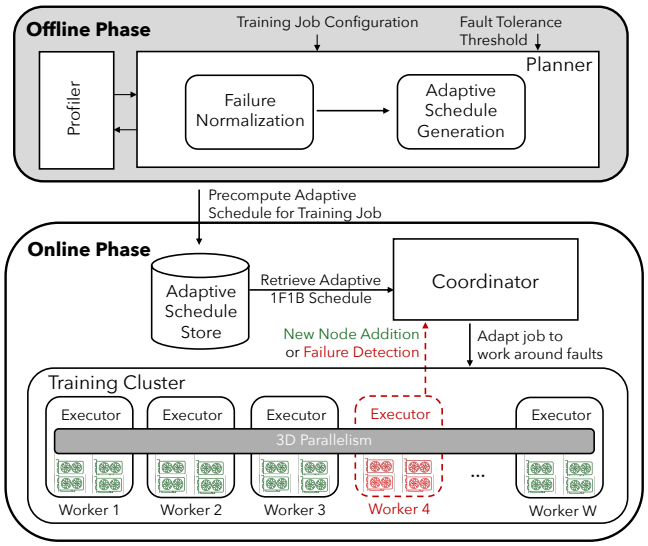


Figure 8. ReCycle system overview.

statistics such as the average micro-batch latency for the forward and backward passes, memory requirements for activations and gradients, and inter-node communication bandwidth. The profiling job executes a small number of training iterations, 100 by default, and typically takes a few minutes. These statistics are used by the *Planner*.

Planner. The *Planner* generates pipeline schedules for the training job under various failure scenarios. It uses dynamic programming and mixed integer linear programming (MILP) to implement the ReCycle techniques (Section 3). We generate a pipeline schedule for each *number* of tolerated simultaneous failures. Each plan is agnostic to which specific worker(s) fail. It encodes the specific pipeline schedule, specifying micro-batch assignments, the sequence of forward and backward tasks, and communication of activations and gradients. The plans are stored in distributed fault-tolerant storage (e.g., *etcd*) to be used by runtime *Executors*.

By default, we generate up to $DP-1$ plans, which accounts for the $DP-1$ simultaneous failures that ReCycle can always handle. Since ReCycle can probabilistically sustain many more failures, we can also generate plans for up to a user-defined fault tolerance threshold.

Executors. ReCycle operates with an *Executor* on each GPU node, tasked with implementing the training plan specific to that node. Overseeing these Executors, a centralized *Coordinator* orchestrates the overall training operation by distributing the appropriate execution plans to each Executor. Additionally, the *Coordinator* actively monitors the status of all Executors to ensure their continuous operation. In the event of a failure, the *Coordinator* reassigns the tasks impacted by failure to the most suitable location, as advised by the *Planner* (refer to Section 4.2.1), and then directs the remaining *Executors* to resume training according to the updated plan that now reflects the updated count of non-operational workers. Training resumes from the iteration during which the failure was identified. If failures are not promptly detected, restoring a checkpoint from remote storage may become necessary. To enhance reliability, the *Coordinator* itself is safeguarded against failures through active replication.

4.2 Planner

Given the cluster configuration, training job, profiling statistics, and failed workers, the *Planner* calculates an adaptive schedule that minimizes the training iteration latency using the fault-free workers. It does this in two phases, *Failure Normalization* followed by *Adaptive Schedule Generation*.

To avoid solving an MILP for the combinatorial number of possible failure locations, the *Failure Normalization* phase first normalizes each given *number* of failures by calculating the suitable location within the pipeline schedule to migrate each failure. For example, if *any* single node fails in the example in Figure 7, the *Planner* will swap the node with the calculated ideal location, e.g., W_{2_3}. Unlike current solutions which require a complete reconfiguration of a data-parallel pipeline [29], normalization only requires a point-to-point copy of model parameters to swap the location of two workers in the pipeline for each failure. The *Adaptive Schedule Generation* then leverages a MILP to derive an adaptive pipeline schedule for each normalized case.

4.2.1 Failure Normalization. The intuition behind *Failure Normalization*, shown in Algorithm 1, is twofold: a) distribute failures across different peer groups to enhance fault tolerance and evenly balance the additional workload, and b) shift failures to later pipeline stages with more bubbles to maximize the opportunity to hide re-routing overheads.

Failure Normalization uses dynamic programming to compute a migration strategy for handling a given number of failures. Given a total of F failures and PP pipeline stages, it returns a list A of length PP, where the sum of elements in A equals F. Each $A[i]$ specifies the number of failures assigned to pipeline stage i across all data parallel pipelines, with the specific pipeline assignments being arbitrary and not impacting performance. For example, if $A[3] = 2$, two failures would be assigned to stage 3, and we could swap the failed nodes with any two from that stage, such as W_{0_3} and W_{2_3} in Figure 7b.

The overhead of handling f failures at stages from 0 to i is represented by $O[i][f]$, while $A[i][f]$ indicates how the f failures are distributed across these stages, with $A[i][f]$ being of length $i+1$. The recurrence relation minimizes overhead by finding the optimal failure assignment for f failures across the first i stages, and the best assignment for F failures is given by $A[PP-1][F]$.

To compute $COST(i, x)$, we estimate the additional time slots needed for handling extra micro-batches due to x failures in the i-th stage using the heuristic from line 27. While

Algorithm 1 Failure Normalization

$DP \leftarrow$ Number of data-parallel pipelines.
$PP \leftarrow$ Number of pipeline stages.
$MB \leftarrow$ Number of microbatches per pipeline.
$F \leftarrow$ Total number of failures.
$O \leftarrow$ an PP × (F+1) array ▷ Rerouting Overheads
$A \leftarrow$ an PP × (F+1) array ▷ Assignments

procedure FAILURE_REORDERING(DP, PP, MB, F)
  for $i \in \{0, \ldots, PP-1\}$ **do**
    for $f \in \{0, \ldots, F\}$ **do**
      if $i == 0$ **then**
        $O[i][f] = \text{COST}(f)$
        $A[i][f] = [f]$
      else
        $x = \arg\min_{x \leq f} \Big(O[i-1][f-x] + \text{COST}(x) \Big)$
        $O[i][f] = O[i-1][f-x] + \text{COST}(x)$
        $A[i][f] = concat\Big(A[i-1][f-x], x\Big)$
      end if
    end for
  end for
return $A[PP-1][F]$
end procedure

procedure COST(f)
  if $f > 0$ **then**
    return $\min\Big(0, MB \times f \times 3 - (DP-f) \times (PP-1) \times 3\Big)$
  end if
  return 0
end procedure

the MILP in Section 4.2.2 could be used, *Failure Normalization* opts for the heuristic to reduce computation time. The complexity of determining the migration strategy for F failures and PP pipeline stages is $O(PP \times F)$.

4.2.2 Adaptive Schedule Generation.

Next, the *Planner* uses normalized failure locations and profiled statistics as inputs for an MILP to generate adaptive schedule. At a high level, the MILP determines how to re-route micro-batches across peers of a failed worker, integrating both regular and re-routed micro-batches (*Adaptive Pipelining*). It considers communication latency T_{comm}, micro-batch latency for forward, backward input and weight pass – T_F, $T_{B_{Input}}$ and $T_{B_{Weight}}$ respectively, task dependencies in forward and backward passes, and memory usage, leveraging the *Decoupled BackProp* and *Staggered Optimizer* techniques.

Notation. Each operation in a training iteration is denoted by the 5-tuple (i, j, k, c, k_s). Here, i represents the pipeline stage within each data parallel pipeline, and k indicates the operation's original data parallel pipeline before any failures. The variable j denotes the micro-batch ID in the training iteration, and c specifies the type of operation for each micro-batch, where $c \in \{F, B_{input}, B_{weight}\}$. Finally, k_s specifies the peer pipeline that executes a micro-batch originally intended for k; which can be the same as k. For instance, a micro-batch ID 14 originally scheduled for W_{2_3} but rerouted to peer W_{1_3} would be identified with $i = 3$, $j = 14$, $k = 2$, and $k_s = 1$.

Inputs. We use micro-batch assignments for each worker as inputs. The *Planner* reassigns micro-batches from failed workers to peer workers within the same group while retaining the original micro-batches. This results in a binary mapping $S^{k_s}_{i,j,k} \in \{0, 1\}$, indicating whether a micro-batch (i, j, k) should run on pipeline k_s. Each operation is assigned to exactly one pipeline: $\sum_{k_s} S^{k_s}_{i,j,k} = 1$.

Additionally, we use profiled statistics, which include T_c for the computational time of each operation c, T_{comm} for communication latency of activations or gradients[1], and $\Delta M^{k_s}_{i,j,k,c}$ for the *change* in memory utilization on worker (i, k, k_s) due to the execution of operation (i, j, k, c, k_s).

$$\Delta M^{k_s}_{i,j,k,c} = \begin{cases} A_B & \text{, if } c = F \text{ and } S^{k_s}_{i,j,k} = 1 \\ A_B - A_{B_{Input}} & \text{, if } c = B_{Input} \text{ and } S^{k_s}_{i,j,k} = 1 \\ -A_{B_{Weight}} & \text{, if } c = B_{Weight} \text{ and } S^{k_s}_{i,j,k} = 1 \\ 0 & \text{, otherwise} \end{cases}$$

Here, A_B is the profiled size of activation at the end of the forward pass. $A_{B_{input}}$ and $A_{B_{weight}}$ are the sizes of gradients at the end of the backward-input and backward-weight passes, respectively. We free $A_{B_{input}}$ and $A_{B_{weight}}$ after completing their respective backward passes.

Variables. We define a binary variable $O_{(i,j,k,c,k_s) \to (i',j',k',c',k'_s)} \in \{0, 1\}$ to represent ordering between pair of operations (i, j, k, c, k_s) and (i', j', k', c', k'_s). This variable is 1 if (i', j', k', c', k'_s) is scheduled after (i, j, k, c, k_s), and 0 otherwise. Ordering is needed only within a stage and between computation phases of the same micro-batch (e.g., F and B_{input}). Additionally, $E^{k_s}_{i,j,k,c}$ represents the ending time of operation (i, j, k, c, k_s).

Objective. Our objective is to minimize the makespan of a single training iteration while adhering to task dependencies and memory constraints. This involves determining the sequence and timing of operations throughout the training pipeline using the sets of variables O and E. Thus, the objective can be formulated as follows:

$$\min_{O,E} \quad \max_{i,j,k,k_s} E^{k_s}_{i,j,k,B_{Weight}} \tag{1}$$

Here, $\max_{i,j,k,k_s} E^{k_s}_{i,j,k,B_{Weight}}$ represents the end time of the *last* operation in the iteration, representing the makespan.

[1]We use a single value as activations and gradients are the same size.

Constraints. The constraints applied to the objective are the following.

Cross-Stage Dependencies.

$$E^{k_s}_{i,j,k,F} \geq S^{k_s}_{i,j,k} \times (\sum_{\hat{k}} (E^{\hat{k}}_{i-1,j,k,F} \times S^{\hat{k}}_{i-1,j,k}) + T_{comm} + T_F) \quad (2)$$

$$E^{k_s}_{i,j,k,B_{Input}} \geq S^{k_s}_{i,j,k} \times (\sum_{\hat{k}} (E^{\hat{k}}_{i+1,j,k,B_{Input}} \times S^{\hat{k}}_{i+1,j,k}) + T_{comm} + T_{B_{Input}}) \quad (3)$$

Equation 2 specifies the dependency of a given micro-batch's forward pass on previous pipeline stages (e.g., stage 0 must execute before stage 1). Similarly, Equation 3 specifies the reverse dependency for the backward pass (e.g., stage 1 must execute before stage 0).

Same-Stage Dependencies.

$$E^{k_s}_{i,j,k,B_{Weight}} \geq S^{k_s}_{i,j,k} \times (E^{k_s}_{i,j,k,B_{Input}} + T_{B_{Weight}}) \quad (4)$$

Equation 4 allows the MILP to reason about *Decoupled BackProp*, specifying that B_{input} must precede B_{weight} for a given micro-batch within a worker.

No Overlapping Computations.

$$E^{k'_s}_{i,j',k',c'} \geq E^{k'_s}_{i,j,k,c} + T_{c'} - \infty(1 - S^{k'_s}_{i,j,k} \times S^{k'_s}_{i,j',k'} + O_{(i,j,k,c,k'_s)\rightarrow(i,j',k',c',k'_s)}) \quad (5)$$

Equation 5 adds a dependency constraint which specifies that two different operations cannot overlap in time if they are executed on the same worker. For example, if operation (i, j', k', c', k'_s) is scheduled after (i, j, k, c, k'_s), they both execute on worker (i, k'_s) and thus (i, j', k', c', k'_s) must begin after (i, j, k, c, k'_s) ends.

Memory Constraint.

$$M_{Limit} \geq \Delta M^{k'_s}_{i,j',k',c'} + \sum_{j,k,c} \Delta M^{k'_s}_{i,j,k,c} \times O_{(i,j,k,c,k'_s)\rightarrow(i,j',k',c',k'_s)} \quad (6)$$

Finally, Equation 6 calculates the activation memory required at any given time on worker (i, k'_s), and constrains operations such that the total memory is below an M_{limit}.

5 Implementation

We implemented ReCycle on top of DeepSpeed [60] and made the following additions to DeepSpeed to support ReCycle.

Detecting failures. ReCycle uses existing hardware error detection mechanisms in GPUs (e.g.; SMBPBI APIs in NVIDIA GPUs) and software error detection in runtime systems (e.g. stderr logs in PyTorch). Workers send periodic heartbeats to a central driver, including worker information and hardware statistics from GPUs such as page retirement count, row-remapping stats, ECC correction stats, XID error logs, and stdout/stderr logs [33]. If the central driver detects abnormalities, it marks the worker as failed.

Rerouting micro-batches to data parallel peers. To handle the dynamic rerouting of micro-batches following node failures, we introduce two new and complementary communication operators: `ReRouteAct` and `ReRouteGrad`. Positioned at the end of a pipeline stage, `ReRouteAct` normally acts as a pass-through, transmitting computed intermediate activations to the next stage within the same data parallel pipeline. Conversely, `ReRouteGrad`, located at the beginning of a stage, typically forwards gradients backward through the pipeline. When a subsequent stage fails, `ReRouteAct` redistributes the micro-batches across the remaining peers in a round-robin fashion, while `ReRouteGrad` adjusts the gradient distribution to ensure that both the forward and backward processes of a micro-batch are handled by the same peer. This strategy maintains operational continuity in ReCycle under fault conditions, mirroring fault-free execution semantics. These operators are integrated as *pipeline instructions* within the DeepSpeed execution engine.

Decoupling Back Propagation in DeepSpeed. Our implementation of *Decoupled BackProp* within DeepSpeed centers around intercepting the weight gradient computations traditionally performed during the backpropagation phase. These computations are temporarily held in a newly created in-memory structure known as *WeightGradStore*, enabling ReCycle to defer weight gradient computation. To facilitate this, two new *pipeline instructions* have been introduced into DeepSpeed's execution engine: `InputBackwardPass` and `WeightBackwardPass`. These instructions are incorporated into ReCycle's execution plan and are then processed by DeepSpeed's execution scheduler, effectively managing the distinct phases of backpropagation.

Bypassing Optimizer Synchronizations. In DeepSpeed, each training iteration ends with an all-reduce collective to synchronize gradients across data-parallel peers, followed by validation checks at each pipeline stage to ensure numerical stability. If any stage detects a potential issue, the optimizer step for that iteration is skipped to prevent further problems. Upon successful validation, the optimizer step is executed, allowing synchronized progression to the next iteration.

In contrast, ReCycle uses a staggered timing approach for optimizer steps across different pipeline stages to improve scheduling efficiency. This method of staggering is not conducive to cross-stage synchronization for validating numerical stability prior to executing the optimizer. To address this, ReCycle shifts numerical validations from a pre-step to a post-step process. After the all-reduce collective, each stage performs its own local validation checks without waiting for downstream stages. Each stage executes its optimizer step based on its own validation results and those of preceding stages. If a downstream stage fails validation, a rollback occurs across all stages before moving to the next iteration. Notably, for many optimizers, including the commonly used AdamW [47], these rollbacks incur no additional memory costs due to the arithmetic reversibility of the operations.

Table 1. Training throughput (samples/sec) with increasing failure frequency, higher is better. Bamboo ran out of memory for GPT-3 3.35B and 6.7B.

Systems	GPT-3 Medium			GPT-3 3.35B			GPT-3 6.7B		
Failure Frequency	**6h**	**2h**	**30m**	**6h**	**2h**	**30m**	**6h**	**2h**	**30m**
Fault-Free DeepSpeed [60]		27.58			14.87			5.33	
Bamboo [67]	19.47	18.98	15.24	OOM	OOM	OOM	OOM	OOM	OOM
Oobleck [29]	27.26	25.37	19.47	14.55	13.44	9.78	4.98	4.65	2.78
ReCycle	27.27	25.42	22.27	14.59	14.17	12.63	5.17	4.85	3.53

This adjustment enables ReCycle to leverage staggered operations while safeguarding the training process from potential instabilities.

6 Evaluation

6.1 Experimental Setup

Cluster Setup. We conducted real-world experiments on a 32-GPU cluster featuring NVIDIA A100 80GB GPUs in Azure, utilizing Standard_NC96ads_A100_v4 instances (8 GPUs, 96 vCPUs, and 880 GB memory each). Each node includes a 600 GB/s NVLink intra-node interconnect and a 640 Gbps inter-node interconnect across 8 NICs. Scalability experiments were conducted using a simulator, discussed in Section 6.3.

Baselines. We evaluated ReCycle against two state-of-the-art baselines discussed in Section 2.2.3 – Bamboo [67] and Oobleck [29]. Both baselines were tested with all their optimizations enabled. Additionally, we report the *fault-free* throughput achieved by DeepSpeed [60] using a 1F1B pipeline schedule [51]. All experiments were conducted on the same Azure cluster for consistency.

Workloads. We evaluated all systems using the Megatron [64] implementation of GPT-3 [10], with three model sizes: Medium (350M), 3.35B, and 6.7B. The applied (PP, DP) degrees were (2, 16), (4, 8), and (8, 4), respectively, with a TP degree of 1 for all models. Training was conducted on wikitext [49] with batch and micro-batch sizes of (8192, 8) for Medium, and (1024, 1) for both 3.35B and 6.7B models. All real-world experiments, unless stated otherwise, ran for 6 hours with 32 workers.

6.2 Training Throughput Under Failures

How well does ReCycle handle failures compared to baselines? We first evaluate the average training throughput of Bamboo, Oobleck and ReCycle on various failure scenarios. We set the frequency of failures from once every 6 hours to once every 30 minutes to cover a wide spectrum of environments [30, 74]. We monotonically reduce the number of available workers without recovery, so the total number of workers steadily decreases over the course of each experiment – for example in the 30m case, only 62.5% of the workers (20 out of 32 workers) remain at the end of training.

Table 1 presents the average throughput for various failure frequencies and model sizes. Bamboo suffers from static overhead due to redundant computations and additional model state copies, which quickly depletes GPU memory. This leads to its inability to train larger models and a significant drop in throughput for even the smallest model—resulting in a 29% reduction in throughput in the 6-hour failure frequency scenario. In contrast, Oobleck effectively manages all model sizes and demonstrates throughput improvements over Bamboo. However, Oobleck's performance declines with increasing failure frequency and model size due to imbalanced heterogeneous pipelines and higher reconfiguration latency.

ReCycle effectively continues training despite failures, consistently matching or exceeding the throughput of both baselines across all models and failure rates. Unlike Bamboo, the techniques utilized in ReCycle—*Adaptive Pipelining*, *Decoupled BackProp*, and *Staggered Optimizer*—introduce no static overhead, enabling uninterrupted training for larger models and increasing throughput by up to 1.46× for GPT-3 Medium. Additionally, ReCycle has reduced reconfiguration overhead; during failure normalization, it requires parameter migration for at most one GPU, compared to the need for reconfiguring an entire data pipeline with Oobleck. This efficiency results in up to 1.29× higher training throughput.

What is ReCycle's throughput advantage in dynamic training scenarios? We next assessed ReCycle's performance in a dynamic training scenario characterized by GPU failures and re-joins, using a real-world failure trace. This trace, derived from GCP instances utilized by Bamboo [67] and Oobleck [29], was replayed over a 6-hour training run on our Azure cluster. While we observed similar outcomes with Bamboo's AWS trace, we opted to omit those plots due to space limitations. During the experiment, we had 24 GPUs available instead of the originally planned 32. Figure 9a illustrates the fluctuating number of available GPUs over time, ranging from a maximum of 24 to a minimum of 15. Unlike the previous experiment, this scenario frequently saw GPUs being removed and reintroduced into the cluster.

In Figure 9b and 9c, we present the training throughput achieved by Bamboo, Oobleck, and ReCycle while replaying the trace for both GPT-3 Medium and GPT-3 6.7B training

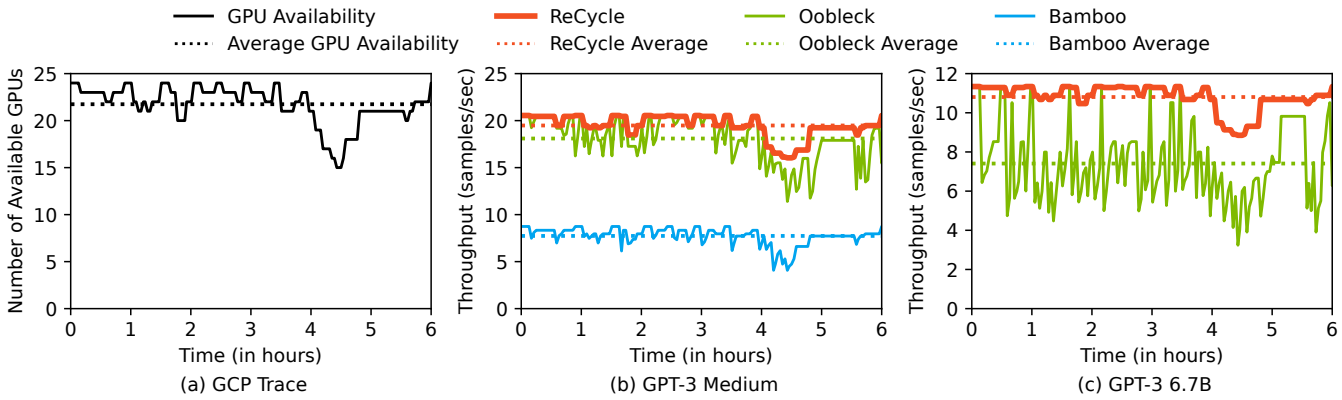


Figure 9. Training throughput (samples/sec), higher is better, for the GPT-3 Medium and GPT-3 6.7B models over the GCP trace. In 9b and 9c, the dashed lines represent the average training throughput achieved by each system within the 6h period.

Table 2. Gap between real-world and simulated throughput across various models and failure rates.

Models	Fault-Free	6h	2h	30m
GPT-3 Medium	-0.87%	+5.98%	-1.93%	-1.48%
GPT-3 3.35B	-0.13%	-1.58%	+2.12%	-1.90%
GPT-3 6.7B	+3.94%	+2.71%	-1.86%	-0.85%

jobs. The solid lines indicate instantaneous training throughput, while the dashed lines represent the average training throughput for each system throughout the 6-hour period. Notably, Bamboo is unable to train GPT-3 6.7B due to memory constraints. ReCycle demonstrates a performance increase of 1.64× in average throughput compared to Bamboo on GPT-3 Medium, and a 1.46× improvement over Oobleck on GPT-3 6.7B. As mentioned earlier, Bamboo incurs overhead from redundant computations, while Oobleck experiences significant stalls due to parameter re-shuffling during GPU failures and re-joins, leading to substantial drops in throughput. Additionally, during stable periods, Oobleck grapples with imbalanced heterogeneous pipelines. In contrast, ReCycle consistently delivers the highest and most stable training throughput throughout the entire trace.

6.3 ReCycle Scalability

Due to the unavailability of a cluster with thousands of GPUs, we developed a simulator capable of calculating the training throughput based on a model and cluster configuration, given a specific execution plan. The simulator leverages real-world profiled statistics for each pipeline operation associated with the respective model. To validate the simulator's accuracy, Table 2 presents the differences between the simulated throughput and the measured real-world throughput on Azure across various failure rates for three GPT-3 models, with a maximum discrepancy of 5.98%. These variations are mainly due to minor fluctuations in the execution time of

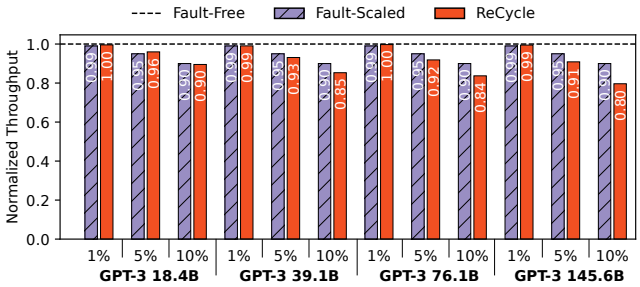


Figure 10. Simulated throughput of ReCycle as model size increases, normalized to the simulated fault-free 1F1B throughput. The Fault-Scaled throughput is the fault-free throughput scaled by the percent of non-failed GPUs.

NCCL collectives, but they have little impact on ReCycle's performance.

How effectively does ReCycle scale to large clusters and models? Using the simulator, we report the training throughput for GPT models with sizes of 18.4B, 39.1B, 76.1B, and 145.6B across different clusters: (256 GPUs, 8 stages per pipeline, 32 pipelines), (512 GPUs, 16 stages per pipeline, 32 pipelines), (1024 GPUs, 32 stages per pipeline, 32 pipelines), and (1536 GPUs, 64 stages per pipeline, 24 pipelines). Figure 10 presents the throughput normalized to the fault-free throughput achieved by DeepSpeed's 1F1B schedule. Additionally, we introduce the *fault-scaled* throughput, calculated by multiplying the fault-free throughput by the percentage of operational GPUs. Notably, we focus on steady-state throughput for scenarios with 1%, 5%, and 10% GPU failures, rather than the throughput during periods of GPU failures and re-joins.

ReCycle demonstrates a strong capability to maintain high throughput across varying failure rates and model sizes. For a failure rate of 1%, ReCycle manages failures at or better than the fault-scaled throughput for all models and cluster sizes,

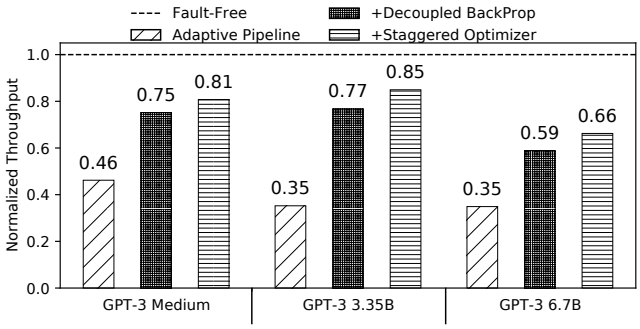


Figure 11. The contribution of the three optimization techniques to ReCycle's training throughput.

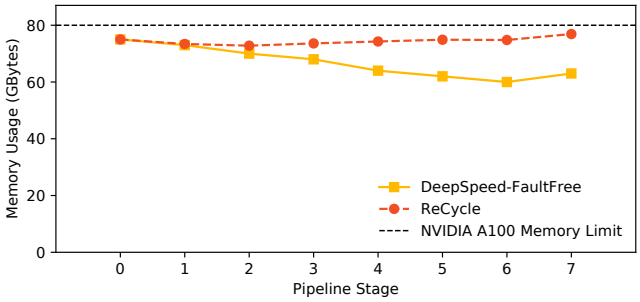


Figure 12. Memory utilization for ReCycle across pipeline stages for GPT-3 6.7B with 30m failure rates compared to DeepSpeed (failure-free). The dashed line shows the 80GB memory capacity of the A100 GPU.

frequently exhibiting no degradation in performance. The bubbles in peer workers are more than adequate to efficiently handle the workload of failed GPUs utilizing ReCycle's optimizations (see Figure 6). As the number of failed GPUs increases, ReCycle continues to sustain high throughput levels. At a 5% failure rate, ReCycle's performance is comparable to the fault-scaled throughput. Even at a significant 10% failure rate (e.g., 154 failed GPUs for GPT-3 145.6B), ReCycle allows clusters to continue training while observing only between 0.5% and 11.5% degradation from the fault-scaled case. This indicates that ReCycle is well-suited for training tasks in supercomputing-scale clusters with dynamic resource availability [24, 33, 79].

6.4 ReCycle Performance Breakdown

How does each of ReCycle's techniques contribute to performance? To assess the benefits and necessity of each technique in ReCycle, as outlined in Section 3, we conducted an ablation study. This involved repeating the real-world experiment on the Azure cluster with GPT-3 models, using a failure frequency of 30 minutes. We adjusted the MILP

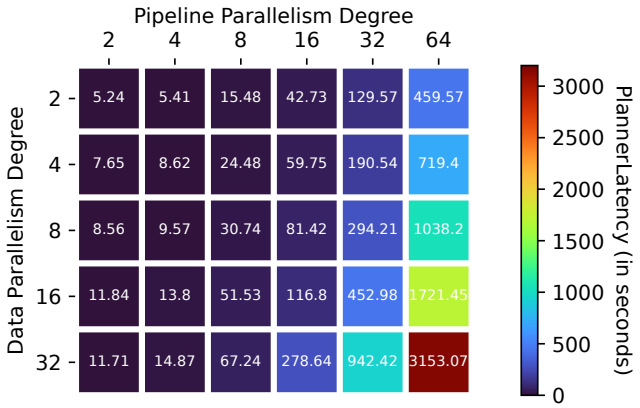


Figure 13. ReCycle *Planner* latency (in seconds) to find optimized schedules for up to 25% of failed GPUs.

formulation in the *Planner* to progressively enable each optimization. Figure 11 shows the throughput achieved at each stage of the study, normalized to the fault-free throughput.

Adaptive Pipelining allows ReCycle to continue training despite failures, but it experiences significant throughput degradation due to the overhead from additional work required by peers of the failed workers. By introducing *Decoupled BackProp*, ReCycle boosts the training throughput by 63% to 118%, effectively utilizing bubbles during the cooldown phase of the pipeline schedule to mask this overhead (see Figure 3b). Further enhancements come from the *Staggered Optimizer*, which boosts throughput by an additional 7% to 11% by leveraging bubbles from the warm-up phase of the pipeline schedule.

Can ReCycle efficiently exploit unused GPU memory in hybrid parallelism? We recognize that GPU memory utilization varies across different stages in hybrid parallelism, with later pipeline stages typically requiring less memory. To leverage this, the *Decoupled BackProp* technique utilizes the available surplus GPU memory as a buffer, allowing for the postponement of B_{weight} computations (see Figure 5). Figure 12 shows the peak GPU memory utilization across the eight stages of pipelined execution for ReCycle compared to the fault-free DeepSpeed, based on training the GPT-3 6.7B model with a 30-minute failure interval.

DeepSpeed – and any system utilizing a pipeline schedule akin to 1F1B – fails to fully utilize GPU memory, especially in the later pipeline stages [51]. This surplus arises because GPU workers in earlier stages must store more intermediate results, as shown in Figure 3. In contrast, ReCycle effectively capitalizes on this opportunity, achieving near-complete utilization of GPU memory to optimize adaptive pipelined execution in the presence of failures. The *Planner*'s MILP formulation is tailored to model and manage these additional memory requirements while adhering to GPU memory constraints, thereby preventing memory exhaustion.

What is the overhead of ReCycle's Planner? To evaluate the latency of the *Planner* in generating adaptive schedules for large clusters, Figure 13 shows the time required to generate all necessary schedules for up to 25% node failures across various hybrid parallelism strategies. Users have the flexibility to configure the fault-tolerance threshold to suit their needs, whether smaller or larger. The reported latency encompasses both phases of the *Planner* and was measured using the Gurobi MILP solver [23] on a 96-core CPU.

In a training setup with 2048 GPUs (with $DP = 32$ and $PP = 64$), the *Planner* can generate all adaptive schedules to accommodate up to 512 failures in just 3153 seconds (or 52.5 minutes). Considering that training large models on thousands of GPUs typically takes weeks [33, 53, 65], the *Planner*'s latency is negligible, accounting for less than 0.1% of the overall training time.

7 Related Work

Parallel Training. Data parallelism [12, 16, 39] is a widely utilized mode of parallelism that distributes the dataset across different partitions for processing. In this mode, the learned weights are synchronized via either an all-reduce approach [12] or by using parameter servers [13, 32, 42]. Alternatively, model parallelism [38, 63, 64] involves distributing the components of a deep neural network (DNN) model across multiple GPU devices, allowing each device to handle a specific subset of the model's parameters for all input data. Recently, pipeline parallelism [27, 36, 43, 46, 51] has emerged as a technique for training large models by dividing the model's layers among different workers and utilizing micro-batches to efficiently use the available computational resources. Prominent deep learning training frameworks like PyTorch [1], DeepSpeed [60], and Megatron [52] have adopted hybrid-parallelism, an approach that integrates data parallelism, model parallelism, and pipeline parallelism. This integration facilitates training on a massive scale while enhancing computational and memory efficiency. Additionally, DeepSpeed introduces ZeRO-style [37, 58, 59, 61, 77] data parallelism, which strategically partitions model states across GPUs, coordinating via communication collectives to synchronize parameters as needed.

Optimizing Hybrid-Parallel Training. Improvements in hybrid parallelism are extensively studied in the context of ML training [27, 36, 37, 43, 44, 46, 51, 52, 60, 77, 78]. Alpa [78] automatically optimizes inter- and intra-operator parallelism using a hierarchical ILP formulation. Tofu [70] employs dynamic programming to optimally partition tensor operations in a single node. FlexFlow [31] uses a randomized search algorithm to quickly find parallelism strategies. TensorOpt [11] introduces a dynamic programming approach capable of optimizing parallelism across multiple resource dimensions, including memory and compute. Piper [66] proposes a two-level dynamic programming algorithm to find optimal hybrid parallelism strategies. In contrast, ReCycle leverages a dynamic programming algorithm and mixed-integer linear programming (MILP) to exploit key functional redundancies in hybrid parallelism, explicitly enhancing training throughput in the presence of failures.

Elastic Training. Elastic training systems can dynamically adjust the resources allocated to a training job. Both Horovod [62] and Torch Distributed [44] offer mechanisms to modify the number of workers, but require either a restart from a checkpoint or an expensive re-shuffling of model parameters. CoDDL [28], Optimus [57], OASiS [6], and Themis [48] are ML cluster schedulers that can dynamically allocate resources across multiple DNN training jobs. Or et al. [55] autoscale the number of workers for a training job. Varuna [4] provides elastic training by leveraging spot instances, but requires restarts from checkpoints to handle unexpected preemptions. While some elastic training systems can affect model consistency by changing critical hyperparameters such as batch size and learning rate, ReCycle guarantees mathematical consistency of operations, regardless of the number of failures.

Fault-tolerant Training. DNN training systems commonly use checkpoints for fault recovery [7, 24, 30, 33, 40, 60, 71, 79], though naive checkpointing can cause stalls. Recent works like CheckFreq [50], Check-N-Run [19], and Gemini [73] reduce overheads by adapting checkpoint frequency, quantizing embedding tables, and scheduling checkpoint traffic across the storage hierarchy, respectively. Megascale [33] alleviates the storage bottleneck during recovery by sharing data between corresponding GPU workers across data parallel groups.

Two recent projects have proposed to utilize pipeline parallelism for efficient training in the presence of failures without spares. Bamboo [67] introduces redundant computation (RC) to provide resilience in the presence of frequent preemptions of training with spot instances. Oobleck [29] provides resiliency through creation of heterogeneous pipelines. ReCycle matches or outperforms them for a wide range of model sizes and failure frequencies.

8 Conclusion

ReCycle enables efficient DNN training in the presence of failures without relying on spare resources. By leveraging the functional redundancies inherent in hybrid parallel training systems for large DNNs, ReCycle introduces innovative techniques to minimize the degradation in training throughput caused by failures. It does this by utilizing unused resources, such as pipeline bubbles. We evaluated an end-to-end prototype of ReCycle and demonstrated its ability to tolerate a high number of concurrent failures across systems and models of varying sizes. Compared to Oobleck and Bamboo, ReCycle improves training throughput under failure conditions by up to 1.46× and 1.64× respectively.

Acknowledgements

We are grateful to the anonymous reviewers and to our shepherd, Laurent Bindschaedler, whose comments have greatly helped improve this paper. This research was partly supported by the Stanford Platform Lab and its affiliates, and by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. Athinagoras Skiadopoulos was supported by a Stanford Graduate Fellowship. Mark Zhao was supported by a Stanford Graduate Fellowship and a Meta PhD Fellowship.

References

[1] 2024. PyTorch. https://pytorch.org/.

[2] Josh others Achiam. 2023. GPT-4 Technical Report. *arXiv e-prints*, Article arXiv:2303.08774 (March 2023), arXiv:2303.08774 pages. https://doi.org/10.48550/arXiv.2303.08774 arXiv:2303.08774 [cs.CL]

[3] Meta AI. 2024. Meta Llama 3. https://ai.meta.com/blog/meta-llama-3/.

[4] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. 2022. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 472–487. https://doi.org/10.1145/3492321.3519584

[5] David F. Bacon. 2022. Detection and Prevention of Silent Data Corruption in an Exabyte-scale Database System. In *The 18th IEEE Workshop on Silicon Errors in Logic – System Effects*.

[6] Yixin Bao, Yanghua Peng, Chuan Wu, and Zongpeng Li. 2018. Online Job Scheduling in Distributed Machine Learning Clusters. arXiv:1801.00936 [cs.DC]

[7] Romain Beaumont. 2022. Large Scale OpenCLIP: L/14, H/14 AND G/14 Trained on LAION-2B. https://laion.ai/blog/large-openclip/.

[8] Stas Bekman. 2022. The Technology Behind BLOOM Training. https://huggingface.co/blog/bloom-megatron-deepspeed.

[9] Rishi Bommasani et al. 2021. On the Opportunities and Risks of Foundation Models. *arXiv e-prints*, Article arXiv:2108.07258 (Aug. 2021), arXiv:2108.07258 pages. https://doi.org/10.48550/arXiv.2108.07258 arXiv:2108.07258 [cs.LG]

[10] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *arXiv e-prints*, Article arXiv:2005.14165 (May 2020), arXiv:2005.14165 pages. https://doi.org/10.48550/arXiv.2005.14165 arXiv:2005.14165 [cs.CL]

[11] Zhenkun Cai, Xiao Yan, Kaihao Ma, Yidi Wu, Yuzhen Huang, James Cheng, Teng Su, and Fan Yu. 2022. TensorOpt: Exploring the Tradeoffs in Distributed DNN Training With Auto-Parallelism. *IEEE Transactions on Parallel and Distributed Systems* 33, 8 (Aug. 2022), 1967–1981. https://doi.org/10.1109/tpds.2021.3132413

[12] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. 2016. Revisiting Distributed Synchronous SGD. *arXiv e-prints*, Article arXiv:1604.00981 (April 2016), arXiv:1604.00981 pages. https://doi.org/10.48550/arXiv.1604.00981 arXiv:1604.00981 [cs.LG]

[13] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 571–582. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chilimbi

[14] Databricks. 2024. Introducing DBRX: A New State-of-the-Art Open LLM. https://www.databricks.com/blog/introducing-dbrx-new-state-art-open-llm/.

[15] Harish Dattatraya Dixit, Laura Boyle, Gautham Vunnam, Sneha Pendharkar, Matt Beadon, and Sriram Sankar. 2022. Detecting silent data corruptions in the wild. *arXiv e-prints*, Article arXiv:2203.08989 (March 2022), arXiv:2203.08989 pages. https://doi.org/10.48550/arXiv.2203.08989 arXiv:2203.08989 [cs.AR]

[16] 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. 2012. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf

[17] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*. USENIX Association, San Francisco, CA. https://www.usenix.org/conference/osdi-04/mapreduce-simplified-data-processing-large-clusters

[18] DGX 2024. Nvidia DGX Systems. https://www.nvidia.com/en-gb/data-center/dgx-systems/.

[19] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. 2022. Check-N-Run: a Checkpointing System for Training Deep Learning Recommendation Models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 929–943. https://www.usenix.org/conference/nsdi22/presentation/eisenman

[20] William Fedus, Barret Zoph, and Noam Shazeer. 2021. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *arXiv e-prints*, Article arXiv:2101.03961 (Jan. 2021), arXiv:2101.03961 pages. https://doi.org/10.48550/arXiv.2101.03961 arXiv:2101.03961 [cs.LG]

[21] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv e-prints*, Article arXiv:1706.02677 (June 2017), arXiv:1706.02677 pages. https://doi.org/10.48550/arXiv.1706.02677 arXiv:1706.02677 [cs.CV]

[22] Saurabh Gupta, Tirthak Patel, Christian Engelmann, and Devesh Tiwari. 2017. Failures in large scale systems: long-term measurement, analysis, and implications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 44, 12 pages. https://doi.org/10.1145/3126908.3126937

[23] Gurobi. 2024. Gurobi. https://www.gurobi.com.

[24] Tao He, Xue Li, Zhibin Wang, Kun Qian, Jingbo Xu, Wenyuan Yu, and Jingren Zhou. 2023. Unicron: Economizing Self-Healing LLM Training at Scale. *arXiv e-prints*, Article arXiv:2401.00134 (Dec. 2023), arXiv:2401.00134 pages. https://doi.org/10.48550/arXiv.2401.00134 arXiv:2401.00134 [cs.DC]

[25] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. 2012. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine* 29, 6 (2012), 82–97. https://doi.org/10.1109/MSP.2012.2205597

[26] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich

Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. 2022. Training Compute-Optimal Large Language Models. *arXiv e-prints*, Article arXiv:2203.15556 (March 2022), arXiv:2203.15556 pages. https://doi.org/10.48550/arXiv.2203.15556 arXiv:2203.15556 [cs.CL]

[27] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. *GPipe: efficient training of giant neural networks using pipeline parallelism*. Curran Associates Inc., Red Hook, NY, USA.

[28] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. 2021. Elastic Resource Sharing for Distributed Deep Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 721–739. https://www.usenix.org/conference/nsdi21/presentation/hwang

[29] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. 2023. Oobleck: Resilient Distributed Training of Large Models Using Pipeline Templates. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 382–395. https://doi.org/10.1145/3600006.3613152

[30] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 947–960. https://www.usenix.org/conference/atc19/presentation/jeon

[31] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2018. Beyond Data and Model Parallelism for Deep Neural Networks. arXiv:1807.05358 [cs.DC]

[32] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 463–479. https://www.usenix.org/conference/osdi20/presentation/jiang

[33] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. 2024. MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 745–760. https://www.usenix.org/conference/nsdi24/presentation/jiang-ziheng

[34] Norman P. Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Cliff Young, Xiang Zhou, Zongwei Zhou, and David Patterson. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. *arXiv e-prints*, Article arXiv:2304.01433 (April 2023), arXiv:2304.01433 pages. https://doi.org/10.48550/arXiv.2304.01433 arXiv:2304.01433 [cs.AR]

[35] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. *arXiv e-prints*, Article arXiv:2001.08361 (Jan. 2020), arXiv:2001.08361 pages. https://doi.org/10.48550/arXiv.2001.08361 arXiv:2001.08361 [cs.LG]

[36] Taebum Kim, Hyoungjoo Kim, Gyeong-In Yu, and Byung-Gon Chun. 2023. BPIPE: memory-balanced pipeline parallelism for training large language models. In *Proceedings of the 40th International Conference on Machine Learning (ICML 23)*. JMLR.org, Article 682, 15 pages.

[37] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2023. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems* 5 (2023).

[38] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv e-prints*, Article arXiv:1404.5997 (April 2014), arXiv:1404.5997 pages. https://doi.org/10.48550/arXiv.1404.5997 arXiv:1404.5997 [cs.NE]

[39] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM* 60, 6 (May 2017), 84–90. https://doi.org/10.1145/3065386

[40] Teven Le Scao et al. 2022. BLOOM: A 176B-Parameter Open-Access Multilingual Language Model. *arXiv e-prints*, Article arXiv:2211.05100 (Nov. 2022), arXiv:2211.05100 pages. https://doi.org/10.48550/arXiv.2211.05100 arXiv:2211.05100 [cs.CL]

[41] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. *arXiv e-prints*, Article arXiv:2006.16668 (June 2020), arXiv:2006.16668 pages. https://doi.org/10.48550/arXiv.2006.16668 arXiv:2006.16668 [cs.CL]

[42] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 583–598. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu

[43] Shigang Li and Torsten Hoefler. 2021. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 27, 14 pages. https://doi.org/10.1145/3458817.3476145

[44] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch distributed: experiences on accelerating data parallel training. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3005–3018. https://doi.org/10.14778/3415478.3415530

[45] Haibin Lin, Hang Zhang, Yifei Ma, Tong He, Zhi Zhang, Sheng Zha, and Mu Li. 2019. Dynamic Mini-batch SGD for Elastic Distributed Training: Learning in the Limbo of Resources. *arXiv e-prints*, Article arXiv:1904.12043 (April 2019), arXiv:1904.12043 pages. https://doi.org/10.48550/arXiv.1904.12043 arXiv:1904.12043 [cs.LG]

[46] Ziming Liu, Shenggan Cheng, Haotian Zhou, and Yang You. 2023. Hanayo: Harnessing Wave-like Pipeline Parallelism for Enhanced Large Model Training Efficiency. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 56, 13 pages. https://doi.org/10.1145/3581784.3607073

[47] Ilya Loshchilov and Frank Hutter. 2017. Decoupled Weight Decay Regularization. *arXiv e-prints*, Article arXiv:1711.05101 (Nov. 2017), arXiv:1711.05101 pages. https://doi.org/10.48550/arXiv.1711.05101 arXiv:1711.05101 [cs.LG]

[48] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and Efficient GPU Cluster Scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 289–304. https://www.usenix.org/conference/nsdi20/presentation/mahajan

[49] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer Sentinel Mixture Models. arXiv:1609.07843 [cs.CL]

[50] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 203–216. https://www.usenix.org/conference/fast21/presentation/mohan

[51] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3341301.3359646

[52] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient large-scale language model training on GPU clusters using megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 58, 15 pages. https://doi.org/10.1145/3458817.3476209

[53] Maxim Naumov, John Kim, Dheevatsa Mudigere, Srinivas Sridharan, Xiaodong Wang, Whitney Zhao, Serhat Yilmaz, Changkyu Kim, Hector Yuen, Mustafa Ozdal, Krishnakumar Nair, Isabel Gao, Bor-Yiing Su, Jiyan Yang, and Mikhail Smelyanskiy. 2020. Deep Learning Training in Facebook Data Centers: Design of Scale-up and Scale-out Systems. *arXiv e-prints*, Article arXiv:2003.09518 (March 2020), arXiv:2003.09518 pages. https://doi.org/10.48550/arXiv.2003.09518 arXiv:2003.09518 [cs.DC]

[54] OpenAI. 2022. ChatGPT: Language models for task-oriented dialogue. https://openai.com/blog/chatgpt/.

[55] Andrew Or, Haoyu Zhang, and Michael Freedman. 2020. Resource Elasticity in Distributed Deep Learning. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 400–411. https://proceedings.mlsys.org/paper_files/paper/2020/file/c443e9d9fc984cda1c5cc447fe2c724d-Paper.pdf

[56] David A. Patterson, Garth Gibson, and Randy H. Katz. 1988. A case for redundant arrays of inexpensive disks (RAID). *SIGMOD Rec.* 17, 3 (jun 1988), 109–116. https://doi.org/10.1145/971701.50214

[57] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 3, 14 pages. https://doi.org/10.1145/3190508.3190517

[58] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '20)*. IEEE Press, Article 20, 16 pages.

[59] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-infinity: breaking the GPU memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 59, 14 pages. https://doi.org/10.1145/3458817.3476205

[60] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '20)*. Association for Computing Machinery, New York, NY, USA, 3505–3506. https://doi.org/10.1145/3394486.3406703

[61] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 551–564. https://www.usenix.org/conference/atc21/presentation/ren-jie

[62] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv e-prints*, Article arXiv:1802.05799 (Feb. 2018), arXiv:1802.05799 pages. https://doi.org/10.48550/arXiv.1802.05799 arXiv:1802.05799 [cs.LG]

[63] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. 2018. Mesh-TensorFlow: Deep Learning for Supercomputers. *arXiv e-prints*, Article arXiv:1811.02084 (Nov. 2018), arXiv:1811.02084 pages. https://doi.org/10.48550/arXiv.1811.02084 arXiv:1811.02084 [cs.LG]

[64] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *CoRR* abs/1909.08053 (2019). arXiv:1909.08053 http://arxiv.org/abs/1909.08053

[65] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, Elton Zhang, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. 2022. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model. *arXiv e-prints*, Article arXiv:2201.11990 (Jan. 2022), arXiv:2201.11990 pages. https://doi.org/10.48550/arXiv.2201.11990 arXiv:2201.11990 [cs.CL]

[66] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. 2021. Piper: Multidimensional Planner for DNN Parallelization. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., 24829–24840. https://proceedings.neurips.cc/paper_files/paper/2021/file/d01eeca8b24321cd2fe89dd85b9beb51-Paper.pdf

[67] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. 2023. Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 497–513. https://www.usenix.org/conference/nsdi23/presentation/thorpe

[68] Hugo Touvron et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *arXiv e-prints*, Article arXiv:2307.09288 (July 2023), arXiv:2307.09288 pages. https://doi.org/10.48550/arXiv.2307.09288 arXiv:2307.09288 [cs.CL]

[69] Pablo Villalobos, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, Anson Ho, and Marius Hobbhahn. 2022. Machine Learning Model Sizes and the Parameter Gap. *arXiv e-prints*, Article arXiv:2207.02852 (July 2022), arXiv:2207.02852 pages. https://doi.org/10.48550/arXiv.2207.02852 arXiv:2207.02852 [cs.LG]

[70] Minjie Wang, Chien-chin Huang, and Jinyang Li. 2019. Supporting Very Large Models using Automatic Dataflow Graph Partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 26, 17 pages. https://doi.org/10.1145/3302424.3303953

[71] Stephanie Wang, John Liagouris, Robert Nishihara, Philipp Moritz, Ujval Misra, Alexey Tumanov, and Ion Stoica. 2019. Lineage stash: fault tolerance off the critical path. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 338–352. https://doi.org/10.1145/3341301.3359653

[72] Tianyang Wang, Jun Huan, and Bo Li. 2018. Data Dropout: Optimizing Training Data for Convolutional Neural Networks. *arXiv e-prints*, Article arXiv:1809.00193 (Sept. 2018), arXiv:1809.00193 pages. https://doi.org/10.48550/arXiv.1809.00193 arXiv:1809.00193 [cs.CV]

[73] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. 2023. GEMINI: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 364–381. https://doi.org/10.1145/3600006.3613145

[74] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS

in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 945–960. https://www.usenix.org/conference/nsdi22/presentation/weng

[75] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. 2017. ImageNet Training in Minutes. *arXiv e-prints*, Article arXiv:1709.05011 (Sept. 2017), arXiv:1709.05011 pages. https://doi.org/10.48550/arXiv.1709.05011 arXiv:1709.05011 [cs.CV]

[76] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: Open Pre-trained Transformer Language Models. *arXiv e-prints*, Article arXiv:2205.01068 (May 2022), arXiv:2205.01068 pages. https://doi.org/10.48550/arXiv.2205.01068 arXiv:2205.01068 [cs.CL]

[77] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *arXiv e-prints*, Article arXiv:2304.11277 (April 2023), arXiv:2304.11277 pages. https://doi.org/10.48550/arXiv.2304.11277 arXiv:2304.11277 [cs.DC]

[78] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 559–578. https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin

[79] Yazhou Zu, Alireza Ghaffarkhah, Hoang-Vu Dang, Brian Towles, Steven Hand, Safeen Huda, Adekunle Bello, Alexander Kolbasov, Arash Rezaei, Dayou Du, Steve Lacy, Hang Wang, Aaron Wisner, Chris Lewis, and Henri Bahini. 2024. Resiliency at Scale: Managing Google's TPUv4 Machine Learning Supercomputer. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 761–774. https://www.usenix.org/conference/nsdi24/presentation/zu