

TENPLEX: Dynamic Parallelism for Deep Learning using Parallelizable Tensor Collections

Marcel Wagenländer
Imperial College London

Guo Li
Imperial College London

Bo Zhao
Aalto University

Luo Mai
University of Edinburgh

Peter Pietzuch
Imperial College London

Abstract

Deep learning (DL) jobs use multi-dimensional parallelism, i.e. combining data, model, and pipeline parallelism, to use large GPU clusters efficiently. Long-running jobs may experience changes to their GPU allocation: (i) resource elasticity during training adds or removes GPUs; (ii) hardware maintenance may require redeployment on different GPUs; and (iii) GPU failures force jobs to run with fewer devices. Current DL frameworks tie jobs to a set of GPUs and thus lack support for these scenarios. In particular, they cannot change the multi-dimensional parallelism of an already-running job in an efficient and model-independent way.

We describe TENPLEX, a state management library for DL systems that enables jobs to change their parallelism dynamically after the GPU allocation is updated at runtime. TENPLEX achieves this through a new abstraction, a *parallelizable tensor collection* (PTC), that externalizes the job state during training. After a GPU change, TENPLEX uses the PTC to transform the job state: the PTC repartitions the dataset state under data parallelism and exposes it to GPU workers through a virtual file system; and the PTC obtains the model state as partitioned checkpoints and transforms them to reflect the new parallelization configuration. For efficiency, TENPLEX executes PTC transformations in parallel with minimum data movement between GPU workers. Our experiments show that TENPLEX enables DL jobs to support dynamic parallelization with low overhead.

1 Introduction

Deep learning (DL) has led to remarkable progress in many domains, including conversational AI [43], natural language processing [7, 10, 22], computer vision [23, 63], and recommender systems [19, 75]. These advances, however, are due to the ever-increasing sizes of deep neural network (DNN) models and training datasets: e.g. OpenAI’s GPT-3 language model has 175 billion parameters, which require over 700 GB of memory [7]. Large DNN models, therefore, are trained in a distributed fashion with parallel hardware accelerators, such as GPUs [52], NPUs [12], or TPUs [30].

Many organizations have invested in DL clusters with thousands of GPUs [28], and DL jobs are deployed on GPUs using *multi-dimensional* parallelism [2, 54, 72]. It combines data [31], pipeline [26, 40], and model/tensor parallelism [8], and these strategies are implemented either by *model-specific libraries*, (Megatron-LM [59], DeepSpeed [55]), or *deployment-time parallelizers* (Alpa [74], Unity [64]).

Due to their high cost [24], organizations must manage GPU clusters efficiently. Users submit training jobs with multi-dimensional parallelism to a *DL job scheduler* [32, 60], which allocates it to GPUs. An emerging requirement is that, due to the long-running nature of DL jobs, the original GPU allocation of a job may change over time [60] for several reasons: (i) **elasticity**—to maintain high cluster utilization, DL jobs want to claim extra GPU resources when they become available [46]; (ii) **redemption**—DL jobs may have to release specific GPUs and migrate to others to reduce fragmentation [68], support hardware maintenance, or handle preemption by higher priority jobs [47]; and (iii) **failure recovery**—DL jobs may lose GPUs at runtime due to failures and must continue training with fewer GPUs after recovering from checkpoints [66].

We observe that current DL systems (PyTorch [46], TensorFlow [1], MindSpore [38]) do not allow DL job schedulers to change GPU resources at runtime. They lack a property that we term *device-independence*: DL jobs are tightly coupled to GPUs at deployment time, preventing schedulers from changing the allocation. As we show in §2.3, changing the GPU allocation of a job with multi-dimensional parallelism also means that its current parallelization strategy may no longer be optimal, thus requiring the replanning of its parallelization approach.

Both industry [60] and academia [36] have recognized the need for changing DL job resources dynamically, resulting in three types of solutions: (a) **model parallelizers**, e.g. Megatron-LM [59] and DeepSpeed [55], can be extended with support to change GPU allocation during training. Such approaches, however, are limited to supported DNN architec-

tures, such as specific transformer models [59]; (b) **elastic DL systems**, e.g. Torch Elastic [49], Elastic Horovod [25], and KungFu [36] can adapt the number of model replicas on GPUs at runtime. By only adapting model replicas, such solutions are limited to data-parallel DL jobs only and do not support generic multi-dimensional parallelism; and (c) **virtual devices**, used by e.g. VirtualFlow [44], EasyScale [34], and Singularity [60], decouple DL jobs from physical devices: jobs assume a maximum number of virtual GPUs, which are then mapped to fewer physical GPUs at runtime. While this is transparent to job execution, it requires complex virtualization at the GPU driver level [60] and also does not support changes with multi-dimensional parallelization [34].

In this paper, we explore a different point in the design space for supporting dynamic resource changes in DL clusters. Our idea is to create a **state management library** for DL systems that (i) *externalizes* the training state from a DL job (i.e. the model and dataset partitions); and then (ii) *transforms* the state in response to dynamic GPU changes.

To design such a library, we answer several questions: (1) what is a suitable abstraction for representing the DL job state, so that it can be transformed when adapting multi-dimensional parallelism after a GPU change? (2) how can a state management library retrieve the job state from the DL system with little change to its implementation? (3) how can the library deploy changes to the multi-dimensional parallelism of large DL jobs with low overhead?

We describe **TENPLEX**, a state management library for DL systems that enables jobs with multi-dimensional parallelism to support dynamic changes to GPU resources during training. TENPLEX makes the following new technical contributions:

(1) Externalizing DL job state. TENPLEX extracts the DL job state from the DL system and represents it using a tensor-based abstraction, which we call a *parallelizable tensor collection* (PTC). A PTC is a hierarchical partitioned collection of tensors that contains the (i) *dataset state* of the job, expressed as a set of training data partitions, and (ii) the *model state*, expressed as partitioned checkpoints of the DNN model parameters. The PTC partitioning depends on the multi-dimensional parallelization of the job, i.e. how the job uses data, pipeline, and model parallelism.

TENPLEX must expose the DL job state to a PTC and support efficient access by the DL system. TENPLEX stores a PTC in a hierarchical virtual file system (implemented using Linux’ FUSE interface [65]), which is maintained in memory for efficient access: (1) for the dataset state, TENPLEX loads the training data into the workers’ host memory. To support data parallelism, each data partition has a virtual directory. It contains the files with the training data samples that the worker must process; (2) for the model state, TENPLEX retrieves the partitioned model checkpoints created by the DL system, and the PTC stores them as a hierarchy of virtual files. The hierarchy mirrors the layered structure of the partitioned model tensors, simplifying state transformations when the

multi-dimensional parallelization changes.

(2) Transforming DL job state. When the DL job scheduler alters the GPU allocation, TENPLEX transforms the state maintained as a PTC to change the multi-dimensional parallelization configuration. After a GPU change, TENPLEX requests a new parallelization configuration from a parallelizer (e.g. Megatron-LM [59] or Alpa [74]). It then applies *state transformations* to the PTC that updates the partitioning of the tensors that represent the dataset and model states.

The state transformations ensure that the PTC remains consistent, i.e. the convergence of the DL job is unaffected. For the dataset state, TENPLEX repartitions the training data and makes the new data partitions available to workers while keeping the data access order of samples unaffected across iterations; for the model state, TENPLEX repartitions the model layers and associated tensors and creates new partitioned model checkpoints. The partitioned checkpoints are then loaded by the new set of GPU devices.

(3) Optimizing DL job state changes. The reconfiguration of the DL job state must be done efficiently, e.g. reducing data transfer to disseminate the new state to workers. TENPLEX therefore parallelizes the PTC transformations across all workers: it then sends the minimum amount of data to establish the correctly partitioned state on all workers. TENPLEX allows workers to fetch sub-tensors from the PTC through an HTTP API, avoiding unnecessary data movement. TENPLEX also overlaps the sending of samples of new dataset partitions with model training, which permits the DL system to resume training before the full dataset partitions are received.

We implement TENPLEX as a Go library with 6,700 lines of code. It integrates with existing DL libraries, such as PyTorch [46], and systems, such as Megatron-LM [59] and DeepSpeed [55]. Our evaluation shows that TENPLEX can dynamically change a DL job’s GPU resources with any parallelization configuration with good performance: it reduces training time by 24% compared to approaches that only scale along the data parallelism dimension; resource reconfiguration takes 43% less time than approaches that migrate all GPU state and 75% less compared to maintaining state centrally.

2 Resources Changes in DL Training

Next, we describe DL jobs with multi-dimensional parallelism (§2.1). We then motivate resource changes during training (§2.2) and discuss associated challenges (§2.3). We finish with a survey of current approaches for adapting resources during training and their limitations (§2.4).

2.1 Deep learning jobs with parallelism

Training DNN models, e.g. large language models (LLMs) [51], is resource-intensive and must scale to clusters with many accelerators, such as GPUs [52], NPUs [12], or TPUs [30]. A single DL job may be executed on 1,000s of GPUs [62] by distributing it across workers,

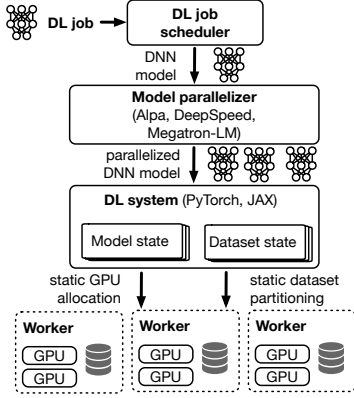


Fig. 1: Training deep learning (DL) jobs with multi-dimensional parallelism on a shared GPU cluster

each with multiple GPUs.

Fig. 1 shows a typical deployment for DL jobs in a shared GPU cluster. A *DL job scheduler* (e.g. Pollux [50], Gandiva [70]) manages the GPUs and assigns jobs to them. When running a job, a *model parallelizer* (e.g. Alpha [74], DeepSpeed [55], Megatron-LM [59]) decides on a parallelization configuration for the job by considering multiple dimensions: (1) *data parallelism* [31] partitions training data across workers and replicates the DNN model on those workers. Workers compute model updates using their local data partitions and synchronize these updates after each training iteration; (2) *model parallelism* [8] splits the model, i.e. the operators and parameters in the computational graph [1, 38, 46], and assigns partitions to workers; and (3) *pipeline parallelism* [26, 40] partitions the model into stages [26]. Training data batches are then split into smaller micro-batches and pipelined across workers.

Recent advances [18, 29, 41, 59] have shown that a combination of parallelism along these dimensions, i.e. *multi-dimensional parallelism*, improves the performance of large DNN model training. Different parallelization configurations have different properties in terms of their scalability, device utilization, and memory consumption: data parallelism alone cannot scale to large deployments due to its synchronization overheads and reliance on large batch sizes [58]; pipeline parallelism under-utilize devices due to pipeline bubbles [40]; and model parallelism incurs high communication overheads but must be used to fit large models that surpass GPU memory [5]. By combining multiple strategies, model parallelizers [64, 74] navigate this trade-off space.

After generating a multi-dimensional parallelization plan, the DNN model training is performed on the GPU cluster by a DL system (e.g. PyTorch [46], TensorFlow [1], MindSpore [38]). The deployed DL job consists of the *dataset state* and *model state*: the dataset state contains the partitions with the training data samples, and records the read positions across these partitions; the model state consists of the partitioned model and optimizer parameters. The partitioned state is assigned to the workers (see Fig. 1).

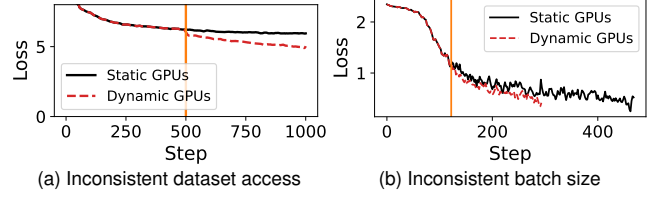


Fig. 2: Impact of GPU change on training convergence (Changing GPUs from 2 to 4 with GPT-3 and MNIST)

2.2 Need for dynamic resource changes

Since it may take hours, days, or weeks to run a single DL job, e.g. training a large language model (LLM) [11, 42, 67], the DL job scheduler may change the GPU allocation of jobs at runtime. There are several reasons for this:

(1) Elasticity. DL job schedulers may *elastically* increase and decrease the allocated GPUs for a job based on the available resources [32, 71]. When a job completes, an elastic scheduler can re-allocate the freed-up GPUs to other jobs, e.g. giving each job a fair share of GPUs [15]. Higher priority jobs submitted to the cluster may need to take GPU resources away from already-running jobs [45].

In cloud environments [17, 37], elastic schedulers can take advantage of differences in GPU pricing. When lower cost “spot” GPUs become available [48], the scheduler may add them to existing jobs; when spot GPUs are preempted, jobs must continue execution with fewer GPUs. Elastic schedulers thus improve shared cluster utilization, reduce cost, and decrease completion times [69].

(2) Redeployment. DL job schedulers may reallocate jobs to a new set of GPUs for operational reasons. For example, before performing hardware maintenance or upgrades, a job may have to be shifted to a new set of GPUs at runtime.

The redeployment of jobs can also reduce fragmentation in the allocated GPUs. If a job uses GPUs spread across disjoint workers, communication must use lower-bandwidth networks (e.g. Ethernet or InfiniBand) as opposed to higher-bandwidth inter-connects between GPUs (e.g. NVLink). A scheduler may therefore change the GPU allocation of a job to de-fragment it onto fewer workers [68].

(3) Failure recovery. Long-running jobs may lose GPU resources due to failures, caused by hardware faults, network outages, or software errors [11, 42]. After a fault, the job must continue execution after recovering from the last state checkpoint [39]. In some cases, the failed worker or GPUs can be replaced by new resources before resuming the job; in other cases, the job can resume with fewer GPUs, which affects its optimal parallelization configuration.

2.3 Challenges when changing GPU resources

Changing GPUs for a job at runtime adds challenges:

(1) Impact on convergence. When changing the number of GPUs, the convergence of a job may be affected, and the final trained model may have e.g. a different accuracy. In today’s

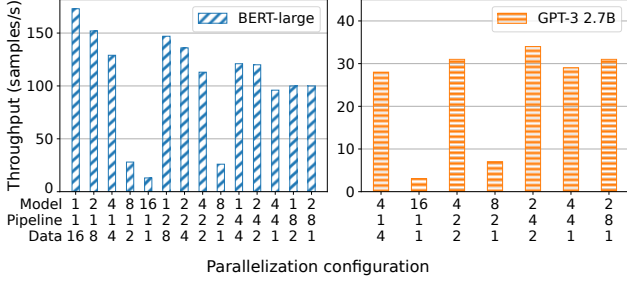


Fig. 3: Performance impact of different parallelization configurations on 16 GPUs

DL systems, job convergence depends on the specific set of GPUs used, as current jobs are not *device-independent*. There are multiple reasons for this:

Consistency of training dataset. A DL job must maintain dataset consistency during training, i.e. it must process training data samples exactly once and in a consistent order in each training epoch. Dataset consistency must also hold when GPU changes under data parallelism, which affects the data sharding and requires re-partitioning. For example, when re-partitioning the dataset within an epoch, the order in which data samples are ingested from that point onwards must not change for convergence to be unaffected.

Fig. 2a shows how model convergence, plotted as the loss value, is affected after adding a GPU (vertical orange line) under data parallelism. The solid black line shows regular model convergence with a static GPU allocation; the dashed red line shows convergence after the scale-out event when the dataset is processed inconsistently after re-partitioning: when resuming the training in the middle of the epoch, the first half of the training data is used twice, which overfits the model and reduces the loss value unreasonably.

Consistency of hyper-parameters. Hyper-parameter choices, such as batch sizes, and learning rate [61], depend on the GPU resources of a job. For example, the local batch size is fixed for each GPU and is typically chosen to keep devices fully utilized with data; the global batch size therefore changes with the number of GPUs.

In Fig. 2b, we show how the global batch size must be kept constant after adding a GPU (vertical orange line) under data parallelism. The solid black line shows model convergence (measured as loss) without the GPU change. The dashed red line shows the divergence when the GPU allocation changes but the device batch size remains constant.

(2) Impact on performance. The best parallelization configuration for a DL job, i.e. one achieving the lowest time-to-accuracy, depends on the GPU resources used by the job.

Parallelization configuration. The best multi-dimensional parallelization, in terms of data, model, and pipeline parallelism, depends on many factors, including the number and type of GPUs, the bandwidth and latency of the GPU inter-connect and the network between workers, and the size and struc-

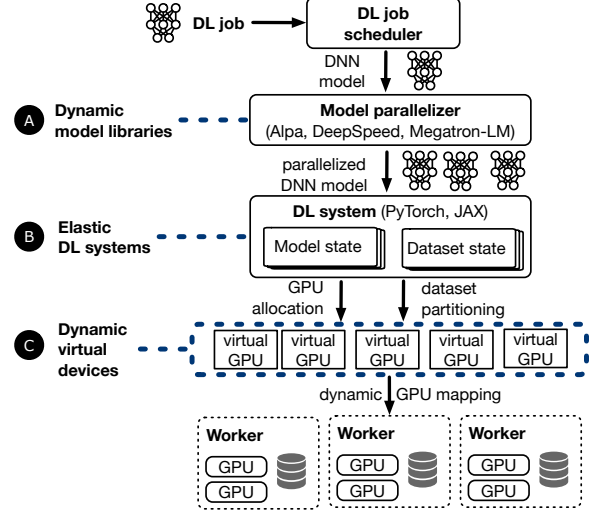


Fig. 4: Approaches for dynamic resource changes in DL jobs

ture of the DNN model architecture. Model parallelizers, e.g. Alpa [74] and Unity [64], consider these factors based on profiled performance data and/or analytical cost models when choosing a parallelization configuration.

When the GPU resources of a DL job change at runtime, a parallelization configuration that was optimal at deployment time may no longer be optimal with the new GPUs. We demonstrate this empirically in Fig. 3, which shows the training throughput (in samples/second) when training BERT [10] and GPT-3 [7] models using Megatron-LM [59] on 16 GPUs under a range of parallelization configurations. Each parallelization configuration varies the degree of model, pipeline and data parallelism, and thus alters the GPU allocation.

As the results show, the training throughput differs by over $10\times$ between the best and the worst configuration, despite the fact that each configuration uses the same number of GPUs (16). The configuration $(M, P, D) = (2, 4, 2)$ performs well, because it uses communication-intensive model parallelism only within workers; the configuration $(16, 1, 1)$ performs the worst, because model parallelism must use slower inter-worker links.

Reconfiguration cost. After changing parallelization, a job’s partitioned dataset and model state are no longer correct. The state must be re-partitioned and the new partitions must be sent to workers, which may involve large data movement. For example, prior work [60] reports that reducing the GPU allocation for a DL job from 16 to 8 GPUs may take 122 secs.

2.4 Current approaches

A number of approaches have been proposed to allow DL job schedulers to change GPU resources dynamically. We give an overview, and then discuss specific proposals, assessing them against the challenges from §2.3.

Fig. 4 shows the main approaches: **A** *dynamic model libraries* adapt to changes in GPUs by producing a new parallelization configuration at runtime; **B** *elastic DL systems*

Approach	Systems	Consistency		Parallelism						Reconfiguration overhead
		Dataset	Hyper-params	Static DP	Static PP	Static MP	Dynamic DP	Dynamic PP	Dynamic MP	
A Model libraries	Alpa [74]	-	-	✓	✓	✓	-	-	-	-
	Megatron-LM [59]	-	-	✓	✓	✓	✓	×	×	full state
	DeepSpeed [55]	✓	✓	✓	✓	×	✓	×	×	full state
B Elastic DL systems	Elastic Horovod [25]	×	×	✓	-	-	✓	-	-	full state
	Torch Distributed [49]	✓	×	✓	✓	(✓)	✓	(✓)	(✓)	full state
	Varuna [4]	✓	✓	✓	✓	-	✓	✓	-	full state
	KungFu [36]	✓	✓	✓	-	-	✓	-	-	full state
C Virtual devices	VirtualFlow [44]	✓	✓	✓	-	-	✓	-	-	full state
	EasyScale [34]	✓	✓	✓	-	-	✓	-	-	full state
	Singularity [60]	✓	✓	✓	✓	✓	✓	×	×	GPU state
State management	TENPLEX	✓	✓	✓	✓	✓	✓	✓	✓	minimal state

Tab. 1: Comparison of existing proposals for supporting dynamic GPU changes in DL jobs

include support to scale GPU resources out and in at runtime; and **C** *virtual devices* decouple physical from logical GPUs, allowing the mapping to change at runtime.

Tab. 1 compares systems that implement these approaches:

A Model libraries. The Alpa model parallelizer [74] provides a parallelization configuration at job deployment time and does not support dynamic changes. Megatron-LM [59] and DeepSpeed [55] support dynamic resource changes under data parallelism only by dividing batches into micro-batches [53, 62]. By changing the allocation of micro-batches, DeepSpeed ensures consistency after resource changes. Since the full training state is moved to and from remote storage, there is a high reconfiguration overhead.

In summary, model libraries do not handle dynamic multi-dimensional parallelism, and they lack integration with DL systems, requiring manual state re-partitioning.

B Elastic DL systems. Elastic Horovod [57] exposes the model state through a user-defined state object. It allows users to synchronize state across workers when changing data parallelism, but state re-distribution must be implemented manually. In particular, the dataset state can become inconsistent if scaling does not occur at epoch boundaries. Torch Distributed Elastic/Checkpoint [35] provides a model broadcast API to save/resume model checkpoints and allows users to implement re-partitioning operations. Users must ensure the consistency of hyper-parameters and perform the required data movement between workers though. Varuna [4] lets users define cut-points at which the model pipeline is partitioned at runtime when resources change. KungFu [36] uses a broadcast operation to distribute the training state and with it the model replicas.

Overall, elasticity support either does not account for full multi-dimensional parallelism or requires users to implement state re-partitioning and distribution manually.

C Virtual devices make DL jobs device-independent by virtualizing resources and allowing the mapping between virtual/physical resources to change at runtime. The set of virtual resources exposed to a job represents the maximum resources available at runtime. VirtualFlow [44] uses an all-gather oper-

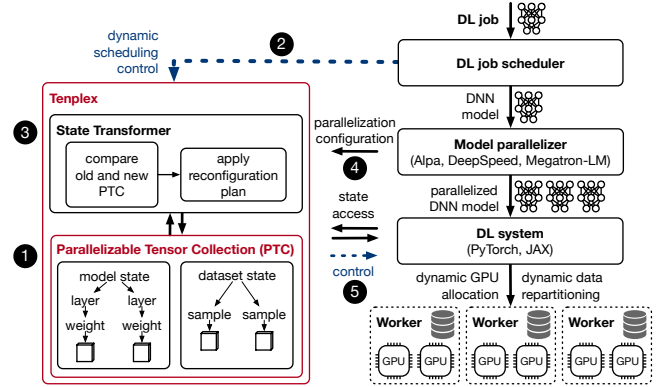


Fig. 5: TENPLEX design

ation to send the current training state to the new workers. To ensure dataset consistency, it follows exactly once semantics for data loading. EasyScale [34] uses a thread abstraction and performs process snapshotting to capture state. Singularity [60] obtains the full GPU state through virtualization at the CUDA driver level.

As a consequence, virtual device approaches are effective at supporting dynamic data parallelism, which does not require re-partitioning, but they cannot support runtime changes with multi-dimensional parallelism.

3 TENPLEX Design

TENPLEX’s goal is to support dynamic GPU changes of DL jobs while (i) ensuring the consistency of the training result, (ii) supporting arbitrary reconfiguration of jobs with multi-dimensional parallelism, and (iii) maintaining a low reconfiguration overhead (see Tab. 1).

TENPLEX’s design is based on the observation that resource changes at runtime affect a DL job’s state, but existing DL systems lack an abstraction to expose the state and transform it at runtime. Fig. 5 shows the idea behind TENPLEX’s design as a *state management library* for DL systems. TENPLEX *externalises* the job state from the DL system (see 1) and manages it as a *parallelizable tensor collection* (PTC). A PTC provides a hierarchical tensor representation of the job’s model and dataset state, and it enables TENPLEX to modify the state

across GPU devices after the parallelization configuration is updated, transparently to the DL system.

The updates to the GPU resources of a job are decided by a DL scheduler at runtime. The scheduler can increase or decrease a job’s GPU allocation and notify TENPLEX (②). TENPLEX then invokes its State Transformer (③), which first obtains a new parallelization configuration for the job from a model parallelizer, such as Alpa [74] or Megatron-LM [59] (④). Based on the new configuration, the State Transformer calculates a *reconfiguration plan*. The plan describes how the state represented by the PTC must change across the GPU devices to implement the new parallelization configuration. The reconfiguration plan is then executed by re-partitioning and re-distributing the data and model state (⑤).

We explain how the PTC abstraction allows TENPLEX to manage the state of a DL job with multi-dimensional parallelism in §4, and how TENPLEX implements the state changes required by the reconfiguration plan efficiently in §5.

4 Parallelizable Tensor Collection

Next, we describe the PTC abstraction (§4.1) and how it is used by TENPLEX to compute reconfiguration plans (§4.2).

4.1 PTC overview

A *parallelizable tensor collection* (PTC) is TENPLEX’s abstraction to represent the parallelized state of a DL job. Such an abstraction must satisfy several requirements: a PTC must match how the DL system represents parallelized state, so that TENPLEX can obtain and adapt the state correctly (R1); it must represent the state of any multi-dimensional parallelization strategies to make TENPLEX compatible with current parallelizers and their parallelization approaches (R2); and it must facilitate the computation of an efficient reconfiguration plan for transforming the state from the current configuration to a new one, e.g. with minimal data movement between GPU workers (R3).

Since a PTC must capture the full execution state of a DL job, it includes (i) the *dataset state*, which consists of the training data and an iterator that records the processed data in the current epoch; and (ii) the *model state*, which constitutes of the DNN model parameters of all layers. A PTC expresses both types of state in a unified manner as a collection of *tensors*, which allows TENPLEX to manipulate the tensors when changing the parallelization configuration.

It is efficient for TENPLEX to manage both types of state using the PTC (R1): the dataset state is maintained directly in the PTC by TENPLEX and exposed to the DL system through a suitable data access API; and the current model state is retrieved prior to reconfiguration from distributed model checkpoints created by the DL system (see §5.2).

The PTC abstraction must be compatible with any multi-dimensional parallelism strategies (R2). This allows TENPLEX to support an arbitrary parallelization configuration that combines data, model, and pipeline parallelism [6, 26, 55], as

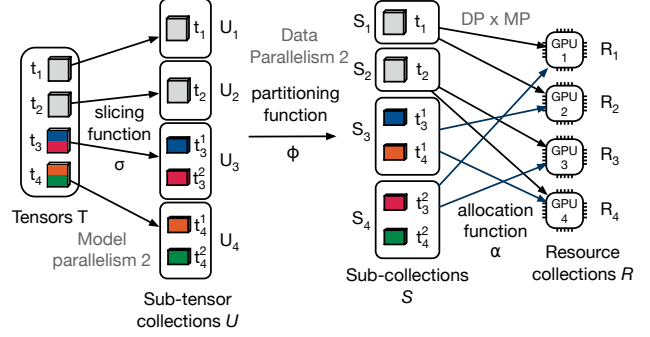


Fig. 6: A PTC example. Edges denote tensor mappings.

provided by model parallelization libraries (e.g. Megatron-LM [59]) or auto parallelizers (e.g. Alpha [74], Unity [64]). TENPLEX must then capture the impact of the parallelization strategy on the PTC state. Here, we observe that any multi-dimensional parallelization strategy can be expressed as a *slicing* of state tensors, followed by a *partitioning* of these tensors across GPU devices.

PTC exploits this observation to define parallelism with three mapping functions: (i) a *slicing* function encodes how tensors are split into sub-tensors, as dictated by model parallelism; (ii) a *partitioning* function then groups these sub-tensors into collections that can be assigned to devices, capturing data and pipeline parallelism; and (iii) an *allocation* function maps these sub-tensor collections to GPU devices for execution. Despite their simplicity, these three functions are sufficient to express any multi-dimensional parallelization strategies in a DL job.

We define the PTC as a tensor collection T , slicing function σ , partitioning function ϕ , and allocation function α :

$$\text{PTC} = (T, \sigma, \phi, \alpha) \quad (1)$$

where $T = D \cup M$ are the tensors that make up the model M and dataset D , $T = \{t_1, \dots, t_n\}$; σ slices a tensor t into sub-tensors, $\sigma(t) = \{t^1, \dots, t^m\}$, $t \in T$. We denote all sets of sub-tensors as U , i.e. $U = \{\sigma(t_1) \dots \sigma(t_n)\}$; ϕ partitions U into sub-collections S , $\phi(U) = \{S_1, \dots, S_p\}$; α allocates sub-collections to GPUs of the resource pool R , $\alpha(S_i) = \{r_1, \dots, r_q\}$.

Fig. 6 shows an example of a PTC that describes the state of a job deployed on 4 GPUs with both model parallelism and data parallelism of degree 2. Here, the data samples $\{t_1, t_2\}$ and model parameters $\{t_3, t_4\}$ are tensors. With two-way model parallelism, each model tensor must be split into 2 sub-tensors: the slicing function σ slices each tensor in T and creates a collection of sub-tensors, $U = \{U_1, \dots, U_4\}$. With two-way data parallelism, each data tensor becomes its own sub-collection, and the model tensors are grouped by sub-tensor offset j of t_i^j : the partitioning function ϕ takes U and maps it to 4 sets. These sets are then assigned to the GPUs, R_1 to R_4 , by the allocation function α , forming a cross-product of the data and model sub-collections.

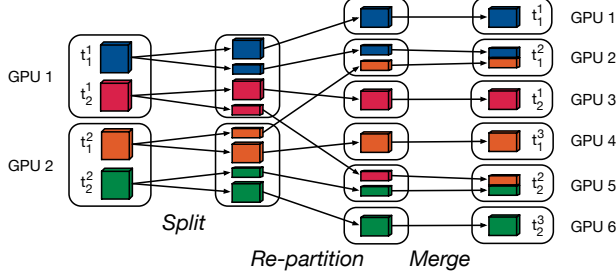


Fig. 7: Example of reconfiguration plan. Edges denote tensor mappings.

4.2 Reconfiguration plan

The PTC abstraction allows TENPLEX to decide how to reconfigure a DL job by computing a “delta” between the two PTCs: if there is a current PTC and a new PTC’, it is possible to compute a minimal sequence of operations that must be executed to turn the state of PTC into that of PTC’. We term such a sequence of operations a *reconfiguration plan*.

We observe that a reconfiguration plan can be expressed only in terms of *split*, *re-partition* and *merge* operations. These operations update the data and model tensors in the PTC: the reconfiguration plan takes the sliced/partitioned tensors that exist on the GPU workers, as described by PTC, and transforms them, so they become the state described by PTC’. This is done efficiently by only exchanging a minimal set of sub-tensors between GPU workers.

To generate such a reconfiguration plan, TENPLEX considers the differences between the current PTC functions, (σ, ϕ, α) , and the new ones, $(\sigma', \phi', \alpha')$. It then generates a sequence of operations: (i) if the sub-tensors U of PTC and U' of PTC’ are different, a *split* operation slices the sub-tensors according to the current slicing function σ and the new σ' ; (ii) a *re-partition* operations move the split tensors from a previous GPU R to R' ; and (iii) if sub-tensors were previously split but are now on the same GPU, a *merge* operation combines them again to reflect σ' . Performing the *re-partition* operation between the *split* and *merge* operations minimizes data movement, because only necessary tensors are moved.

Fig. 7 shows an example of a reconfiguration plan, which contains model parallelism (MP) with 2 GPUs, to PTC’, which combines model (MP) and pipeline parallelism (PP) with 6 GPUs. The tensors must change from a slicing into two sub-tensors, $\sigma(t_i) = \{t_i^1, t_i^2\}$, to a slicing, $\sigma'(t_i) = \{t_i^1, t_i^2, t_i^3\}$, into 3 sub-tensors. Therefore, the *split* operation divides up each sub-tensor into two parts, which the *re-partition* operation moves to new GPUs, as defined by the new partition function ϕ' , forming the sub-tensor t_i^j . In the final step, the *merge* operation takes the split tensors and merges them into required sub-tensors. With PP of degree 2, there is only 1 tensor per stage, and with MP of degree 3, there is only one sub-tensor per GPU.

Alg. 1 formalizes the computation of reconfiguration

Algorithm 1: Reconfiguration plan generation

Data: PTC = $(T, \sigma, \phi, \alpha)$, PTC’ = $(T, \sigma', \phi', \alpha')$
Resources R, R'
Result: Reconfiguration plan \mathcal{P}

```

1  $U \leftarrow \{\sigma(t) \mid t \in T\}$  // get sub-tensor collections
2 foreach  $r \in R$  do // start SPLIT
3    $V \leftarrow \{v \mid v \in U, \alpha(\phi(U)) = r\}$  // get sub-tensors of  $r$ 
4   foreach  $v \in V$  do
5      $\mathcal{P} \leftarrow \mathcal{P} \parallel \text{split}(v, \sigma, \sigma')$ 
6  $S' \leftarrow \phi'(\{\sigma'(t) \mid t \in T\})$  // get sub-collections
7 foreach  $r' \in R'$  do // start RE-PARTITION
8    $S'_r \leftarrow \{S'_i \mid S'_i \in S', \alpha(S'_i) = r'\}$  // get sub-tensors of  $r'$ 
9   foreach  $s' \in S'_r$  do
10     $t \leftarrow \text{get\_base\_tensor}(\sigma', \phi', s')$ 
11     $W \leftarrow \text{get\_split\_tensors}(t, \sigma, \sigma')$ 
12    foreach  $w \in W$  do
13       $r_w \leftarrow \text{get\_resource}(\phi, \alpha, w)$ 
14       $\mathcal{P} \leftarrow \mathcal{P} \parallel \text{move}(w, r_w, r')$  // add MOVE
15     $\mathcal{P} \leftarrow \mathcal{P} \parallel \text{merge}(W)$  // add MERGE

```

plan \mathcal{P} from PTC and PTC’. First, TENPLEX performs the *split*: it starts by generating the current sub-tensor set U based on the slicing function σ . For each resource r , it filters the sub-tensors by the sub-tensors v that are on resource r (line 3). For each sub-tensor v , it then adds a *split* operation to the reconfiguration plan \mathcal{P} (line 5). Where to split is decided based on the current σ and new slicing σ' . Second, TENPLEX considers *re-partition*: it starts with all sub-collections S' and filters them by resource r' (lines 8). For each sub-tensor, it gets the base tensor $t \in T$ for the sub-tensor s' and how it is divided after the *split* operation (lines 10–11). For each split tensor, it obtains the resource r_w for split tensor w . It then appends a *move* sub-operation to the reconfiguration plan \mathcal{P} to move w from r_w to r' (line 14). Finally, TENPLEX adds a *merge* operation to \mathcal{P} , which merges the split tensors in W (line 15).

5 TENPLEX Architecture

In this section, we describe TENPLEX’s architecture and how it implements the PTC abstraction to reconfigure DL jobs efficiently after resource changes.

As shown in Fig. 8, TENPLEX executes on each worker and has two main components: a distributed *State Transformer* and an in-memory *Tensor Store*. The State Transformer inputs the model and dataset partitions from a previous PTC and creates updated partitions to comply with a new PTC’ after a resource change; the Tensor Store maintains the model and dataset state partitions represented by the PTC in a hierarchical virtual in-memory file system. It offers APIs to interface with the DL system’s support for model checkpointing and to allow the DL system to ingest training data.

5.1 State Transformer

When the resources of a DL job change, TENPLEX must create new state partitions based on the updated parallelization plan, so that the DL system can resume executing the job. Since this state transformation can be parallelized, TENPLEX maintains

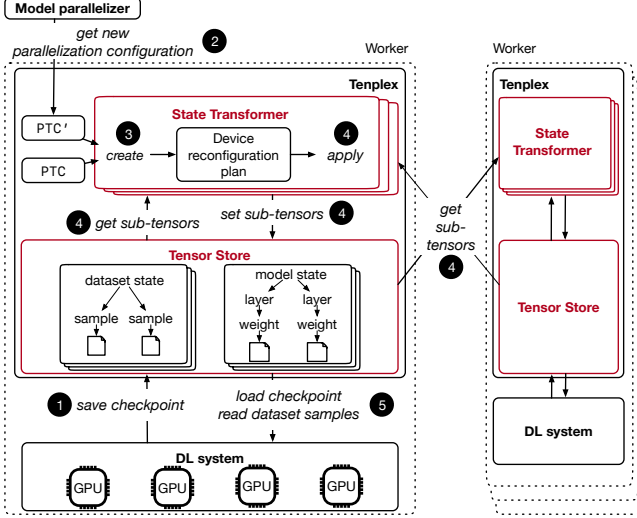


Fig. 8: TENPLEX architecture

an instance of the *State Transformer* for each resource r on a worker. Each *State Transformer* instance then applies its part of the reconfiguration plan (see §4.2).

TENPLEX executes the following steps to modify the job after a resource change from the scheduler (see Fig. 8): ❶ TENPLEX obtains the training state from the DL system by retrieving a model checkpoint partition per GPU. Each checkpoint is written to a model state partition in the Tensor Store; ❷ it then requests a new parallelization configuration from the model parallelizer. The new configuration is expressed as PTC' and becomes the basis for the reconfiguration; ❸ each *State Transformer* instance uses Alg. 1 to create the device's reconfiguration plan. It compares PTC and PTC' and infers the local transformation operations; ❹ the *State Transformer* then applies the *split*, *re-partition*, and *merge* operations to generate new state partitions (see §4.2). It retrieves the necessary sub-tensors from either the local or remote Tensor Stores and saves them in the local Tensor Store; and ❺ it instructs the DL system to restore the job from the checkpoints based on the transformed model partitions in the local Tensor Store. After resuming the job, the DL system continues reading data samples from the local Tensor Store.

The *State Transformer* is designed to interact with different model parallelizers, and thus provides a universal method for describing parallelization configurations. It processes these configurations as JSON objects with the following structure: the top level is a list of objects where each follows the structure of the model that a single GPU hosts, with the tensor shapes of the model parameters as leaves. From these objects, the PTC and PTC' can be constructed.

5.2 Tensor Store

Each worker has an in-memory *Tensor Store* that contains the model and dataset partitions for each local GPU. The Tensor Store maintains the model and dataset tensors in a hierarchical in-memory file system. The tree hierarchy follows the

model structure, with model parameters and dataset samples as leaves.

Model state. The Tensor Store exposes an API to the *State Transformer* to apply a reconfiguration plan, and to the DL system to load/store the model state. It supports a NumPy-like array interface [21] for requesting tensors via a REST API: a query request with a path attribute obtains a tensor; an upload request adds a new tensor at a given path.

A unique feature of the API is that it can be used to request *sub-tensors* by defining a range for each dimension, similar to a Python slice. Requesting sub-tensors is important for performance, e.g. when re-slicing under model parallelism, because it reduces data movement between workers. The *State Transformer* can request sub-tensors instead of complete tensors that would have to be split after transfer. To obtain a sub-tensor, a query includes a range attribute whose value specifies the dimension and range. For example, a query for the second dimension of a tensor is `range=[: , 2 : 4]`, which returns the sub-tensor for `[2, 4]`.

The Tensor Store uses a simple API to move model state in and out of the DL system. To obtain the current model state, `scalai.save(model, path)` maps a Python dictionary with the model state to its hierarchical representation; `scalai.load(path)` maps it back to a Python dictionary that can be consumed by the DL system.

The model state in the Tensor Store is represented as a hierarchical tree with node grouping parameters. For example, `"/2/embedding/weight"` is a weight parameter in the embedding layer of the model state partition 2. The leaves are sub-tensors, which are implemented as NumPy arrays to offer compatibility with most DL systems.

Dataset state. The training dataset consists of binary files with data samples, which either reside on the local disk or remote storage. Data samples are tensors and TENPLEX represents them as NumPy [21] (npz or npy) arrays.

TENPLEX also maintains a *dataset index* that maintains the locations of all data samples. Specifically, for each data sample, it holds the paths to the binary files and byte ranges within those files. Based on the dataset index, the *State Transformer* can repartition the dataset as necessary into partitions, each with its own indices. To read a data sample, the DL system invokes a data loader, which uses the partition-specific index to decide on the relevant binary file and byte range, reading the corresponding part of the file.

In contrast to the model state, the dataset is immutable and consumed sequentially. TENPLEX leverages this to improve performance by overlapping training and dataset fetching. It streams the dataset into the Tensor Store on the workers while training iterations take place. Since the data sample order is known at the beginning of an epoch, TENPLEX derives which samples to fetch first to unblock training.

In a typical cloud deployment, the training dataset is stored on remote storage, e.g. S3 [3] or other blob stores [17, 37].

For training to resume, the data must be accessible by the workers, but the network bandwidth to remote storage is typically lower than the inter-worker bandwidth [66]. To reduce the impact of this, TENPLEX tracks the location of data samples in the dataset index and distinguishes between remote and locally available data samples. It then prioritizes fetching samples from other workers and only uses the remote storage if samples are otherwise unavailable.

5.3 Fault tolerance

When workers or GPU devices fail during job execution, TENPLEX relies on the DL system to recover. After a failure, the job state is restored from the persisted checkpoints created by the DL system. Since a failure can be seen as a resource reduction, TENPLEX’s reconfiguration support can be used to resume a job immediately, without waiting for new GPU resources. TENPLEX thus resumes the job with fewer GPU devices but with an optimal new parallelization plan.

A deployment with TENPLEX is subject to the usual trade-off between checkpointing frequency and overhead: with infrequent checkpointing, some job progress is lost after failure. TENPLEX tries to avoid re-executing training steps due to stale checkpoints: for DL jobs with data parallelism, TENPLEX exploits that the model state is replicated among workers. As long as at least one model replica remains after a failure, the state can be retrieved from that GPU.

To accommodate frequent failures in a cluster during training, TENPLEX can replicate the model state in the Tensor Store across workers in a round-robin fashion, adding more state redundancy to the job. To obtain n replicas, the state is replicated to the Tensor Stores of the next n workers. If a worker fails and the state in the worker’s Tensor Store is lost, the state can be recovered from another worker.

6 Evaluation

We evaluate TENPLEX in the context of three use cases: supporting elastic scaling with multi-dimensional parallelism (§6.2), enabling job redeployment (§6.3), and handling failure recovery (§6.4). After that, we investigate TENPLEX’s reconfiguration overhead (§6.5), the impact of different parallelization strategies (§6.6), and its scalability in terms of cluster size (§6.7). We finish by exploring TENPLEX’s effect on model convergence when changing parallelism (§6.8).

6.1 Experimental setup

Our experiments have the following setup:

Cluster. We conduct on-premise experiments with 16 GPUs (4 machines with 4 GPUs each). Each machine has an AMD EPYC 7402P CPU, $4 \times$ NVIDIA RTX A6000 GPUs, and PCIe 4.0. The machines are interconnected with 100-Gbps InfiniBand, and the GPUs use third-generation NVLink. We also conduct 32-GPU cloud experiments on Azure [37] with Standard_NC24s_v3 VMs, each with 4 NVIDIA V100 GPUs.

Baselines. We compare TENPLEX to multiple external base-

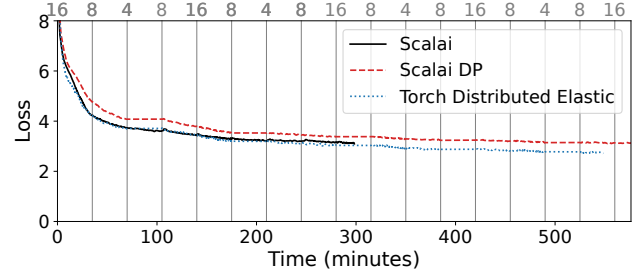


Fig. 9: Elastic DL job convergence with multi-dimensional parallelism under dynamic GPU changes

lines: (i) Torch Distributed Elastic v2.0 and (ii) Horovod-Elastic v0.28 [25], which are state-of-the-art elastic DL systems; and (ii) DeepSpeed v0.6 [55] with Megatron-LM v23.06, which represents the model library approach. All of these solutions can only support dynamic reconfiguration of DL jobs while changing the degree of data parallelism. We therefore also compare with (iv) TENPLEX-DP, which only reconfigures data parallelism, and (v) TENPLEX-Central that performs all state repartitioning at a central node.

Models and datasets. We use these representative DNN models: (i) *BERT-large* with 340M parameters; (ii) *GPT-3* with 1.3B (XL), 2.7B, and 6.7B parameters; and (iii) *ResNet-50* with 25M parameters. For the training data, we use: (i) *OpenWebText* [16] with 2M samples with a sequence length of 1024; (ii) *Wikipedia* [13] with 6.8M samples and the same sequence length; and (iii) *ImageNet* [9] with 1M samples.

6.2 Elastic multi-dimensional parallelism

First, we explore the benefits of supporting elasticity in DL jobs with multi-dimensional parallelism, scaling across all parallelism dimensions when the GPU allocation changes.

In this experiment, we train DL jobs with the GPT-3 XL model on the on-premise 16-GPU cluster. The job runtime and elastic scaling events are derived based on Microsoft’s Philly trace [28]: over the runtime of 538 mins, we scale based on the average every 35 mins. During a scaling event, we change the number of GPUs for a job between 16, 8, and 4 GPUs.

We compare the training convergence of TENPLEX to TENPLEX-DP and Torch Distributed Elastic. TENPLEX-DP is similar Torch Distributed Elastic with Megatron-LM by only scaling dynamically along the data parallelism dimension.

In terms of the scaling decisions, TENPLEX reconfigures the (model, pipeline, data) parallelism from $(M, P, D) = (2, 4, 2)$ to $(2, 2, 2)$ to $(2, 1, 2)$, which are the parallelization configurations that achieve the best performance. TENPLEX-DP and Torch scale $(M, D, P) = (2, 4, 2)$ to $(2, 4, 1)$, and pause training with 4 GPUs, because a configuration with pipeline parallelism of 4 and model parallelism of 2 cannot run on 4 GPUs.

Fig. 9 shows the loss over time, and the scaling events are indicated as grey vertical lines, annotated by the GPU count. As we can see, TENPLEX only takes 298 mins to reach the same step that TENPLEX-DP reaches after 576 mins and Torch

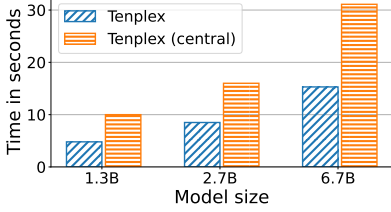


Fig. 10: Redeployment time of DL job

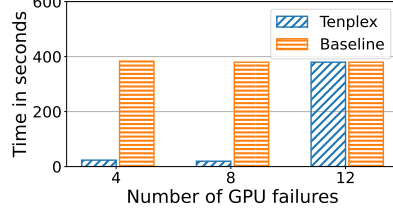


Fig. 11: Failure recovery time (GPT-3 2.7 B)

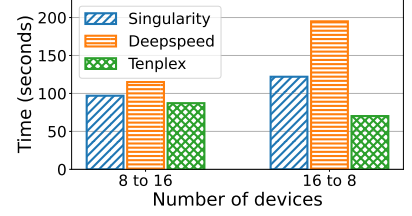


Fig. 12: Reconfiguration time against DeepSpeed and Singularity

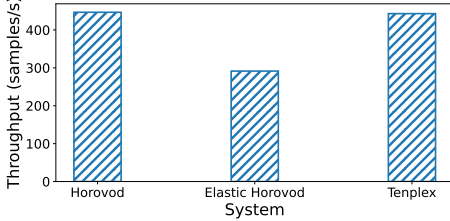


Fig. 13: Reconfiguration time against Horovod

after 548 mins—a reduction by 46%. Since TENPLEX can support scaling along all dimensions in DL jobs, it exploits more optimal parallelization configuration using the GPU resources more effectively.

6.3 Job redeployment

Next, we evaluate how long TENPLEX takes to redeploy DL jobs with different model sizes onto a new set of GPU resources. As a baseline, we compare against TENPLEX-Central, which follows the approach of PyTorch Elastic [49] or DeepSpeed [55]: it holds all DL job state at a single central worker. In this experiment, we therefore specifically explore the benefit of TENPLEX’s distributed state management.

We redeploy a DL job with multi-dimensional parallelism from one set of 8 GPUs to another 8 GPUs. We measure the redeployment time on the on-premise cluster with the GPT-3 model with sizes of 1.3 B, 2.7 B, and 6.7 B. The parallelization configuration is $(M, D, P) = (4, 2, 1)$ and remains the same, because the number of GPUs remain unchanged.

Fig. 10 shows the redeployment time under different model sizes. In all cases, TENPLEX achieves a lower redeployment time than TENPLEX-Central: the time for TENPLEX-Central is $2.1\times$ for the 1.3 B model, $1.9\times$ for 2.7 B model, and $2\times$ for 6.7 B model, respectively, higher compared to TENPLEX. With its distributed state management between State Transformer instances on different workers, TENPLEX can migrate state directly between workers. This prevents the network bandwidth of any single worker from becoming a bottleneck, which would increase redeployment time.

6.4 Failure recovery

We explore how TENPLEX manages to recover efficiently from failures, even in scenarios that require dynamic reconfiguration due to a change in the number of GPUs. We emulate faults of 4, 8, and 12 GPUs and measure the failure recovery and reconfiguration time. We use the GPT-3 2.7 B model with the Wikipedia dataset on the on-premise cluster. We com-

pare TENPLEX to a system that always recovers from the last checkpoint (denoted as Baseline), which results in an average loss of 50 training steps. The parallelization configuration is $(M, P, D) = (4, 2, 2)$, i.e. there are two model replicas.

Fig. 11 shows that the recovery time in seconds with different numbers of failed GPUs. TENPLEX recovers faster than the baseline if there exists at least one model replica, i.e. for failures with 4 and 8 GPUs. Here, TENPLEX does not need to rerun the lost training steps, because it does not rely on the stale checkpointed state for recovery. With 8 GPUs, TENPLEX takes only 5% of the recovery time of the baseline, and exhibits the same cost as for 12 GPUs.

When there is no redundant model replica available, TENPLEX uses the last checkpoint and only achieves a slight performance benefit over the baseline. This is due to using local storage instead of remote storage when recovering from the checkpointed state. We conclude that TENPLEX reduces failure recovery times when model replicas are available due to the parallelization configuration.

6.5 Reconfiguration overhead

This experiment compares the reconfiguration approach of TENPLEX with (i) a model library of an elastic DL system (DeepSpeed) and (ii) a virtual device approach that performs full GPU state migration (Singularity).

We use the GPT-3 XL model with the Wikipedia dataset on the on-premise cluster. We perform one experiment that scales down resources from 16 to 8 GPUs and another that scales up from 8 to 16 GPUs. Since Singularity [60] is a closed-source system, we report numbers from a similar experiment in its paper, run on similar hardware.

Fig. 12 shows the reconfiguration time. When changing from 8 to 16 GPUs, TENPLEX requires 24% less time than DeepSpeed and 10% less time than Singularity. Singularity is slower, because, besides the training state, it also moves the full GPU device state. DeepSpeed suffers from the fact that it does not include an explicit mechanism for notifying DeepSpeed about reconfiguration, but instead uses its failure detection mechanism. The state management approach of TENPLEX is the fastest, because it minimizes state movement due to its awareness of data locality.

The difference becomes larger when scaling from 16 to 8 GPUs: TENPLEX needs 64% less time than DeepSpeed and 43% less than Singularity. In this case, DeepSpeed relies on Torch Distributed Elastic’s failure mechanism, which in-

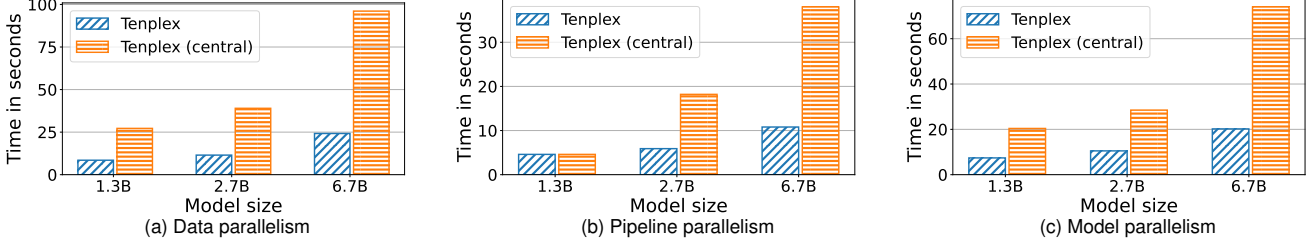


Fig. 14: Reconfiguration time with different parallelizations

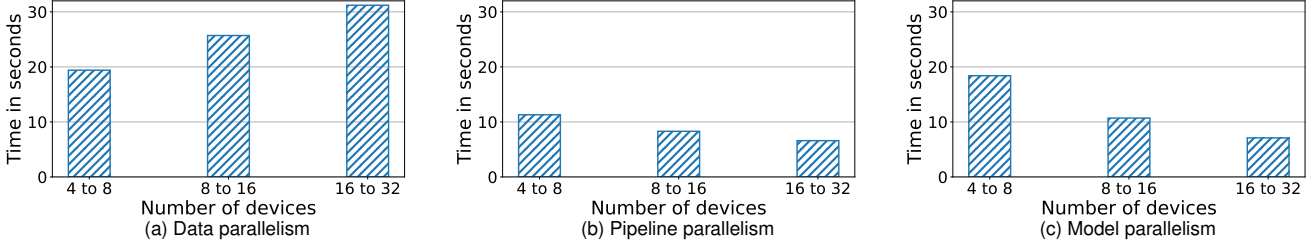


Fig. 15: Reconfiguration time with different cluster sizes

creases time; Singularity must copy the full GPU state even though there already is a model replica on the GPUs.

We also compare TENPLEX’s overhead to Horovod, a distributed training library without elasticity support, and Horovod-Elastic, which also supports scaling under data parallelism only by periodically checkpointing the model state. We deploy a ResNet50 model with the ImageNet dataset in the on-premise cluster, and measure throughput when training on 2 GPUs.

Fig. 13 compares the training throughput, measured as samples per second, for Horovod, Elastic Horovod, and TENPLEX. Horovod achieves 446 images/s, but with elasticity support, the throughput reduces to around a third (291 images/s). In contrast, TENPLEX has a throughput of 443 images/s, which is about the same as regular Horovod without elasticity support. Therefore, there is no overhead for TENPLEX and an improvement of 34% compared to Elastic Horovod.

TENPLEX outperforms Horovod Elastic due to its tight integration with the DL system: unlike Horovod Elastic, it avoids explicit checkpointing of the state in a blocking manner, which would interrupt training progress when performing a reconfiguration after a resource change.

6.6 Impact of parallelization type

Next, we examine the impact of the parallelization configuration on reconfiguration time for different model sizes. We deploy TENPLEX and TENPLEX-Central, which manages the state in a single node, with the different GPT-3 models on the on-premise cluster. For data parallelism (D), we change the configuration from $(M, P, D) = (4, 2, 1)$ to $(4, 2, 2)$; for pipeline parallelism (P) from $(4, 2, 1)$ to $(4, 4, 1)$; and for model parallelism (M) from $(4, 2, 1)$ to $(8, 2, 1)$.

Fig. 14 shows the reconfiguration time for the different parallelization configurations and model sizes. With data parallelism (Fig. 14a), TENPLEX-Central with GPT-3 6.7 B takes

$4\times$ longer than TENPLEX, because of the limited network bandwidth of a single worker in comparison with a distributed peer-to-peer state reconfiguration. We observe similar behavior for pipeline and model parallelism: under pipeline parallelism (Fig. 14b), TENPLEX-Central takes $3.5\times$ longer and, under model parallelism (Fig. 14c), it takes $3.7\times$ longer. The only exception is pipeline parallelism with 1.3 B parameters. In this case, network bandwidth does not become a bottleneck, because the parallelization configuration does not involve splitting and merging sub-tensors.

We conclude that centralized state management becomes a bottleneck with many model parameters, due to the limited network bandwidth and the reduced parallelism when all state transformations are performed by one worker.

6.7 Impact of cluster size

We want to explore how TENPLEX is affected by the GPU cluster size. In this experiment, we keep the model size fixed but change the GPU resources in the cluster to evaluate how the cluster size and parallelization configuration impact reconfiguration time.

We use the GPT-3 XL on the Wikipedia dataset deployed in the 32-GPU cloud testbed. We scale the resources from 4 to 8, 8 to 16, and 16 to 32 GPUs for data, model, and pipeline parallelism, respectively. For each parallelization configuration, if the number of GPUs doubles, the degree of parallelism also doubles. We compare TENPLEX with the baseline TENPLEX-Central, as it is the only baseline that supports full multi-dimensional parallelism.

Fig. 15 shows the reconfiguration time with different device counts. For data parallelism (Fig. 15a), the time increases linearly with the number of GPUs, because the number of model replicas is proportional to the parallelism degree; for pipeline parallelism (Fig. 15b), the reconfiguration time decreases with the number of devices, because the model size

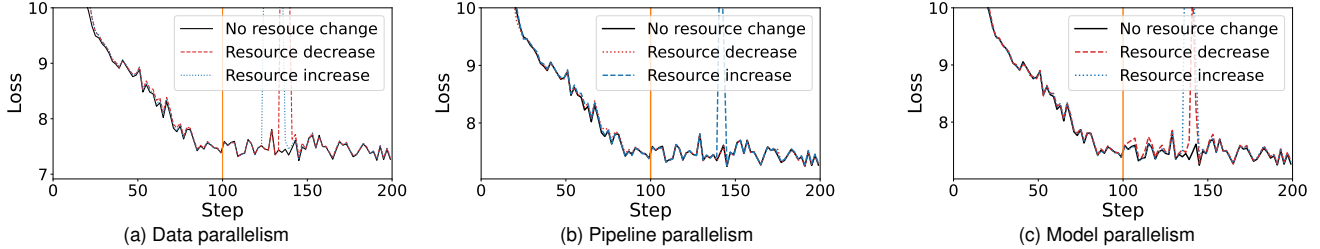


Fig. 16: Model convergence with reconfiguration

is constant and the total network bandwidth increases with the GPU count; for model parallelism (Fig. 15c), the time decreases with the GPU count, because the model size is constant and the network bandwidth increases with the devices.

Comparing data, model, and pipeline parallelism, the reconfiguration time is the highest with data parallelism, because the amount of data increases with the number of replicas. While the amount of data stays constant with pipeline and model parallelism, model parallelism must split and merge sub-tensors. The reconfiguration time is lowest with pipeline parallelism, which only needs repartitioning.

6.8 Impact on model convergence

Finally, we evaluate TENPLEX’s impact on model convergence. For this, we use the BERT-large model with the OpenWebText dataset deployed on the on-premise cluster. At training step 100, we either increase or decrease the resources and compare them to a baseline without change.

Fig. 16 shows the model convergence as the loss over the training steps. With data parallelism (Fig. 16a), i.e. changing the parallelization configuration from (1, 1, 4) to (1, 1, 8), the loss does not diverge when the resources increase/decrease because TENPLEX maintains consistent hyper-parameters and consistent data order. With pipeline parallelism (Fig. 16b), i.e. changing from (1, 4, 1) and (1, 8, 1), and with model parallelism (Fig. 16c), i.e. changing from (4, 1, 1) and (8, 1, 1), convergence is equally unaffected.

7 Related Work

Elastic ML systems support resource changes but only focus on data parallelism by adding/removing model replicas to/from GPU workers. KungFu [36] and Horovod [57] dynamically change the number of workers by adjusting the micro-batch size per worker, keeping the global batch size constant. This requires over-provisioning for the maximum worker count. In the same vein, DeepPool [45] frees up underutilized GPUs and allocates them to other jobs.

Hydrozoa [20] supports static multi-dimensional parallelism before training, but only allows for dynamic changes to data parallelism. Varuna [4] needs the user to define partitioning points for pipeline parallelism. It supports data parallelism, but it is missing support for model parallelism to handle arbitrary multi-dimensional parallelism. GoldMiner [73] focuses on adapting data preprocessing in ML systems, but it does not support multi-dimensional parallelism.

Dynamic GPU scheduling systems, e.g. Lyra [33], adjust the numbers of GPUs assigned to jobs. They, however, rely on elastic ML libraries for the reconfiguration of training jobs, making them complementary to TENPLEX.

Checkpointing systems store snapshots of model parameters. When resources change, e.g. after failure, the DL system retrieves the latest checkpoint before the failure and resumes training. CheckFreq [39] dynamically adjusts the checkpointing frequency, while Check-N-Run [11] uses lossy compression, trading accuracy for storage efficiency. Similarly, Gemini [66] and Oobleck [27] provide fast checkpointing: while the former stores checkpoints in CPU memory, the latter maintains checkpoints in GPU memory. These approaches, however, focus only on the performance of failure recovery, as opposed to the generic runtime reconfiguration of TENPLEX.

DL job parallelization. DL systems have built-in parallelization support: PyTorch [46] and Tensorflow [1] offer data parallelism; JAX [14] has *pmap* and *xmap* functions to parallelize computation; and MindSpore [38] uses *auto parallel* search to find an effective parallelization strategy. All of these approaches only support a subset of multi-dimensional parallelism and do not offer runtime reconfiguration.

Unity [64], similar to Alpa [74], searches for an optimal distribution plan and implements a suitable runtime for it. TENPLEX reuses such parallelizers when making reconfiguration decisions. Google XLA [56] and MindSpore include resharding operators, which are needed for automatic parallelism. These operators, however, are applied to a single GPU instead of a GPU cluster. They thus cannot handle the reconfiguration of distributed DL jobs with multi-dimensional parallelism, as supported by TENPLEX.

8 Conclusion

We described TENPLEX, a dynamic state management library for DL jobs with multi-dimensional parallelism. By describing the state as a parallelizable tensor collection (PTC), TENPLEX generates efficient reconfiguration plans when the underlying GPU resources for the job change at runtime. Its distributed state transformers implement the reconfiguration plan on each GPU with a minimum amount of data movement between workers. Therefore, TENPLEX is a step towards making large-scale long-running deep learning jobs fully adaptive to resource changes.

References

- [1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, and Michael Isard. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.
- [2] Samson B. Akintoye, Liangxiu Han, Xin Zhang, Haoming Chen, and Daoqiang Zhang. A Hybrid Parallelization Approach for Distributed and Scalable Deep Learning. *IEEE Access*, 10:77950–77961, 2022.
- [3] Amazon. Cloud Object Storage - Amazon S3. <https://aws.amazon.com/pm/serv-s3/>.
- [4] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: Scalable, Low-Cost Training of Massive Deep Learning Models. In *Proceedings of the Seventeenth European Conference on Computer Systems, EuroSys '22*, page 472–487, New York, NY, USA, 2022. Association for Computing Machinery.
- [5] AWS. SageMaker Distributed Model Parallelism Best Practices. <https://docs.aws.amazon.com/sagemaker/latest/dg/model-parallel-best-practices.html>.
- [6] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, et al. Pathways: Asynchronous distributed dataflow for ml. *Proceedings of Machine Learning and Systems*, 4:430–449, 2022.
- [7] 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. Language Models are Few-Shot Learners. In Hugo Larochelle, Marc’Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.
- [8] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc’Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. Large Scale Distributed Deep Networks. In Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, pages 1232–1240, 2012.
- [9] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.
- [10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Tamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.
- [11] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. Check-N-Run: a Checkpointing System for Training Deep Learning Recommendation Models. In Amar Phanishayee and Vyas Sekar, editors, *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, pages 929–943. USENIX Association, 2022.
- [12] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural Acceleration for General-Purpose Approximate Programs. *IEEE Micro*, 33(3):16–27, 2013.
- [13] Wikimedia Foundation. Wikimedia Downloads. <https://dumps.wikimedia.org>.
- [14] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 4(9), 2018.
- [15] Patrick Fu. GPU Partitioning: Fair Share Scheduling. <https://www.geminioopencloud.com/en/blog/gpu-partitioning-fair-scheduling/>, 2022.
- [16] Aaron Gokaslan and Vanya Cohen. OpenWebText Corpus. <http://Skyllion007.github.io/OpenWebTextCorpus>, 2019.
- [17] Google. Google Cloud. <https://cloud.google.com/>.

- [18] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyröla, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *CoRR*, abs/1706.02677, 2017.
- [19] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. DeepFM: A Factorization-Machine based Neural Network for CTR Prediction. In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 1725–1731. ijcai.org, 2017.
- [20] Runsheng Guo, Victor Guo, Antonio Kim, Josh Hildred, and Khuzaima Daudjee. Hydrozoa: Dynamic Hybrid-Parallel DNN Training on Serverless Containers. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 779–794, 2022.
- [21] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020.
- [22] Tomoki Hayashi, Ryuichi Yamamoto, Katsuki Inoue, Takenori Yoshimura, Shinji Watanabe, Tomoki Toda, Kazuya Takeda, Yu Zhang, and Xu Tan. Espnet-TTS: Unified, Reproducible, and Integratable Open Source End-to-End Text-to-Speech Toolkit. In *2020 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2020, Barcelona, Spain, May 4-8, 2020*, pages 7654–7658. IEEE, 2020.
- [23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016.
- [24] Marius Hobbhahn and Tamay Besiroglu. Trends in GPU Price-Performance, 2022.
- [25] Horovod. Elastic Horovod. https://horovod.readthedocs.io/en/latest/elastic_include.html.
- [26] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhiheng Chen. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d’Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 103–112, 2019.
- [27] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient Distributed Training of Large Models Using Pipeline Templates. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 382–395. ACM, 2023.
- [28] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, Renton, WA, July 2019. USENIX Association.
- [29] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.
- [30] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gotipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datcenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*, pages 1–12. ACM, 2017.
- [31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In Peter L. Bartlett, Fernando C. N.

- Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, pages 1106–1114, 2012.
- [32] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. Aryl: An Elastic Cluster Scheduler for Deep Learning, 2022.
- [33] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. Lyra: Elastic scheduling for deep learning clusters. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 835–850, 2023.
- [34] Mingzhen Li, Wencong Xiao, Biao Sun, Hanyu Zhao, Hailong Yang, Shiru Ren, Zhongzhi Luan, Xianyan Jia, Yi Liu, Yong Li, Depei Qian, and Wei Lin. EasyScale: Accuracy-consistent Elastic Training for Deep Learning, 2022.
- [35] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. PyTorch distributed. *Proceedings of the VLDB Endowment*, 13(12):3005–3018, 2020.
- [36] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. KungFu: Making Training in Distributed Machine Learning Adaptive. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 937–954, 2020.
- [37] Microsoft. Microsoft Azure. <https://azure.microsoft.com/>.
- [38] MindSpore. Mindspore Deep Learning Training/Inference Framework. <https://github.com/mindspore-ai/mindspore>, 2020.
- [39] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 203–216, 2021.
- [40] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. PipeDream: generalized pipeline parallelism for DNN training. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSOP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 1–15. ACM, 2019.
- [41] 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. Efficient large-scale language model training on GPU clusters using megatron-LM. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, page 58, 2021.
- [42] Bogdan Nicolae, Jiali Li, Justin M. Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. DeepFreeze: Towards Scalable Asynchronous Checkpointing of Deep Learning Models. In *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGRID 2020, Melbourne, Australia, May 11-14, 2020*, pages 172–181. IEEE, 2020.
- [43] OpenAi. Introducing ChatGPT. 2022.
- [44] Andrew Or, Haoyu Zhang, and Michael J Freedman. VirtualFlow: Decoupling Deep Learning Models from the Underlying Hardware. *arXiv preprint arXiv:2009.09523*, 2020.
- [45] Seo Jin Park, Joshua Fried, Sunghyun Kim, Mohammad Alizadeh, and Adam Belay. Efficient Strong Scaling Through Burst Parallel Training. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 748–761, 2022.
- [46] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, and Luca Antiga. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in neural information processing systems*, 32, 2019.
- [47] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys ’18, New York, NY, USA, 2018*. Association for Computing Machinery.
- [48] Shashank Prasanna. Train Deep Learning Models on GPUs using Amazon EC2 Spot Instances. <https://aws.amazon.com/blogs/machine-learning/train-deep-learning-models-on-gpus-using-amazon-ec2-spot/>, 2019.
- [49] PyTorch. Torch Elastic. <https://pytorch.org/elastic/latest/>.
- [50] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive

- Cluster Scheduling for Goodput-Optimized Deep Learning. In Angela Demke Brown and Jay R. Lorch, editors, *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. USENIX Association, 2021.
- [51] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. 2019.
- [52] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. Large-scale deep unsupervised learning using graphics processors. In Andrea Pohoreckyj Danyluk, Léon Bottou, and Michael L. Littman, editors, *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*, volume 382 of *ACM International Conference Proceeding Series*, pages 873–880. ACM, 2009.
- [53] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. DeepSpeed-MoE: Advancing Mixture-of-Experts Inference and Training to Power Next-Generation AI Scale. In *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, pages 18332–18346, 2022.
- [54] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. ArXiv, May 2020.
- [55] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 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 and Data Mining*, pages 3505–3506, 2020.
- [56] Amit Sabne. Xla: Compiling machine learning for peak performance. *Google Res*, 2020.
- [57] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.
- [58] Chris Shallue and George Dahl. Measuring the Limits of Data Parallel Training for Neural Networks. <https://blog.research.google/2019/03/measuring-limits-of-data-parallel.html>, 2019.
- [59] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *CoRR*, abs/1909.08053, 2019.
- [60] Dharma Shukla, Muthian Sivathanu, Srinidhi Viswanatha, Bhargav Gulavani, Rimma Nehme, Amey Agrawal, Chen Chen, Nipun Kwatra, Ramachandran Ramjee, Pankaj Sharma, Atul Katiyar, Vipul Modi, Vaibhav Sharma, Abhishek Singh, Shreshth Singhal, Kaustubh Welankar, Lu Xun, Ravi Anupindi, Karthik Elangovan, Hasibur Rahman, Zhou Lin, Rahul Seetharaman, Cheng Xu, Eddie Ailijiang, Suresh Krishnappa, and Mark Russinovich. Singularity: Planet-Scale, Preemptive and Elastic Scheduling of AI Workloads, 2022.
- [61] Samuel L. Smith and Quoc V. Le. A Bayesian Perspective on Generalization and Stochastic Gradient Descent. In *International Conference on Learning Representations*, 2018.
- [62] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, Elton Zheng, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model. *CoRR*, abs/2201.11990, 2022.
- [63] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 2818–2826. IEEE Computer Society, 2016.
- [64] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efraim Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Patrick S. McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. Unity: Accelerating DNN Training Through Joint Optimization of Algebraic Transformations and Parallelization. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 267–284. USENIX Association, 2022.
- [65] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 59–72, Santa Clara, CA, February 2017. USENIX Association.

- [66] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. GEMINI: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP '23*, page 364–381, New York, NY, USA, 2023. Association for Computing Machinery.
- [67] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 945–960, Renton, WA, April 2022. USENIX Association.
- [68] Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. Beware of Fragmentation: Scheduling GPU-Sharing Workloads with Fragmentation Gradient Descent. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 995–1008, Boston, MA, July 2023. USENIX Association.
- [69] Yidi Wu, Kaihao Ma, Xiao Yan, Zhi Liu, Zhenkun Cai, Yuzhen Huang, James Cheng, Han Yuan, and Fan Yu. Elastic Deep Learning in Multi-Tenant GPU Clusters. *IEEE Transactions on Parallel and Distributed Systems*, 33(1):144–158, 2022.
- [70] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, Carlsbad, CA, October 2018. USENIX Association.
- [71] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548. USENIX Association, November 2020.
- [72] Shiwei Zhang, Lansong Diao, Chuan Wu, Siyu Wang, and Wei Lin. Accelerating Large-Scale Distributed Neural Network Training with SPMD Parallelism. In *Proceedings of the 13th Symposium on Cloud Computing, SoCC '22*, page 403–418, New York, NY, USA, 2022. Association for Computing Machinery.
- [73] Hanyu Zhao, Zhi Yang, Yu Cheng, Chao Tian, Shiru Ren, Wencong Xiao, Man Yuan, Langshi Chen, Kaibo Liu, Yang Zhang, et al. Goldminer: Elastic scaling of training data pre-processing pipelines for deep learning. *Proceedings of the ACM on Management of Data*, 1(2):1–25, 2023.
- [74] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, and Joseph E Gonzalez. Alpa: Automating Inter-and Intra-Operator Parallelism for Distributed Deep Learning. *arXiv preprint arXiv:2201.12023*, 2022.
- [75] Guorui Zhou, Xiaoqiang Zhu, Chengru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep Interest Network for Click-Through Rate Prediction. In Yike Guo and Faisal Farooq, editors, *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, pages 1059–1068. ACM, 2018.