TrainVerify: Equivalence-Based Verification for Distributed LLM Training

Yunchi Lu
University of Michigan
Microsoft Research Asia

Youshan Miao
Microsoft Research Asia

Cheng Tan
Northeastern University
Microsoft Research Asia

Peng Huang
University of Michigan

Yi Zhu
Microsoft Research Asia

Xian Zhang
Microsoft Research Asia

Fan Yang
Microsoft Research Asia

Abstract

Training large language models (LLMs) at scale requires parallel execution across thousands of devices, incurring enormous computational costs. Yet, these costly distributed trainings are prone to correctness bugs, causing silent errors and potentially wasting millions of GPU hours. These bugs are challenging to expose through testing.

We introduce TrainVerify, a system for *verifiable distributed training* of LLMs to *eliminate* parallelization bugs. Given a deep learning model's logical specification as the ground truth, TrainVerify formally verifies that a distributed parallel execution plan is mathematically equivalent to it. Direct verification is notoriously difficult due to the sheer scale of LLMs which often involves billions of variables and highly intricate computation graphs. Therefore, TrainVerify introduces a stage-wise parallel verification algorithm and shape-reduction techniques that significantly reduce complexity while preserving formal correctness. TrainVerify scales to frontier LLMs, including the successful verification of the Llama3 405B and DeepSeek-V3 671B training plans.

CCS Concepts: **• Computing methodologies → Machine learning**; **• Software and its engineering → Software verification**.

Keywords: deep neural network, model parallelization, equivalence checking, formal verification, symbolic execution

ACM Reference Format:
Yunchi Lu, Youshan Miao, Cheng Tan, Peng Huang, Yi Zhu, Xian Zhang, and Fan Yang. 2025. TrainVerify: Equivalence-Based Verification for Distributed LLM Training. In *ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP '25), October 13–16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3731569.3764850

SOSP '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1870-0/25/10
https://doi.org/10.1145/3731569.3764850

1 Introduction

Recent studies show that scaling a deep neural network (DNN) generally yields better performance [46]. This triggers an "arms race" for large models. Training large models requires extensive resources, necessitating distributed training over a growing number of GPUs [26, 27, 38]. Latest large language models (LLMs), for instance, are trained on clusters with thousands to tens of thousands of high-end GPUs for several weeks or months, costing millions of dollars [8, 34, 38].

Distributed training, however, is notoriously complex. It requires intricate spatial-temporal scheduling and coordination across a large set of devices using sophisticated parallelism techniques [48, 52, 65, 83]. The entire workflow is error-prone (§2.2). Worse still, the bug symptoms are often *silent*, such as causing wrong gradient updates and improper scaling of the model states, which are difficult to detect and debug.

In this context, the cost of bugs becomes particularly high. Even a single bug can lead to wasting significant resources, financial loss, and low productivity for developers [17, 38]. Moreover, an incorrectly trained model can have far-reaching consequences, especially in critical applications where model accuracy is paramount [37, 45, 68].

Given the high stakes, there is an urgent need to verify the correctness of distributed training. To achieve this goal, one option is to design a new training software stack that is correct-by-construction. While ideal, this is a daunting task from a verification perspective. Large-scale training involves a myriad of external dependencies as well as high concurrency, both of which are challenging to verify [1, 9, 32, 80, 82]. The incompatibility with well-established training libraries [2, 21] and potential performance loss will also pose practical barriers.

In this work, we explore an alternative direction and propose a methodology that aims to provide strong correctness guarantees without rebuilding the whole training stack.

Our key insight is that the correctness of distributed training can be rigorously verified at the level of **execution plans**. By establishing what we call **parallelization equivalence**, we prove that the *execution plan* of a DNN model is equivalent to its *logical definition*. In other words, for *every* input, the distributed execution must yield an output equivalent to that of the original model. This verifiability is achieved without running the training task at full scale.

Specifically, the *logical definition* of a DNN model can be represented as a data flow graph (DFG), where a node represents an operator like matrix computation, and an edge denotes a tensor produced by its upstream operators and consumed by its downstream operators. This representation is logical because operators and tensors are described mathematically. This logical form is usually scrutinized through rigorous math reasoning and empirical validations, providing assurance of correctness [75]. When developers define a model in code, deep learning frameworks still internally use or provide ways to expose a DFG representation [71, 72].

The parallelization of a model decides how operators and tensors in the model are partitioned and scheduled across devices using parallelism techniques. This implementation logic is captured by a materialized *execution plan* [52], which can be structured as a parallelized DFG. Training systems use this plan to orchestrate the execution of a training task.

Parallelization equivalence guarantees that, for any input, the parallelized DFG yields outputs equivalent to those of the logical DFG. Focusing on this property allows the verification effort to be manageable. It also supports existing frameworks, which makes the approach practical. At the same time, since the execution plan captures the essence of how a training task is parallelized, proving this property *eliminates* major classes of bugs that jeopardize the correctness of distributed training, providing strong guarantees (§8.3).

Several key research challenges arise: (1) *how to formulate the verification problem?* (2) *what representation to use to carry out the verification?* (3) *how to scale the verification to large DNNs that have hundreds of billions of parameters?*

To address these challenges, we first formally define parallelization equivalence and its verification (§3). We then design TrainVerify with multiple techniques to address the needs of representation and scalability.

- *Symbolic DFG* (§5.1): TrainVerify defines symbolic operators that encapsulate the mathematical foundations of modern deep learning operations. It converts a logical model and its execution plan into symbolic dataflow graphs and verifies the equivalence between these two representations.
- *Staged verification* (§5.2) addresses the intractability of verifying long symbolic expressions in DNNs by partitioning the DFGs into stages, each treated as a separate verification unit and verified concurrently. TrainVerify then chains these units into an end-to-end proof by verifying input-output equivalence for the dependent units and leveraging lineage metadata to ensure correctness across the entire network.
- *Shape reduction* (§5.3) mitigates scale explosion in large models by reducing tensor shapes while preserving their structural and functional properties. It enables verification on smaller, manageable shapes and provably extends the results to larger shapes with the same structure.

To our best knowledge, TrainVerify is the first to offer *provably correct execution plans* for distributed training. It has been integrated into nnScaler [52], a state-of-the-art training framework. We identify the changes needed to make nnScaler amenable to verification. Specifically, we enhance the DFGs to incorporate all training-related computations. In addition, verification requires *lineage* to effectively map tensor values between the logical DFG and its parallelized counterpart. This information is discarded after parallelization in nnScaler, but is preserved in TrainVerify to enable equivalence checking.

Experiments show that TrainVerify successfully verifies the executions plans for state-of-the-art large language models, Llama3 (8B/70B/405B) [38] and DeepSeek-V3 (16B/236B/671B) [36]. The verification finishes within minutes to hours for small and medium models, and up to half a day for the largest models. We also show that TrainVerify can detect and eliminate major classes of real-world parallelization bugs in distributed training. TrainVerify is open sourced at https://github.com/verify-llm/TrainVerify.

2 Background and Motivation

2.1 Distributed Training

At a high level, distributed training partitions the computational operations over high-dimensional data into multiple operations over smaller data, schedules the partitions among multiple devices *spatially*, executes them in a specific order *temporally*, and synchronizes state across different devices.

Different parallelization approaches exist, including *data parallelism*, *tensor parallelism*, and *pipeline parallelism* [6, 41, 67]. Large-scale training commonly employs a combination of these techniques, such as 3D parallelism [30, 38].

To parallelize a specific model, ML engineers typically adapt or handcraft detailed parallelization code on top of a library (*e.g.*, Megatron-Core [9]) that provides APIs for common parallelism techniques. Recent training frameworks [52, 83] also explore the search space automatically to generate efficient parallelization strategies for a given model. To produce a valid distributed program, they operate on computation graphs and apply a series of graph transformation passes, including partitioning tensors and operators, assigning them to devices, injecting inter-device communication and planning the execution schedule.

2.2 Error-Proneness of Parallelization

Distributed training is inherently error-prone due to its significant complexities. It requires correctly partitioning the operators and tensors, scheduling them into the right devices, inserting proper communication operators, coordinating execution in the right order, and ensuring that all of this preserves the original model's semantics under *diverse* training configurations and parallelization schemes.

Cause	MegatronLM	DeepSpeed	nnScaler
op-transformation	16	18	19
scheduling	4	1	4
communication	8	13	5
Total	26	28	25

Table 1. *Parallelization-specific bugs* in distributed training systems, based on commit logs and GitHub issues (categories may overlap).

Indeed, as Table 1 shows, we study three state-of-the-art distributed training systems [52, 66, 67] and find that they all encounter bugs *specific to parallelization* in nearly every aspect of the workflow. For instance, in distributed data loaders, bugs can cause improper slicing and uneven distribution of global batches [5]. In forward and backward propagation, an operator can be incorrectly transformed, or a tensor can be improperly partitioned [16]. Similarly, the calculation of metrics such as loss averaging and gradient norms can be affected if tensor shards are mistakenly treated as replicated or partitioned [4, 10, 15, 17]. Pipeline scheduling can also misorder operators from different micro-batches or misallocate gradient updates [11, 13, 14]. In addition, a synchronization step may miss or use incorrect communication operations, *e.g.*, assigning wrong GPU ranks as sources or destinations, or using the wrong type of primitive [3, 12, 18, 19].

2.3 The Need for Rigorous Parallelization

What makes parallelization errors particularly concerning is that they are often *silent*, so the training appears deceptively normal, and the impact only becomes apparent later. For instance, incorrect gradient synchronization might only affect certain layers. These errors can evade detection until weeks into training, wasting valuable resources. Such errors are also challenging for developers to diagnose and debug. For example, a subtle incorrect loss scaling bug in MegatronLM took developers and users months of discussions to pinpoint [17].

It is also challenging for traditional testing to expose parallelization bugs. Deep learning relies on floating-point computation, which naturally introduces value drift during parallel training due to variations in the order of operations [69, 79]. This drift is exacerbated by mixed-precision training, which employs low-precision data types, and the use of diverse kernel implementations for the same operator due to efficiency. The numerical instability makes it challenging to distinguish normal numerical noise from actual errors, undermining methods like differential testing that directly compare numerical results between unparallelized and parallelized training [56, 58]. For large-scale training that involves thousands of GPUs, it is also infeasible to obtain another full-scale single-device training result for comparison. Using a smaller-scale testing can easily miss bugs that only appear in full-scale training.

Given the importance of distributed training, the financial costs of training failures, the consequences of an incorrectly trained model, as well as the challenges in detection and diagnosis, it is imperative to provide formal correctness guarantees for parallelization. In other words, we should aim for *eliminating* parallelization bugs.

2.4 Insight: Verifying Parallelization Equivalence

While verifying the entire distributed training stack would be ideal, doing so is impractical. The stack includes numerous interacting components such as DNN compilers, kernel libraries, collective-communication runtimes, *etc.*, each of which is complex and often depends on opaque hardware vendor code [1, 9, 32, 80, 82]. The efforts required for verification would be prohibitively high, and compatibility and performance will also likely be sacrificed. Moreover, new parallelism techniques and operator optimizations emerge rapidly, making it difficult to maintain the verified system.

To alleviate this complexity, our key insight is that crucial correctness of distributed training resides in the **execution plan**—a representation of a full training iteration of the distributed model—generated by the parallelization framework. Once the plan is fixed, subsequent runtime merely executes the prescribed operations. Based on this insight, we shift the verification focus from the system to **parallelization equivalence (PE)**: the execution plan is equivalent to the model's logical definition. Both the execution plan and logical definition can be represented as data flow graphs (DFGs). PE requires that for all inputs, executing the parallelized DFG yields outputs that are identical to those produced by the logical DFG.

This verification approach offers several benefits. First, it can support existing training frameworks. As will be introduced in §5.1, it mainly requires the framework to provide a DFG representation for the parallel execution plus lineage information. In our experience, this is not difficult. Second, the state space to verify becomes more tractable. We reason about one concrete plan each time, not the enormous space of all possible plans. Third, the verification of execution plan is less intrusive. Focusing on the output execution plan makes verification largely agnostic to internal transformations, improving transferability and robustness to framework evolution. Moreover, PE reasoning is symbolic over *real arithmetic*, making it immune to numerical noise. Finally, PE provides strong correctness guarantees: it covers all admissible inputs, and an execution plan faithfully captures the computation and communication logic involved in a training iteration. Frameworks such as Alpa [83] and nnScaler [52] further materialize an execution plan into executable code. If PE holds, the distributed run is *functionally indistinguishable* from the scrutinized logical model; if it fails, some bug must be present.

```python
def require_allreduce_dgrad(cfg) -> Bool: pass
def get_model_tp_group(cfg) -> ProcessGroup: pass
class LinearWithFrozenW(torch.autograd.Function):
    def backward(self, gy):
        gx = gy.matmul(self.w)
        if require_allreduce_dgrad(self.cfg):
            torch.distributed.all_reduce(gx,
                group=get_model_tp_group(self.cfg))
        return gx
```

Figure 1. Code snippet of a backward computation.

3 Parallelization Equivalence

In this section, we formulate and define the problem of verifying *parallelization equivalence*.

Formulation. Let $f : \mathcal{X} \to \mathcal{Y}$ denote a logical neural network model, assuming for single-device execution. $\mathcal{X}$ and $\mathcal{Y}$ denote the universe of input and output tensors, respectively. A parallelization procedure $\mathcal{P}(\cdot)$ transforms f into a model g that executes on multiple devices. g is parallelization equivalent to f iff $\forall x \in \mathcal{X}, g(x) = f(x)$.

While the problem is theoretically broad and intractable—f may involve arbitrary operators and $\mathcal{P}$ can be extremely complex—it is constrained in practice. We focus on f as valid neural networks, *e.g.*, transformer-based models, and $\mathcal{P}$ as feasible parallelization plans.

Operations in f include, but are not limited to, matrix operations (`Linear`, `MatMul`, `Transpose`, `Reshape`), activations (`Softmax`, `SiLU`), and others (`Layernorm`, `Embedding`). Parallelization $\mathcal{P}$ covers data parallelism [6], tensor parallelism [67], pipeline parallelism [41, 61], and their combinations. Expert [64], context [55], and sequence [51] parallelisms fall within our broader definition of tensor parallelism. $\mathcal{P}$ is applied to f by inserting partitioning and communication operators (`Chunk`, `AllReduce`, `AllToAll`, etc).

Graph Representation. Both f and g can be encoded as data flow graphs (DFGs), where nodes represent operators and edges carry tensors. Then, the parallelization procedure $\mathcal{P}$ transforms the logical DFG denoted as $G_l := (V_l, E_l)$ into a parallelized DFG denoted as $G_p := (V_p, E_p)$, by partitioning or replicating operators and tensors, and adding aggregation and communication operators. $\mathcal{P}$ assigns and orders G_p to multiple devices. Let $\mathcal{M} : E_l \to {E_p}^{\mathbb{N}}$ denote the mapping between each logical tensor and its partitioned counterparts.

Definition 1 (Parallelization Equivalence)**.** *G_p is parallelization equivalent to G_l iff: $\forall t \in E_l, \forall x \in \mathcal{X}, t(x) = \bigoplus_{t_i \in \mathcal{M}(t)} t_i(x)$, where $t(x)$ denotes the concrete value of a tensor t given the concrete input x. $\bigoplus$ is a composition operation determined by the parallelization scheme. In practice, $\bigoplus$ typically instantiates as duplication (identity), all-reduce summation (+), or tensor concatenation.*

We refer to G_l as **logical model**, and G_p as the **execution plan** for the distributed training task. Both encompass all operations within a training iteration, including all micro-batches' forward passes, backward passes, optimizer steps, and metric computations.

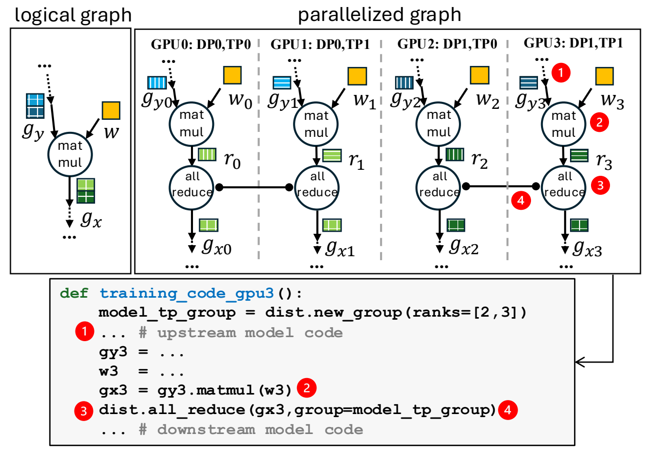


Figure 2. Logical and parallelized data flow graphs (simplified) for Figure 1 and the distributed training code for GPU 3.

Example. To illustrate how parallelization equivalence applies in practice, consider a custom autograd layer during back propagation, shown in Figure 1. It multiplies the upstream gradient g_y with the fixed weight w to obtain the gradient g_x. Figure 2 depicts the corresponding logical DFG.

Assume we train the same layer on four GPUs using 2-way data parallelism and 2-way tensor parallelism. Each GPU computes a partial result r_i for the partitioned tensors. As Figure 2 shows, the parallelized DFG introduces an `AllReduce` operation to aggregate r_i across tensor-parallel replicas.

Parallelization equivalence asserts that all the following algebraic equalities must hold for all valid inputs:

- $E_1 : g_y$[`lower-half`] $== g_{y2} + g_{y3}$
- $E_2 : g_x$[`lower-half`] $== r_2 + r_3$
- $E_3 : g_x$[`lower-half`] $== g_{x2} == g_{x3}$

These invariants are not just abstract properties; they capture classes of real-world bugs. To name a few:

- *Inconsistent tensor partitioning*, due to bugs when producer and consumer layers partition the same tensor inconsistently ❶. They will violate E_1.
- *Incorrect computation operator parallelization*. For example, if the operator is a custom operator, more complex than `MatMul`, involving rank-dependent computation, computation logic bugs can easily occur in ❷, violating E_2.
- *Missing or wrong communication operator*. For example, if `require_allreduce_dgrad` mistakenly returns `False`, the necessary `AllReduce` operation at ❸ will be missing. Similarly, incorrect primitives may be introduced, *e.g.*, `AllGather`. All those cases will violate E_3.
- *Incorrect communication group*. For example, `get_model_tp_group` may be mistakenly assigned as a global group at ❹, *i.e.*, `(0,1,2,3)`. They will also violate E_3.

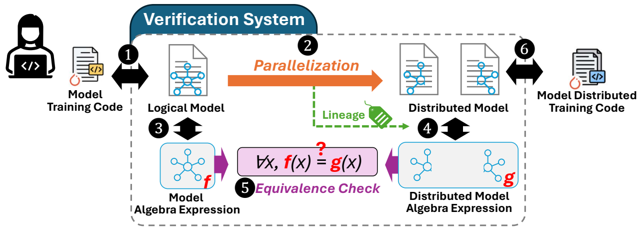


Figure 3. TrainVerify System Overview.

4 Overview of TrainVerify

To realize our verification methodology for DNN models in practice, we develop TrainVerify, a system that provides provably correct execution plans for distributed training.

As shown in Figure 3, developers write the code to describe the logical definition of a neural network. This code is usually device independent or assumes running on a single virtual device with infinite power. The availability of a single-device definition aligns with common development practices. Such code essentially represents a mathematical form of the model and can be rigorously scrutinized, reflecting the intended semantics of the model. To scale training to multiple GPUs, developers create distributed training code, either implemented or by leveraging auto-parallelization frameworks [52, 83].

TrainVerify works in three steps. It takes both versions of training code as inputs and extracts two data flow graphs (DFGs) (❶, ❻) via tracing. Nodes in the DFGs represent operations such as `MatMul`, `AllReduce`, etc., while edges carry data tensors. One (logical) DFG defines the logical model, while the other parallelized DFG encodes how the model is partitioned and scheduled for distributed execution.

Next, TrainVerify symbolizes each graph by converting tensors into symbolic ones, and executable operations into symbolic computation (❸, ❹). This symbolic abstraction allows TrainVerify to reason about the underlying algebraic semantics of the computation without being limited by specific data or affected by numeric noises. TrainVerify uses lineage metadata (❷) to track how tensors and operations in the parallelized graph correspond to their counterparts in the logical model.

Finally, TrainVerify checks the parallelization equivalence (PE) for the two symbolic DFGs by constructing formulas representing PE and using an SMT solver to formally verify that these formulas satisfy for all inputs (❺). If PE does not hold, TrainVerify can output counter-examples.

Scope. To make the verification tractable, TrainVerify has a focused scope. It treats the logical model as the specification and assumes it represents the desired semantics. It performs verification at the level of execution plans, which *algebraically* describe a complete training iteration. It does not target bugs that cause the training frameworks to crash before generating an execution plan, as such bugs can be handled by traditional tools. It focuses on verifying the parallel execution logic and eliminating correctness bugs essential to parallelization, such as incorrect transformation, wrong synchronization, and incorrect rank assignment. It does not aim to verify the runtimes for carrying out an execution plan, such as the kernel or communication libraries, or the operators' implementation details beyond their semantics (*e.g.*, internal buffer operations in optimizers and collective communication primitives). There are complementary works like dl^2 [40], GPUVerify [29], and GKLEE [50] tackling these tricky issues in communication deadlock, memory check, and GPU kernel verification.

5 System Design

In this section, we present the design of the TrainVerify system, which enables the representation of Symbolic Data Flow Graphs (sDFG) for verification (§5.1). We also introduce two key techniques that enhance scalability: staged verification (§5.2) and shape reduction (§5.3).

5.1 Symbolic Data Flow Graph

As introduced in §2, deep learning models are commonly represented as data flow graphs in many DNN frameworks. However, these off-the-shelf graphs are insufficient for fully capturing the semantics of parallelized model training, due to two limitations: (1) they often omit critical computations involved in training, such as backpropagation, optimizer logic and metrics calculation, *e.g.*, gradient norm (gnorm); and (2) many operators in deep learning frameworks lack formal mathematical definitions that are compatible with verification tools such as SMT solvers. Therefore, TrainVerify extends conventional graphs into *symbolic data flow graphs* (sDFGs).

Graph Completion. In PyTorch, models are usually defined with only the forward pass. PyTorch tracing therefore yields a DFG of the forward path alone, omitting other operations from the full training iteration that are crucial to verify overall parallelization correctness. Thus, TrainVerify reconstructs the backward pass by following PyTorch built-in automatic differentiation functionality [7, 62]. TrainVerify also reproduces optimizer and metric calculations in the graph, adhering to the implementation in distributed training systems [22, 52]. After completion, a DFG will include all the computations in a training iteration.

Graph Symbolization. TrainVerify symbolizes the completed DFG into sDFG by replacing nodes and edges with algebraically defined symbolic tensors and operators. *Symbolic tensors* share the same shape with the original tensors in DFG, but with symbolic `Real` as elements without numeric instability, rather than typical FP32 or BF16 data types. *Symbolic operators* represent the same arithmetic operations over tensors as their DNN counterparts, including common PyTorch operators such as `MatMul`, `AllReduce`, etc. These operators are rewritten in TrainVerify to be compatible with symbolic operands. For example, the symbolic `MatMul` is defined as: $C_{i,j} = \sum_k A_{i,k} \cdot B_{k,j}$ where A, B, and C are symbolic

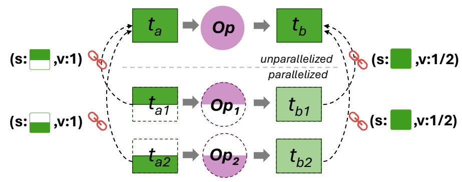


Figure 4. Lineage of tensors, in format `(s,v)`. `s`: index mapping from a subtensor to its parent tensor; `v`: value aggregation required to recover the parent tensor.

tensors, and the computation is performed element-wise using symbolic expressions. This formulation allows the output tensor to be represented as an algebraic expression over the input tensors through symbolic operations. By recursively substituting input tensors with their corresponding symbolic expressions, one can compose consecutive operators, ultimately constructing an algebraic expression representing the entire model.

Lineage. *Lineage* is a data structure that preserves the semantics of parallelization, indicating how a tensor in the original logical model is partitioned or replicated during parallelization, analogous to DTensor [70] from PyTorch and many other similar concepts [52, 81, 83]. TrainVerify also includes tensor lineage between the graph for the logical model definition and the corresponding distributed graph as tensor properties. As illustrated in Figure 4, operator `Op` is partitioned into sub-operators $\texttt{Op}_1$ and $\texttt{Op}_2$, with the input tensor t_a spatially partitioned into sub-tensors t_{a1} and t_{a2}. The lineage implies that $t_a == \texttt{concat}([t_{a1}, t_{a2}], \texttt{dim} = 0)$. Similarly, the output tensor t_b is partitioned into t_{b1} and t_{b2}, where the lineage maintains that t_{b1} and t_{b2} share the same shape as t_b but contain partial values, implying $t_{b1} + t_{b2} = t_b$.

Not all tensors have lineage, such as the tensors created during parallelization which lack counterpart tensors in the graph for the logical model definition.

Lineage serves as a crucial bridge between the sDFG for logical model definition and the distributed sDFG during equivalence checking. Therefore, all input and output tensors must carry lineage information to enable end-to-end equivalence checking. The lineage of intermediate tensors is used in stage-wise parallel equivalence verification.

5.2 Staged Verification

Problem. Modern LLMs are often built with extremely deep architectures. Verifying equivalence of such models experience exponential complexity growth due to their combinatorial nature. As model depth increases, symbolic expressions become intractable, often causing solvers to return `unknown`, or fail to terminate due to memory overflow, even for comparing a single element access in tensors. This highlights the need for an effective approach—decomposing the end-to-end verification into smaller, tractable sub-problems.

Insight. Inspired by pipeline parallelism, the equivalence verification of the entire model can be decomposed into finer-grained stages, with each stage verified individually. If, for every stage in the logical (non-parallel) model, we can verify that it is equivalent to its corresponding stage in the parallelized model, then it naturally follows that the entire model, formed by composing these stages sequentially, is also equivalent. Given that the available *lineage* specifies the mapping of tensors between the logical graph and the partitioned graph, both G_l and G_p can be partitioned into stages accordingly. Formally, if two corresponding stages f and f' are equivalent for all inputs, and similarly for g and g', then their composition must also be equivalent:

$$\forall x_1,\ f(x_1) = f'(x_1) \wedge \forall x_2,\ g(x_2) = g'(x_2) \\ \Rightarrow\ \forall x,\ f(g(x)) = f'(g'(x))$$

Design. TrainVerify introduces *staged verification*. Logical graph G_l and multi-device graph G_p are aligned, then partitioned into *stages*. Each stage S consists of two corresponding subgraphs, $S.G_l$ and $S.G_p$, drawn from the two respective graphs. If all stages pass the verification, the process guarantees end-to-end equivalence between the two graphs.

TrainVerify obtains *tensor alignment* via lineage analysis, identifying bundles that each contains an original tensor in G_l along with its counterparts in G_p. In determining this alignment, TrainVerify traverses G_l in topological order while simultaneously traversing G_p.

TrainVerify then *determines stages* by repeatedly applying backward slicing [78] on the dual graphs. Tensor alignment aids in aligning operators and guides the partitioning of the entire graph into stages. Specifically, during a topological traversal of G_l, whenever an unvisited tensor with lineage information is encountered, it is designated as the output tensor of a new stage S. A backward trace is then initiated from this tensor to identify the corresponding input tensors of the stage in G_l. This trace follows the reverse direction of graph edges and terminates upon reaching tensors that also carry lineage, thereby defining the boundaries of a subgraph in G_l. Using the lineage of each boundary tensor, the corresponding boundaries in G_p are also identified. The stage S is then constructed by extracting the associated subgraphs from both G_l and G_p. Subsequent stages are determined by continuing the traversal over the remaining unvisited portions of G_l and repeating this process. The procedure eventually terminates at the model's output tensors, yielding a complete partitioning of all stages for verification.

Parallel Solving. Algorithm 1 shows an overview of stage-parallel verification. The global relation pool R manages verified relationships among tensors (line 1). The input graphs are partitioned into stages following the aforementioned process (line 2). For each stage, the desired equivalence of both input and output is pre-encoded by lineage, then workers can be invoked sequentially and run in parallel (line 4). After a

Algorithm 1 Stage Parallel Verification

Input: Logical graph G_l, multi-device graph G_p, lineage L
Output: A Boolean indicating whether G_l is equivalent to G_p.

1: $R \leftarrow$ equivalence of G_l and G_p inputs ▷ global relation pool
2: `stages` $\leftarrow$ `align_and_partition`(G_l, G_p, L)
3: **for** stage S in `stages` **do**
4: **async run** Worker(S, R.`snapshot()`)
5: R.`add`($S.L_{out}$)
6: **return** `all`([S.`worker.result()`==**True** for S in `stages`])
7:
8: **function** Worker($\mathcal{S}$, $\mathcal{R}$)
9: **if** ($S.L_{in}$ **not** in $\mathcal{R}$) **then**
10: **return False**
11: $X \leftarrow$`infer_rx_shapes`($\mathcal{S}.G_l$, $\mathcal{S}.G_p$, $\mathcal{S}.L_{in} \cup \mathcal{S}.L_{out}$)
12: $\mathcal{S}$.`init_symbolic_variables(shape=`X`)`
13: `outputs` $\leftarrow$ symbolically execute $\mathcal{S}.G_l$ and $\mathcal{S}.G_p$
14: `cond` $\leftarrow S.G_l$.`inputs` $\overset{S.L_{in}}{==}$ $S.G_p$.`inputs`
15: `target` $\leftarrow S.G_l$.`outputs` $\overset{S.L_{out}}{==}$ $S.G_p$.`outputs`
16: **if** (`target` is **not** `AlwaysTrue` **given** `outputs` $\cup$ `cond`) **then**
17: **return False**
18: **return True**

stage worker is issued, its target output equivalence is assumed correct and added to the global relation pool R (line 5). Subsequent workers are invoked without waiting for prior ones to complete. TrainVerify uses a process pool for worker management, with a configurable concurrency limit. All workers are later synchronized at a barrier to ensure successful completion (line 6).

In each stage's worker, input equivalence is first checked against previously verified relations (line 9). If the equivalence condition does not always hold, then algorithm terminates and returns `False`. The reduced shapes of the involved tensors are inferred (line 11), which will be detailed in §5.3. Symbolic variables for inputs are then instantiated based on reduced shapes (line 12). Output equivalence is verified (lines 15 to 17) based on in-stage computation after applying the corresponding operators (line 13) given the verified relation between inputs (line 14). The verification goals, including losses, final gradient updates, and metrics, correspond to ensuring the desired output equivalence at their affiliated stages.

Supporting Customized Equivalence Approximations. Parallelization of certain model architectures may intentionally introduce approximate mathematical equivalence to optimize performance. For instance, in distributed training of ResNet models, many implementations adopt per-device local BatchNorm as an approximation of global BatchNorm, omitting the additional cross-device averaging that is implicitly embodied in the formulation for the logical model definition. Similarly, in distributed training of MoE LLMs such as DeepSeek-V3, routing is often restricted to a subset of nodes rather than spanning the entire cluster in order to reduce communication costs. TrainVerify supports such approximations in staged verification by overriding the approximate computation (line 13) with its strict-equivalence version, as configured by the user. For approximated BatchNorm, the overridden computation restores the omitted cross-device averaging. For MoE routing, it bypasses the logic that restricts routing to a subset of nodes.

5.3 Shape Reduction

Problem. Modern DNN models often contain tens to hundreds of billions of variables. Even with the stage-parallel design, a single worker may still handle millions of variables, making verification computationally infeasible at such scale. The *shape reduction* mechanism reduces complexity while preserving verification fidelity.

Insight. Although DNN operators typically process large high-dimensional tensors, many operations exhibit redundancy. This redundancy arises from the SIMD (Single Instruction, Multiple Data) property inherent to DNN operators, which apply the same computation across different data elements of a tensor. As shown in Figure 5, elements $c_{1,1}$ and $c_{2,2}$ from output tensor are computed with the same algebraic expression but over distinct input sets:

$$c_{1,1} = a_{1,1} \cdot b_{1,1} + a_{1,2} \cdot b_{2,1} + a_{1,3} \cdot b_{3,1}$$
$$c_{2,2} = a_{2,1} \cdot b_{1,2} + a_{2,2} \cdot b_{2,2} + a_{2,3} \cdot b_{3,2}$$

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix} = \begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,1} & c_{3,2} & c_{3,3} \end{bmatrix}$$

Figure 5. DNN operator `MatMul`: different output elements $c_{1,1}$ and $c_{2,2}$ are calculated using the same function but on different inputs.

Therefore, verification may be performed on the same sDFG with only tensor shapes reduced, omitting redundant elements without compromising the verification outcome. We formally prove that verification on a shape-reduced sDFG is equivalent to verifying the original one. A proof sketch is provided in §6.

Minimum Shapes. Shape reduction shrinks each dimension of a tensor while preserving its total number of dimensions, resulting in a smaller volume. The reduction must satisfy *shape alignment* constraints, which enforce consistency across multiple tensors, and *semantic intactness* constraints, which preserve the representative operations from the original scale.

For example, matrix multiplication, expressed in shape notation as `MatMul`$([M, K] \times [K, N]) = [M, N]$, enforces equality relations among the dimensions M, K, and N respectively across multiple tensors as part of shape alignment. It also enforces semantic intactness, requiring $K \geq 2$ to avoid invalidating the `Add` computation along this dimension. Shape alignment may further involve algebraic reasoning. For instance, a `reshape` operation that converts a tensor from shape $[M, N]$ to $[M, P, Q]$ must satisfy the constraint $P \times Q = N$ to ensure that the total number of elements remains unchanged.

Algorithm 2 Inferring Minimum Reduced Shapes

Input: Logical graph G_l, multi-device graph G_p, lineage L
Output: Tensors' reduced shapes `shape_map`

```
tensors ← G_l.inputs + G_p.inputs
constraints ← ∅
for all t ∈ tensors do
    for d ← 1 to t.ndim do
        t.rx_shape[d] ← init_symbolic_int()
        constraints.add(
            1 ≤ t.rx_shape[d] ≤ t.shape[d]) ❶
for all op ∈ G_l ∪ G_p do
    inputs ← op.inputs()
    outputs ← op.outputs()
    constraints.add(
        op.shape_align_cons(inputs, outputs)) ❷
    constraints.add(
        op.semantic_cons(inputs, outputs)) ❸
constraints.add(L.align(G_l.inputs, G_p.inputs)
                + L.align(G_l.outputs, G_p.outputs)) ❹
objective ← ∑_{t∈tensors} ∏(t.rx_shape) ❺
model ← optimizer.minimize(objective)
shape_map ← {t : model.eval(t.rx_shape)
                for t ∈ tensors}
return shape_map
```

Algorithm. The procedure of shape reduction is shown in Algorithm 2. By collecting the minimum feasible shape for each tensor dimension, it formulates the problem as an integer quadratic optimization, and computes per-stage minimum shapes subject to following constraints: ❶ Each dimension's minimum size should be between 1 and its original size; ❷ Operator enforces *shape alignment* constraints over input and output tensors' dimensions; ❸ Operator enforces *semantic intact* constraints over tensor dimensions; ❹ Lineage enforces that each input/output tensor in G_p holds consistent shapes with its counterpart (sub)tensors in G_l. The optimization goal ❺ uses the total tensor volume to approximate solver complexity, while allowing alternative objectives for solver efficiency. The reduced shapes for *all* tensors in G_l and G_p are then determined.

6 Correctness Proof for Shape Reduction

This section presents a proof sketch[I] for the correctness of the shape reduction method—*verified parallelization equivalence for shape-reduced plans extends to their full-size counterparts*.

DNN operators are predominantly SIMD (Single-Instruction Multiple-Data), performing repeated, homogeneous computations (a *kernel function*) over array elements. This SIMD property is central to enabling our shape reduction. We begin by introducing a formal definition of an SIMD function.

[I]The complete proof is in https://arxiv.org/abs/2506.15961

6.1 Formalization

Consider a function $f(\mathbf{x}) \to \mathbf{y}$, where $\mathbf{x} \in R^{d_1^a \times d_2^a \times \cdots \times d_m^a}$ and $\mathbf{y} \in R^{d_1^b \times d_2^b \times \cdots \times d_n^b}$. So, $rank(\mathbf{x}) = m$ and $rank(\mathbf{y}) = n$. If f is an SIMD function, a kernel function θ associated with f takes a subtensor from $\mathbf{x}$ and outputs a scalar value. Formally:

Definition 2 (Kernel function)**.** *A kernel function θ is a function that takes k scalar inputs and produces a single scalar output:*

$$\theta : \mathbb{R}^k \to \mathbb{R}.$$

Next, we define which input subtensor is associated with each output element. Consider the same function $f(\mathbf{x}) \to \mathbf{y}$. A dependency mapping τ associated with f is a function that maps each index $\mathbf{i}$ in the output $\mathbf{y}$ to a list of indices in the input $\mathbf{x}$. Formally:

Definition 3 (Dependency mapping)**.** *A dependency mapping τ is an affine transformation that maps a vector of integers (an index of tensor $\mathbf{y}$) to a list of indices in another tensor (i.e., $\mathbf{x}$):*

$$\tau : idx(\mathbf{y}) \in \mathbb{N}^n \to [idx(\mathbf{x}), \dots] \in \mathbb{N}^{k \times m},$$

where $idx(\cdot)$ is the indexing function of the tensor; n and m are the ranks of $\mathbf{x}$ and $\mathbf{y}$; and k is the number of inputs in θ.

Definition 4 (SIMD function)**.** *A function $f(\mathbf{x}) \to \mathbf{y}$ is an SIMD function if each output element $\mathbf{y}[\mathbf{i}]$ is computed as:*

$$\mathbf{y}[\mathbf{i}] = \theta(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k),$$

where θ is the kernel function of f, and

$$\mathbf{x}_j = \mathbf{x}[\tau(\mathbf{i})[j]], \quad 1 \le j \le k$$

where τ is the dependency mapping of f.

An SIMD function f is characterized by its dependency mapping θ_f and kernel function τ_f. In compact notation, we write its operation as $\mathbf{y}[\mathbf{i}] = \theta_f(\mathbf{x}[\tau_f(\mathbf{i})])$. [II]

Another type of LLM operators are reductional operations, such as `sum`. A reductional function $f : \mathbb{R}^n \to \mathbb{R}$ returns a single output element from processing a reduction operation among all elements in the input tensor, with the reduction operation satisfying the commutative and associative laws.

Definition 5 (Reductional function)**.** *For an input tensor $X \in \mathbb{R}^n$, the reductional function f applies a binary operation $\odot$ to all elements of X such that:*

$$f(X) = x_1 \odot x_2 \odot \cdots \odot x_n,$$

and $\odot$ satisfies commutativity ($a \odot b = b \odot a$) and associativity ($(a \odot b) \odot c = a \odot (b \odot c)$).

[II]For brevity, we assume a single-tensor input for function f throughout this paper; our method trivially extends to multiple input and output tensors.

6.2 Observations

We make two key observations about LLM operators that are computational (excluding operators like `concat` and `slice`) but not reductional.

Observation 1: LLM operators have kernel functions. We observe that all layers in the transformer architecture have their own kernel functions (Definition 2), including Feed Forward layers, Multi-Head Attention layers without masking (masks discussed in Appendix A), Add & Norm layers, ReLU, Softmax, and Residual addition.

Consider matrix multiplication (i.e., MatMul) as an example. Given two matrices $\mathbf{A} \in \mathbb{R}^{m\times p}$ and $\mathbf{B} \in \mathbb{R}^{p\times n}$, the resulting matrix $\mathbf{C} \in \mathbb{R}^{m\times n}$ has elements $c_{i,j}$ (short for $\mathbf{C}[i][j]$) computed by: $c_{i,j} = \sum_{k=1}^{p} a_{i,k} \cdot b_{k,j}$. Therefore, MatMul has a kernel function:

$$\theta(a_{i,1}, \ldots, a_{i,p}, b_{1,j}, \ldots, b_{p,j}) = \sum_{k=1}^{p} a_{i,k} \cdot b_{k,j}$$

Observation 2: LLM operators have dependency mappings that can be expressed as affine transformations. This property is intuitive, as the "striding" of kernel functions across tensors typically occurs at regular, constant intervals [57]. Consequently, when the input to the dependency mapping—corresponding to the output tensor's index—changes, the resulting input indices change linearly. That is, the mapping takes the affine transformations:

$$\tau(\mathbf{i}) = [\mathbf{M}_1 \cdot \mathbf{i} + \mathbf{b}_1, \ \ldots, \ \mathbf{M}_k \cdot \mathbf{i} + \mathbf{b}_k].$$

For example, in the above MatMul case, the dependency mapping τ^A can be written as:

$$\tau^A(\begin{bmatrix} i \\ j \end{bmatrix}) = [\mathbf{M}_{A1} \begin{bmatrix} i \\ j \end{bmatrix} + \mathbf{b}_{A1}, \ldots, \mathbf{M}_{Ap} \begin{bmatrix} i \\ j \end{bmatrix} + \mathbf{b}_{Ap}], \text{where}$$

$$\mathbf{M}_{A1} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \mathbf{b}_{A1} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \ldots, \mathbf{M}_{Ap} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \mathbf{b}_{Ap} = \begin{pmatrix} 0 \\ p \end{pmatrix}$$

Putting it all together. MatMul is an SIMD function (Definition 4) because it has

- a kernel function:
  $$\theta(a_{i,1}, \ldots, a_{i,p}, b_{1,j}, \ldots, b_{p,j}) = \sum_{k=1}^{p} a_{i,k} \cdot b_{k,j};$$
- a dependency mapping for each input tensor:
  $$\tau^A(i,j) = [(i,k) | 1 \le k \le p],$$
  $$\tau^B(i,j) = [(k,j) | 1 \le k \le p],$$
  where τ^A and τ^B are the dependency mappings for input matries $\mathbf{A}$ and $\mathbf{B}$;
- and MatMul can be expressed as:
  $$c_{i,j} = \theta(A[\tau^A(i,j)] \oplus B[\tau^B(i,j)]),$$
  where $\oplus$ represents list concatenation.

6.3 Proof Sketch

We establish the correctness of TrainVerify's shape reduction by proving that equivalence between two data flow graphs (DFGs) on size-reduced tensors implies their equivalence on full-scale tensors. We denote the logical and transformed DFGs (before and after applying parallelization) as functions f and g, respectively.

6.3.1 Prerequisite Relations. Before presenting the main theorem, we begin with two equivalent definitions that serve as the foundation for the proof.

Definition 6 (Mapping permutation equivalence)**.** *For two dependency mappings τ_1 and τ_2, we say they are mapping permutation equivalent, denoted $\tau_1 \cong_P \tau_2$, if there exists a permutation function P, such that*

$$\forall i, \quad \tau_1(i) = P(\tau_2(i))$$

Mapping permutation equivalence captures LLM operators with commutative properties, where permuting the inputs does not affect the output. Similarly, we need to define a corresponding equivalence relation for kernel functions.

Definition 7 (Kernel permutation-set equivalence)**.** *For two kernel functions θ_1 and θ_2, we say they are kernel permutation-set equivalent, denoted $\theta_1 \cong_Q \theta_2$, if there exists a non-empty set Q of permutation functions, such that*

$$\forall P \in Q, \ \forall \mathbf{x}, \quad \theta_1(\mathbf{x}) = \theta_2(P(\mathbf{x}))$$

6.3.2 Premises from SMT Solver. In TrainVerify, we use an SMT solver (Z3) to verify that a shape-reduced model preserves parallelization equivalence. Specifically, if the solver passes, it proves that for all inputs, the logical dataflow graph of the shape-reduced model is equivalent to that of the parallelized version.

This result yields a premise for each stage (§5.2) in TrainVerify of the following form, where $\mathbf{i}$ is an n-dim index.

$$\forall \mathbf{x}, \forall \mathbf{i} \in \mathcal{I}, f(\mathbf{x})[\mathbf{i}] = g(\mathbf{x})[\mathbf{i}],$$

$$\text{where } \mathcal{I} = \{\sum_{j=0}^{n} a_j \mathbf{e}_j \mid a_j \in \{0,1\} \text{ for all } j\}$$

In the equation, $\mathbf{e}_i$ denotes the standard basis vectors in $\mathbb{R}^n$, defined as:

$$(\mathbf{e}_i)_j = \begin{cases} 1 & \text{if } j = i, \\ 0 & \text{otherwise.} \end{cases}$$

Each $\mathbf{e}_i \in \mathbb{N}^n$ is a column vector with a single 1 in the i-th position and 0 elsewhere, except for e_0 which is all 0s; namely,

$$\mathbf{e}_0 = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}_{n\times 1}, \mathbf{e}_1 = \begin{pmatrix} 1 \\ \vdots \\ 0 \end{pmatrix}_{n\times 1}, \ldots, \mathbf{e}_n = \begin{pmatrix} 0 \\ \vdots \\ 1 \end{pmatrix}_{n\times 1}$$

The above premise holds due to Algorithm 2, lines 4 and 12 where TrainVerify enforces that, for any output dimension of each operator—excluding those not involved in computation (e.g., batch dimensions or element-wise operations)—both the logical and parallelized dataflow graphs retain a size of at least two in those dimensions. Meanwhile, the equivalence for arbitrary input $\mathbf{x}$ is established by symbolic execution.

6.3.3 Main Proof.

We now present the main proof of shape reduction correctness. The argument proceeds in three steps:

1. We first prove $\theta_f \cong_Q \theta_g$ given SMT solver's premises.
2. We then prove $\tau_f \cong_P \tau_g$ based on the premises.
3. Finally, we apply $\theta_f \cong_Q \theta_g$ and $\tau_f \cong_P \tau_g$ to establish the shape reduction theorem.

Lemma 1. *For SIMD functions f and g:*

$$\forall \mathbf{x}, f(\mathbf{x})[\mathbf{e}_0] = g(\mathbf{x})[\mathbf{e}_0] \implies \theta_f \cong_Q \theta_g.$$

We prove this lemma by contradiction—if θ_f and θ_g are not kernel permutation-set equivalent, then there must exist some input $\mathbf{x}'$ where $\theta_f(\mathbf{x}') \neq \theta_g(\mathbf{x}')$, which contradicts the premise that for all $\mathbf{x}$, $f(\mathbf{x})[\mathbf{e}_0] = g(\mathbf{x})[\mathbf{e}_0]$.

Lemma 2. *For SIMD functions f and g:*

$$\forall \mathbf{x}, \forall \mathbf{i} \in \mathcal{I}, f(\mathbf{x})[\mathbf{i}] = g(\mathbf{x})[\mathbf{i}], \quad \textit{where } \mathcal{I} = \{\sum_{j=0}^{n} a_j \mathbf{e}_j \mid a_j \in \{0,1\} \textit{ for all } j\} \Rightarrow \tau_f \cong_P \tau_g.$$

By the premise, we can prove for each dimension $j \in [1, k]$, $\exists P_j, \forall \mathbf{i}, \tau_f(\mathbf{i})[j] \cong_{P_j} \tau_g(\mathbf{i})[j]$. Later, we prove that all P_js are the same permutation (i.e., $\tau_f \cong_P \tau_g$).

> Theorems 3 and 4 prove the *essence* of shape reduction: verified parallelization equivalence on shape-reduced models *faithfully extends* to the original models.

Theorem 3. *Given SIMD functions f and g,*

$$\forall \mathbf{x}, \forall \mathbf{i} \in \mathcal{I}, f(\mathbf{x})[\mathbf{i}] = g(\mathbf{x})[\mathbf{i}], \quad \textit{where } \mathcal{I} = \{\sum_{j=0}^{n} a_j \mathbf{e}_j \mid a_j \in \{0,1\} \textit{ for all } j\} \implies \forall \mathbf{x},\ f(\mathbf{x}) = g(\mathbf{x})$$

Proof. Given the premise:

- By Lemma 1, $\theta_f \cong_Q \theta_g$.
- By Lemma 2, $\tau_f \cong_P \tau_g$ and $P \in Q$.
- Finally, we prove

$$\theta_f \cong_Q \theta_g \ \wedge\ \tau_f \cong_P \tau_g \ \wedge\ P \in Q \implies f = g$$

$$\begin{aligned}
\forall \mathbf{x}, \forall \mathbf{i}, f(\mathbf{x})[\mathbf{i}] &= \theta_f(\mathbf{x}(\tau_f(\mathbf{i}))) && \text{[by Definition 4]}\\
&= \theta_g(P(\mathbf{x}[\tau_f(\mathbf{i})])) && [\text{by } \theta_f \cong_Q \theta_g \wedge P \in Q]\\
&= \theta_g(\mathbf{x}[P(\tau_f(\mathbf{i}))]) && \text{[by tensor indexing rules]}\\
&= \theta_g(\mathbf{x}[\tau_g(\mathbf{i})]) && [\text{by } \tau_f \cong_P \tau_g]\\
&= g(\mathbf{x})[\mathbf{i}]
\end{aligned}$$

Because for any input $\mathbf{x}$, $f(\mathbf{x})$ and $g(\mathbf{x})$ produce the same result, therefore $f = g$. □

Theorem 4. *Given reductional functions f and g,*

$$\forall \mathbf{x} \in \mathbb{R}^2, f(\mathbf{x}) = g(\mathbf{x}) \implies \forall \mathbf{x} \in \mathbb{R}^m, n \geq 2, f(\mathbf{x}) = g(\mathbf{x}).$$

We prove the theorem by mathematical induction.

7 Implementation

We implement TrainVerify in Python with 6,000 lines of code, using the Z3 solver [35] for equivalence checking.

Algebraic Expression of Models. TrainVerify leverages nnScaler, a state-of-the-art distributed training framework [52] that adopts graph-based parallelization. It traces model code written for single-GPU training to obtain an intermediate representation, then compiles it into a parallelized graph (IRGraph), and finally emits the distributed training code.

Building full-fledged execution plans. TrainVerify naturally derives the multi-device plan from the IRGraph, without tracing the distributed code. The IRGraph encodes both forward and backward passes, with nodes representing PyTorch operators [63], communication primitives, and custom operators. As the distributed `gnorm` computation resides in static code outside the graph, TrainVerify completes the execution plan by injecting semantically equivalent logic as custom operators. The current version of TrainVerify supports only ZeRO Stage 1 [65], and thus abstracts the optimizer as a local gradient update operation. The logical model is obtained by invoking nnScaler to emit a single-GPU execution plan. This process faithfully captures operators during tracing, so the execution plan aligns with its model definition in code. Parallelization equivalence is then verified between the single-device and multi-device execution plans.

Tracking tensor lineage between dual graphs. TrainVerify infers tensor lineage from nnScaler's metadata, where the index mapping specifies how a sub-tensor is sliced from the full tensor, and the value mapping indicates whether aggregation with sibling tensors is required for reconstruction. However, this metadata has two limitations. First, corresponding tensors across the dual graphs may have mismatched IDs. We resolve this by using the source code locations to match operators and align their associated tensors. Second, both index and value mappings are scoped to a single micro-batch within one data pipeline, which leads to inconsistent parallelization semantics between the graphs. To resolve this, TrainVerify shifts index mappings based on DP rank and micro-batch ID, to reflect the global batch context; and uses SSA-based tensor IDs along with static analysis of communication nodes to identify value-wise aggregation across data pipelines.

Adapting operators for shape-reduced symbolic tensors. Operators in the IRGraphs are not directly applicable to symbolic variables. TrainVerify implements shape-reduction rules and symbolic adaptations for the operators used in GPT [31], Llama3, and DeepSeek-V3 models (Table 4 in Appendix A.1 lists them). TrainVerify rewrites both forward and backward computations. The rewritten operation logic strictly adheres to the original mathematical definitions.

Extension for new model architectures. Adaptation is only needed if new operators are introduced. Shape reduction rules are specified using simple operators like `equal`, `larger`,

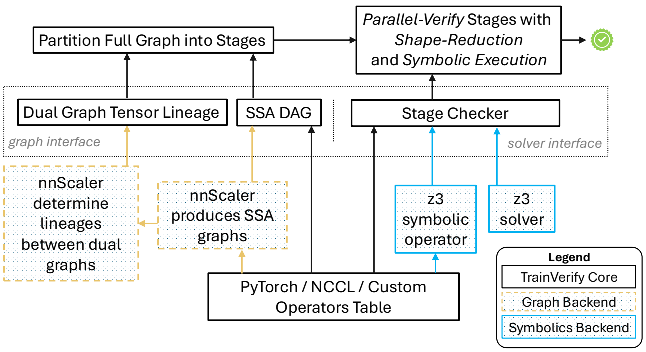


Figure 6. Implementation Overview.

`smaller`, and `divisible`, making them straightforward to specify and maintain. The operator rewriting effort is also moderate, primarily involving the use of NumPy [20] and Python's built-in operators to emulate PyTorch and communication operations. This convenience is enabled by Z3 variables supporting Python operator overloading, allowing symbolic expressions to behave like native types (e.g., `int`, `float`). In our experience, it took 32 development hours to rewrite 40 operators, with 500 lines of code.

Appendix A.2 uses modularized `feedforward` to showcase efforts for operator adaptation. Appendix A.3 describes how we address other practical challenges, without compromising correctness, in shape reduction and adapting non-SIMD operators (*e.g.*, the `embedding` layer and `topK` operator).

Adaptation to Other Frameworks. TrainVerify defines a *graph interface* and a *solver interface* to decouple from both nnScaler and Z3, as shown in Figure 6. To enable verification, a parallelization framework must provide an SSA computation graph enriched with tensor lineage. This may require framework-specific knowledge when such representations are not natively supported. The modular design of stage checker also facilitates potential switching between symbolic engines (*e.g.*, SymPy [59]) to support operator rewriting and solver-specific optimizations.

Accelerating Solving. To enable practical verification time and memory usage for extremely large execution plans, we design several solver-specific optimizations.

Efficient Equivalence Checking. We design a hybrid equivalence checker. It first applies a superficial but efficient expression comparison: if the expressions are equal under simplification, the checker short-circuits without invoking the solver; otherwise, it falls back to the SMT solver. This design exploits the fact that expression comparison is sound and significantly faster than formal solving, though it may yield false alarms due to its superficial nature. To preserve completeness, any case that fails the expression checker is validated by the SMT solver. TrainVerify implements the expression checker using `z3.Tactic("solve-eqs")` [24], which simplifies equations via variable elimination and often reduces the problem to an empty conflict core when expressions are equivalent.

Exp. ID	Model	Layers	DP	TP	PP	NM
L1	Llama3-8B	32	512	1	1	1
L2	Llama3-70B	80	16	8	4	32
L3	Llama3-405B	126	64	8	16	16
D1	DS-V3-16B	27	16	4	2	16
D2	DS-V3-236B	60	16	8	4	16
D3	DS-V3-671B	61	32	8	8	16

Table 2. Evaluated real-world large models.

Lazy Symbolization. TrainVerify instantiates Z3 symbolic variables only within parallel stage solvers, while maintaining metadata in the main process. This design addresses two issues we observed. First, symbolic variables and expressions are costly under Python `fork`, even with copy-on-write; they can add several seconds to each `fork` operation. Second, Z3 requires variables and solvers to be initialized within context tables, which have limited capacity (`unsigned_int`), making it impractical to store all symbolic tensors globally. Managing metadata alone also significantly reduces memory overhead.

8 Evaluation

We answer the following questions: (1) Can TrainVerify verify large parallelized graphs of real-world distributed training settings? (§8.1); (2) How does TrainVerify scale with training parameters? (§8.2); (3) What classes of bugs can TrainVerify eliminate? (§8.3)

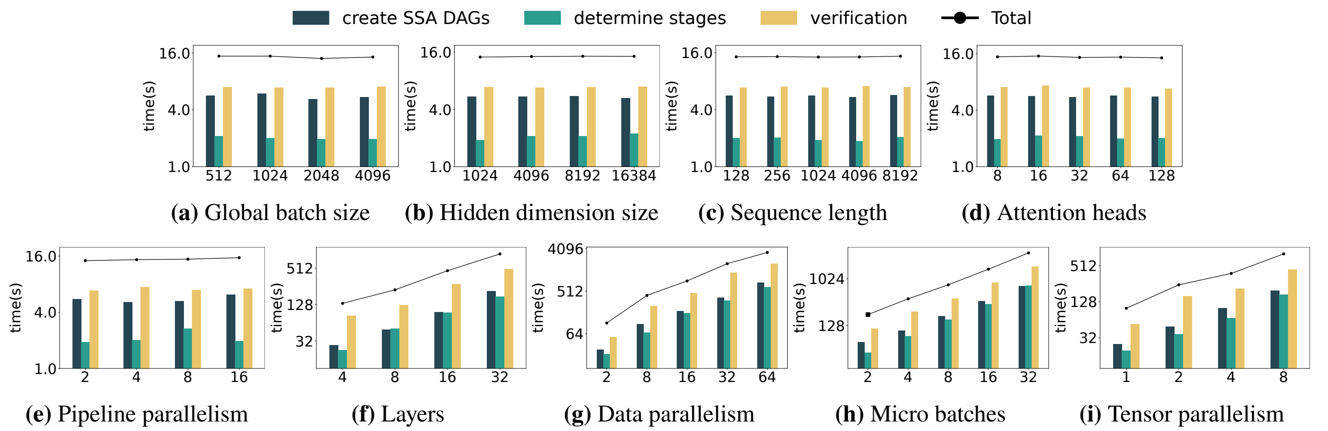
8.1 Verifying Real-World LLM Parallelization

To demonstrate TrainVerify's practicality, we experiment on verifying execution plans for Llama3 and DeepSeek-V3 models under various real-world setups.

We generate the evaluated models' DFGs using nnScaler on a machine with 4 NVIDIA A6000 GPUs. The verification runs on an Azure instance equipped with a 32-core Intel Xeon(R) Platinum 8473C CPU and 1.34 TB memory.

Table 2 shows an overview of the parallelization. Appendix B lists the detailed model specifications. We evaluate the Llama3 models at 8B, 70B, and 405B scales, with plans configured for up to 8192 GPUs, which follows production setup [54, 74]. We evaluate the DeepSeek-V3 models at 16B, 236B, and 671B scales, with plans configured for up to 2048 GPUs. Expert parallelism is treated as a form of tensor parallelism in nnScaler, thus sharing the same degree as TP. As nnScaler currently enforces orthogonality between parallelism strategies, experiment D3 employs the same scale, but uses a different parallelization to approximate the official setup [36].

Table 3 shows the results. For moderate-scale plans, such as L1 and D1, TrainVerify completes verification within half an hour. For large-scale plans, such as L3 and D3, verification takes up to half a day. The increased cost arises from both the larger execution plans and the reduced solver parallelism due to memory constraints. While this cost is non-trivial, it remains acceptable given the correctness guarantees provided,



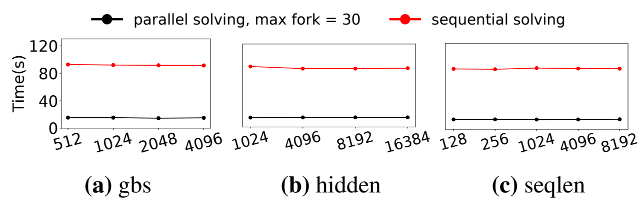
(a) Global batch size (b) Hidden dimension size (c) Sequence length (d) Attention heads

(e) Pipeline parallelism (f) Layers (g) Data parallelism (h) Micro batches (i) Tensor parallelism

Figure 7. TrainVerify's performance trends regarding different training configurations. The y-axes use a $\log_2$ scale. Bars indicate the time breakdown by component, while lines represent the end-to-end verification time.

	L1	L2	L3	D1	D2	D3
Solver Parallelism	30	30	30	30	30	10
End-to-end Time	0.2h	2.4h	8.0h	0.4h	2.4h	9.0h

Table 3. Verification time for the evaluated models.



(a) gbs (b) hidden (c) seqlen

Figure 8. Verification time with vs. without stage parallelism.

especially when compared to the extended training durations, often several weeks, required for models at this scale.

8.2 Scalability

We evaluate how TrainVerify verification time scales under various parallelization and model configurations. The base model used for variable-controlled experiments is Llama3-8B. Each experimental group is averaged over 5 independent runs. Detailed configurations are provided in Appendix B.

Invariance to Original Tensor Shapes. As shown in Figures 7a to 7d, the verification time of TrainVerify, except for variance introduced by shape reduction solving, is independent of the actual tensor sizes. This invariance is achieved through the shape reduction technique, which effectively compresses the shapes of symbolic tensors in TrainVerify's graphs. The compressed shapes depend solely on partitions rather than their original dimensions, so the verification time remains constant regardless of variations in these parameters. For attention heads, increasing the number logically introduces more partitions. However, in efficient implementations, each GPU stacks its local heads and computes them with batched `matmul`. Thus, a larger number of attention heads is treated as reducible and does not impact the verification cost.

Linear Complexity to Parallelization. Stage-parallel verification time consists of two main components: (1) the *main process time* for iterating over stages and inspecting tensor shard metadata to prepare inputs for the solver; (2) the *parallel solver time* for initializing symbolic variables, computing, and checking equivalence of each stage in the workers.

When resources are not constrained, the solvers are sufficiently parallelized, making the main process time dominate the verification time. Pipeline parallelism introduces a sub-linear increase in verification time, as Figure 7e shows, since additional stages contribute only a small number of communication operators. Increasing model layers adds operators sequentially, leading to a linear increase in the number of stages and time, as shown in Figure 7f. For data parallelism, micro-batching, and tensor parallelism, the verification time grows linearly due to increased tensor copies proportional to the parallelism degree, along with the introduction of a small number of communication nodes, as Figures 7g to 7i show.

As the degree of parallelism increases, parallel solver time gradually dominates the verification time. This is because higher parallelism can cause the number of symbolic variables to grow quadratically, leading to quadratic or even greater complexity in the solver. For instance, in experiment D3 (DS-V3–671B in §8.1), the average main process time is much less than 1s per stage, whereas the average solver time reaches 69s per stage, causing the main process to stall while waiting for worker completion. Similar trends are observed in other large experiments. While the dominance of solver time introduces deviations from linear scaling, the overall verification time remains upper-bounded by the statistics reported in §8.1.

Stage-Parallel Verification. We demonstrate the benefit of stage-parallel design by comparing it against a sequential configuration where solver parallelism is disabled. Figure 8 shows the results. These experiments use a relatively small-scale setup (Llama3-8B with 8 GPUs) where individual solvers are already lightweight. Disabling solver parallelism increases the end-to-end verification time from under 18 seconds to over 90 seconds, slowing down by 5 times. The ratio will increase under larger setups as the solver time dominates.

8.3 Eliminating Broad Categories of Bugs

To understand whether TrainVerify addresses correctness issues that arise in real training systems, we consider broad categories of bugs that have been found in popular parallelization frameworks [52, 66, 67] and discuss whether enforcing parallelization equivalence eliminates them.

1. **Incorrect communication operators.** Such as missing `AllReduce` operations required for gradient synchronization, or activating unnecessary communication primitives.
2. **Incorrect device assignment.** Such as miscalculating the participating GPUs when initializing communication groups; or confusing the per-data-parallel mesh with the global device mesh.
3. **Incorrect partitioning of computational operators.** Such as splitting a non-partitionable dimension of a norm operator without proper aggregation or intentional approximation; or applying misaligned masking and slicing when extracting partitioned tensor shards.
4. **Incorrect scaling of distributed states.** Such as incorrect loss scaling and gradient norm calculation due to miscounting tensor replicas under interleaved parallelism, *e.g.*, expert and context parallelism.
5. **Incorrect pipeline scheduling.** Such as gradient synchronization occurring before the final backward iteration of a batch; or micro-batches being mistakenly shuffled.
6. **Incorrect local buffer updates.** Such as updating gradient buffers only for a subset of parameters in the final step; or only updating BF16 buffers during mixed precision training.
7. **Incorrect buffer management.** Violations of memory layout constraints such as contiguity, alignment, or padding required by collective communication routines.

Bug categories (1–4) can be eliminated by TrainVerify. These communication and computation errors manifest as incorrect arithmetic logic, which is explicitly represented in the parallelized DFG. TrainVerify's graphs also include loss scaling and gradient norm calculations that adhere to the original manual implementation. Thus, the violations will be detected through equivalence checking in the affected stages. In early debugging of graph parallelization in nnScaler, we observed subtle bugs stemming from in-place operators, which led downstream operators to consume outdated tensor values. TrainVerify also helps eliminating such bugs.

For pipeline scheduling (5), TrainVerify ensures that the execution sequence is valid. Mistakenly shuffled operators that distort the intended data flow are eliminated through early analysis of data dependencies. Omitting micro-batches prior to gradient synchronization is detected through the verification of finalized gradients. While TrainVerify guarantees that micro-batch scheduling is functionally correct, it does not

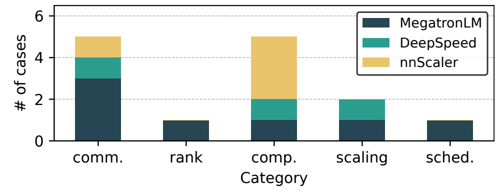


Figure 9. Reproduced incorrect parallelization cases.

enforce adherence to specific scheduling strategies—such as interleaved-1F1B—that aim to optimize pipeline efficiency.

Bugs related to buffers (6–7) are out of scope for TrainVerify as discussed in §4.

Reproduction and Verification. To demonstrate the effectiveness of TrainVerify, we apply it to verify incorrect execution plans. We select 14 non-trivial *real-world* cases from Table 1, covering the broad categories in §8.3, as shown in Figure 9. As these cases span multiple systems, we identify the underlying causes and reproduce them via careful mutation, either in nnScaler's system code or directly in the generated plans, ensuring successful model instantiation. Evaluation is conducted on the Llama3-8B model using 2-way DP, TP, and PP.

TrainVerify detects equivalence violations in all 14 plans, each completing within one minute. Two mutations, one in scheduling and one in communication, are caught early via topological sorting due to broken tensor dependencies. Two other communication mutations are flagged during shape reduction inference for invoking incorrect primitives that cause output shape mismatches, *e.g.* replacing `AllReduce` with `AllGather`. The remaining are detected during verification of their respective stages. Upon detection, TrainVerify generates a counterexample with concrete input/output values and violated lineage for the stage. Appendix B.1 includes a case study for a subtle distributed computation bug in MegatronLM and shows how TrainVerify eliminates such bugs.

Exposing New Violations. In applying TrainVerify, we uncover several classes of buggy plans in nnScaler:

- `C1`: Sharding a non-partitionable dimension.
- `C2`: Violation of the single static assignment convention.
- `C3`: Dangling tensors in backward pass.

They are reported to and confirmed by the nnScaler developers. `C1` errors were discovered in its example code, where the annotation for the modularized operator `apply_rotary_embedding` is inaccurate. Under some strategies, the sequence dimension of the activations is partitioned, but the precomputed positional embeddings are not sliced accordingly, leading to sharded subsequences being mis-encoded as independent starts. `C2` includes tensors that serve as outputs for different nodes, violating the SSA convention. Although in observed cases the output replicas are mathematically equal, this violation exposes a potential vulnerability: downstream transformations, if present, may undesirably interfere with each other. `C3` includes backward tensors with consumers but no producers, which break the data flow. Although they happen to be not reflected in the generated model code due to the autograd accumulation mechanism, this reveals potential errors in the corresponding system implementation.

9 Discussions and Limitations

TrainVerify opts to verify full execution plans without assuming homogeneity across layers or parallelism degrees. This preserves generality, adapting to increasingly flexible parallelization strategies, *e.g.*, differentiated parallelism applied across Transformer layers [54], or across data pipelines [42].

TrainVerify operates at the symbolic level; thus, floating-point precision issues or value drift from accumulation order are out of scope. Certain parallelization bugs, *e.g.*, a missing `AllReduce` for tensor averaging (§8.3), can mimic such numeric symptoms (*e.g.*, FP16 to FP8). TrainVerify eliminates these bugs, avoiding complex debugging where parallelization errors are mistaken for numerical instability.

TrainVerify relies on graph-based execution plans and tensor lineage for scalable verification, so it does not directly apply to code-based parallelization such as MegatronLM [67]. It is possible to support these systems by leveraging tools like PyTorchFX [71] to trace graphs, handling communication operations, and obtaining lineage information through either manual or automated annotation.

TrainVerify is currently built on top of nnScaler [52]. Its interface, however, is system-agnostic: any parallelization system that exposes an SSA graph and lineage can be supported. For example, StableHLO [23], as an MLIR [49] dialect, is inherently SSA-based, making the required graph construction straightforward. Moreover, with minor extensions, programs parallelized via XLA [80]'s auto-sharding can retain the lineage information necessary for TrainVerify's verification.

TrainVerify currently requires manual specification of shape reduction rules and operator rewrites to support symbolic tensors. Automating them, *e.g.*, with `autograd` and program synthesis, is future work.

10 Related Work

Neural network equivalence. A body of work [28, 39, 43, 73] explores neural network equivalence. The closest to TrainVerify is the work by Eleftheriadis et al. [39], which similarly encodes neural networks into SMT clauses, but targets approximate equivalence for knowledge distillation. Their system is restricted to shallow feed-forward networks with at most two layers and thousands of neurons, while TrainVerify scales to state-of-the-art LLMs with billions of parameters.

TASO [43] is a neural network optimization system that accelerates neural networks by substituting certain components with more efficient alternatives. To ensure correctness, TASO includes a verifier that encodes substitution rules as SMT clauses to verify the equivalence of the network before and after substitution. TrainVerify shares a similar philosophy—treating the original network as the ground truth—but differs significantly in scale. Similarly, PET [76] optimizes neural networks via partially equivalent transformations, whose equivalence proof inspires TrainVerify's formalization of shape reduction proof. While TASO and PET handle a small number of operators for each substitution, TrainVerify operates end-to-end, requiring verification of the entire model.

Other works address neural network equivalence for various purposes, including testing DL compiler [56], supporting model rewriting rules [28, 76], repairing models [60], and ensuring the approximation of pruned and distilled models [73]. In contrast, TrainVerify verifies equivalence and focuses on large-scale parallel training, scaling to models with billions of parameters, a level unmatched by prior work.

Neural network verification. In the intersection of deep learning and formal methods, neural network verification [47, 53, 77] addresses a related but orthogonal problem: verifying whether a trained neural network meets specified requirements. These systems encode the network's concrete weights and evaluate whether specific input-output pairs, potentially infinite, satisfy predefined specifications. In contrast, TrainVerify operates on symbolic tensors, verifying equivalence across all possible inputs rather than specific cases.

Correctness of DL training. Prior work enhances DL training via runtime monitoring [25, 33, 44]. TrainCheck [44], for instance, checks invariants inferred from previously validated pipelines to capture end-to-end correctness. Such monitoring provides valuable practical assurance, yet can be constrained by overlooked metrics and numerical drift. Complementary to these efforts, TrainVerify offers symbolic verification with correctness guarantees for parallelization.

11 Conclusion

We present TrainVerify, a system that provides strong correctness guarantees for distributed training by verifying parallelization equivalence. Through multiple techniques such as symbolic representation, staged verification and shape reduction, TrainVerify successfully scales to state-of-the-art LLMs with hundreds of billions of parameters. Our work demonstrates that formal methods can apply to and effectively benefit complex parallel training workflows in practice.

Acknowledgments

We thank our collaborators and colleagues for their contributions, as well as the anonymous SOSP reviewers and our shepherd, Steven Hand, whose feedback improved the clarity and rigor of the paper. This work is supported by the Microsoft Accelerate Foundation Models Research (AFMR) grant program. Cheng Tan is partially supported by NSF CAREER Award #2237295. Yunchi Lu and Peng Huang's participation in the project is supported in part by NSF grants CNS-2317698, CNS-2317751, and CCF-2318937. Youshan Miao is the corresponding author.

References

[1] CUDA. https://docs.nvidia.com/cuda/.
[2] CUTLASS: CUDA Templates for Linear Algebra Subroutines. https://github.com/NVIDIA/cutlass.
[3] DeepSpeed: fix bug where ZeRO2 never uses the reduce method. https://github.com/microsoft/DeepSpeed/commit/a85b6e472534d2e0b61fe234fae4f6a2332c95bf.
[4] DeepSpeed: fix EP grad_scale/grad_norm fix. https://github.com/microsoft/DeepSpeed/commit/e5dd5501c10227ae33dce7d5bdd897741dd3adb7.
[5] DeepSpeed: partition balanced return wrong result. https://github.com/deepspeedai/DeepSpeed/commit/2bdf061f4dc8be70878f032d2e48d2130514f991.
[6] Distributed Data Parallelism. https://pytorch.org/docs/stable/notes/ddp.html.
[7] A gentle introduction to torch.autograd. https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html#computational-graph.
[8] Llama3: Model Details. https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md.
[9] Megatron Core. https://developer.nvidia.com/megatron-core.
[10] Megatron: fix cross entropy loss averaging. https://github.com/NVIDIA/Megatron-LM/commit/adfa873d965b240962be6539cb5d387c508416b9.
[11] Megatron: fix distopt allgathers with interleaved pipeline parallelism. https://github.com/NVIDIA/Megatron-LM/commit/32bbb76d5767fdbf8dc60d4ef07d103cef8aca02.
[12] Megatron: fix *get_data_parallel_src_rank*. https://github.com/NVIDIA/Megatron-LM/commit/9ad1944db1f97000377dc5aee36dcd656b1ae4a2.
[13] Megatron: fix interleaved schedule with sequence-parallel and overlap-p2p-comm. https://github.com/NVIDIA/Megatron-LM/commit/6bd74b0e84577317c06c303f4dae26d249ab55d1#diff-c27f9f2765a43b5c781756ce9fe4b9abcc3618ec96157f5dc67c4a00a7900b73R777.
[14] Megatron: fix no-interleave pipeline schedule. https://github.com/NVIDIA/Megatron-LM/commit/1f387c2cbdb4ce93f0c885862d570efb66dca4a4.
[15] Megatron: fix scaling down expert grads. https://github.com/NVIDIA/Megatron-LM/commit/3373641ff1093073181e219265e8c8ee58d8587c.
[16] Megatron: fix tiling, use correct input size for splits. https://github.com/microsoft/DeepSpeed/commit/c543a41b154a991d50cb6cc8c07dbf46b0d2bdf6.
[17] Megatron issue: Incorrect loss scaling in context parallel code logic. https://github.com/NVIDIA/Megatron-LM/issues/673.
[18] Megatron: LinearWithFrozenWeight backward fix when TP > 1. https://github.com/NVIDIA/Megatron-LM/commit/5fffdfc737f14297bc3781dfc9e273199d1df52e.
[19] Megatron: MoE distributed setup invokes unnecessary all-reduce. https://github.com/deepspeedai/DeepSpeed/issues/6714.
[20] NumPy. https://numpy.org/.
[21] NVIDIA collective communications library. https://developer.nvidia.com/nccl.
[22] PyTorch: Loss Functions. https://pytorch.org/docs/stable/nn.html#loss-functions.
[23] StableHLO. https://github.com/openxla/stablehlo.
[24] Z3 Tactic Online Guide. https://microsoft.github.io/z3guide/docs/strategies/tactics/.
[25] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, page 265–283, Savannah, GA, USA, 2016. USENIX Association.
[26] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.
[27] Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. Gemini: A family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 1, 2023.
[28] Jai Arora, Sirui Lu, Devansh Jain, Tianfan Xu, Farzin Houshmand, Phitchaya Mangpo Phothilimthana, Mohsen Lesani, Praveen Narayanan, Karthik Srinivasa Murthy, Rastislav Bodik, Amit Sabne, and Charith Mendis. TensorRight: Automated verification of tensor graph rewrites. *Proc. ACM Program. Lang.*, 9(POPL), January 2025. https://doi.org/10.1145/3704865.
[29] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. GPUVerify: a verifier for GPU kernels. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, page 113–132, Tucson, Arizona, USA, 2012. Association for Computing Machinery. https://doi.org/10.1145/2384616.2384625.
[30] Zhengda Bian, Qifan Xu, Boxiang Wang, and Yang You. Maximizing parallelism in distributed training for huge neural networks. *arXiv preprint arXiv:2105.14450*, 2021.
[31] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D 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 Ziegler, Jeffrey Wu, Clemens Winter, Chris 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 H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.
[32] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18, page 579–594, Carlsbad, CA, USA, 2018. USENIX Association.
[33] CoreWeave. Weights & Biases. https://wandb.ai/site/.
[34] Ben Cottier, Robi Rahman, Loredana Fattorini, Nestor Maslej, Tamay Besiroglu, and David Owen. The rising costs of training frontier AI models. *arXiv preprint arXiv:2405.21015*, 2024.
[35] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Budapest, Hungary, 2008. Springer-Verlag.
[36] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bochao Wu, et al. DeepSeek-V3 technical report, 2025. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437.
[37] Truong-Dong Do, Minh-Thien Duong, Quoc-Vu Dang, and My-Ha Le. Real-time self-driving car navigation using deep neural network. In *2018 4th International Conference on Green Technology and Sustainable Development (GTSD)*, pages 7–12, 2018.
[38] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[39] Charis Eleftheriadis, Nikolaos Kekatos, Panagiotis Katsaros, and Stavros Tripakis. On neural network equivalence checking using SMT solvers, 2022. arXiv:2203.11629 [cs.AI] https://arxiv.org/abs/2203.11629.

[40] Yanjie Gao, Jiyu Luo, Haoxiang Lin, Hongyu Zhang, Ming Wu, and Mao Yang. dl^2: Detecting communication deadlocks in deep learning jobs. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, FSE Companion '25, page 27–38, Clarion Hotel Trondheim, Trondheim, Norway, 2025. Association for Computing Machinery. https://doi.org/10.1145/3696630.3728529.

[41] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. GPipe: Efficient training of giant neural networks using pipeline parallelism. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32, pages 103–112. Curran Associates, Inc., 2019.

[42] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient distributed training of large models using pipeline templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 382–395, Koblenz, Germany, 2023. Association for Computing Machinery. https://doi.org/10.1145/3600006.3613152.

[43] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 47–62, Huntsville, Ontario, Canada, 2019. Association for Computing Machinery. https://doi.org/10.1145/3341301.3359630.

[44] Yuxuan Jiang, Ziming Zhou, Boyu Xu, Beijie Liu, Runhui Xu, and Peng Huang. Training with Confidence: Catching silent errors in deep learning training with automated proactive checks. In *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '25, Boston, MA, USA, July 2025. USENIX Association.

[45] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with AlphaFold. *Nature*, 596(7873):583–589, 2021.

[46] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[47] Guy Katz, Clark Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 97–117, Cham, 2017. Springer International Publishing.

[48] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. In D. Song, M. Carbin, and T. Chen, editors, *Proceedings of Machine Learning and Systems*, volume 5, pages 341–353. Curan, 2023.

[49] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '21, page 2–14. IEEE Press, 2021. https://doi.org/10.1109/CGO51591.2021.9370308.

[50] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P. Rajan. GKLEE: Concolic verification and test generation for GPUs. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, page 215–224, New Orleans, Louisiana, USA, 2012. Association for Computing Machinery. https://doi.org/10.1145/2145816.2145844.

[51] Shenggui Li, Fuzhao Xue, Yongbin Li, and Yang You. Sequence parallelism: Making 4d parallelism possible. *arXiv preprint arXiv:2105.13120*, 2021.

[52] Zhiqi Lin, Youshan Miao, Quanlu Zhang, Fan Yang, Yi Zhu, Cheng Li, Saeed Maleki, Xu Cao, Ning Shang, Yilei Yang, Weijiang Xu, Mao Yang, Lintao Zhang, and Lidong Zhou. nnScaler: Constraint-guided parallelization plan generation for deep learning training. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 347–363, Santa Clara, CA, July 2024. USENIX Association. https://www.usenix.org/conference/osdi24/presentation/lin-zhiqi.

[53] Changliu Liu, Tomer Arnon, Christopher Lazarus, Clark Barrett, and Mykel J. Kochenderfer. Algorithms for verifying deep neural networks. *arXiv:1903.06758*, 2019.

[54] Guodong Liu, Youshan Miao, Zhiqi Lin, Xiaoxiang Shi, Saeed Maleki, Fan Yang, Yungang Bao, and Sa Wang. Aceso: Efficient parallel DNN training through iterative bottleneck alleviation. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 163–181, Athens, Greece, 2024. Association for Computing Machinery. https://doi.org/10.1145/3627703.3629554.

[55] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context. *arXiv preprint arXiv:2310.01889*, 2023.

[56] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. NNSmith: Generating diverse and valid test cases for deep learning compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 530–543, Vancouver, BC, Canada, 2023. Association for Computing Machinery. https://doi.org/10.1145/3575693.3575707.

[57] Siran Liu, Chengxiang Qi, Ying Cao, Chao Yang, Weifang Hu, Xuanhua Shi, Fan Yang, and Mao Yang. Uncovering nested data parallelism and data reuse in DNN computation with FractalTensor. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 160–177, Austin, TX, USA, 2024. Association for Computing Machinery. https://doi.org/10.1145/3694715.3695961.

[58] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[59] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. SymPy: Symbolic computing in Python. *PeerJ Computer Science*, 3:e103, January 2017. https://doi.org/10.7717/peerj-cs.103.

[60] Satoshi Munakata, Susumu Tokumoto, Koji Yamamoto, and Kazuki Munakata. Towards formal repair and verification of industry-scale deep neural networks. In *Proceedings of the 45th International Conference on Software Engineering: Companion Proceedings*, ICSE '23, page 360–364. IEEE Press, 2023. https://doi.org/10.1109/ICSE-Companion58688.2023.00103.

[61] 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 *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 1–15, Huntsville, Ontario, Canada, 2019. Association for Computing Machinery. https://doi.org/10.1145/3341301.3359646.

[62] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. 2017.

[63] PyTorch Team. PyTorch. https://pytorch.org/.

[64] 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 Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 18332–18346. PMLR, 17–23 Jul 2022. https://proceedings.mlr.press/v162/rajbhandari22a.html.

[65] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 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.

[66] 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 & Data Mining*, KDD '20, page 3505–3506, Virtual Event, CA, USA, 2020. Association for Computing Machinery. https://doi.org/10.1145/3394486.3406703.

[67] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using GPU model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[68] Yi Sun, Ding Liang, Xiaogang Wang, and Xiaoou Tang. DeepID3: Face recognition with very deep neural networks, 2015. arXiv:1502.00873 [cs.CV] https://arxiv.org/abs/1502.00873.

[69] Enyi Tang, Earl Barr, Xuandong Li, and Zhendong Su. Perturbing numerical calculations for statistical analysis of floating-point program (in)stability. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, page 131–142, Trento, Italy, 2010. Association for Computing Machinery. https://doi.org/10.1145/1831708.1831724.

[70] PyTorch Team. PyTorch DistributedTensor (DTensor). https://github.com/pytorch/pytorch/tree/master/torch/distributed/_tensor.

[71] PyTorch Team. TorchFX. https://pytorch.org/docs/stable/fx.html.

[72] PyTorch Team. TorchScript. https://pytorch.org/docs/stable/jit.html.

[73] Samuel Teuber, Philipp Kern, Marvin Janzen, and Bernhard Beckert. Revisiting differential verification: Equivalence verification with confidence. *arXiv preprint arXiv:2410.20207*, 2024.

[74] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[75] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

[76] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 37–54. USENIX Association, July 2021. https://www.usenix.org/conference/osdi21/presentation/wang.

[77] Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and Zico Kolter. Beta-CROWN: Efficient bound propagation with per-neuron split constraints for neural network robustness verification. In *Proceedings of the 35th International Conference on Neural Information Processing Systems*, NIPS '21, Red Hook, NY, USA, 2021. Curran Associates Inc.

[78] Mark Weiser. Program slicing. *IEEE Transactions on software engineering*, (4):352–357, 1984.

[79] Peichen Xie, Yanjie Gao, Yang Wang, and Jilong Xue. Revealing floating-point accumulation orders in software/hardware implementations. In *Proceedings of the 2025 USENIX Annual Technical Conference (USENIX ATC)*. USENIX Association, 2025.

[80] XLA and TensorFlow teams. XLA—TensorFlow, compiled. https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html.

[81] Jinhui Yuan, Xinqi Li, Cheng Cheng, Juncheng Liu, Ran Guo, Shenghang Cai, Chi Yao, Fei Yang, Xiaodong Yi, Chuan Wu, et al. OneFlow: Redesign the distributed deep learning framework from scratch. *arXiv preprint arXiv:2110.15032*, 2021.

[82] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI '20, USA, 2020. USENIX Association.

[83] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, Carlsbad, CA, July 2022. USENIX Association. https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin.