It Takes Two to Entangle

Zhanghan Wang*
New York University
New York, NY, United States
zhanghan.wang@nyu.edu

Ding Ding*
New York University
New York, NY, United States
dding@nyu.edu

Hang Zhu†
ByteDance Seed
Bellevue, WA, United States
hang.zhu@bytedance.com

Haibin Lin
ByteDance Seed
Bellevue, WA, United States
linhaibin.eric@gmail.com

Aurojit Panda†
New York University
New York, NY, United States
apanda@cs.nyu.edu

Abstract

Distributed machine learning training and inference is common today because today's large models require more memory and compute than can be provided by a single GPU. Distributed models are generally produced by programmers who take a sequential model specification and apply several distribution strategies to distribute state and computation across GPUs. Unfortunately, bugs can be introduced in the process, and a distributed model implementation's outputs might differ from the sequential model's outputs. In this paper, we describe an approach to statically identify such bugs by checking *model refinement*, that is, can the sequential model's outputs be reconstructed from the distributed model's outputs? Our approach, implemented in Entangle, uses iterative rewriting to prove model refinement. Our approach can scale to today's large models and deployments: we evaluate it using GPT and Llama-3. Further, it provides actionable outputs that aids in bug localization.

CCS Concepts: • **Software and its engineering → Formal software verification**; • **Computing methodologies → Neural networks**.

Keywords: Formal Verification, Distributed Deep Learning, Equality Saturation, Bug Localization

ACM Reference Format:
Zhanghan Wang, Ding Ding, Hang Zhu, Haibin Lin, and Aurojit Panda. 2026. It Takes Two to Entangle. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 21–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3779212.3790178

1 Introduction

Large machine learning models require more memory than is available on any single GPU. Furthermore, training them or using them for inference requires significant compute capacity, making it infeasible to use a single GPU. Consequently, it is now the norm to deploy these models on multiple GPUs, spread across multiple servers, for training and inference tasks. The approach taken when implementing a distributed ML model has a significant impact on resource efficiency and performance, and thus several distribution strategies [17, 20, 28, 29, 32, 51] are used when implementing ML models. However, implementing distributed models requires programmer effort, and bugs can be introduced during implementation.

To see why bugs can be introduced, we start by looking at a common workflow for creating a distributed model implementation: First, an ML model architect specifies a model architecture as a series of operations. This architecture specification is sequential, i.e., it is written assuming that the operations run on a single GPU (or processor) and operate on local data. Next, an implementer converts the specification into a distributed version by deciding how to partition model state and computation. When doing this, the implementer needs to add communication and transformation operations to preserve the sequential specification semantics. Unfortunately, an implementer might use incorrect parameters (e.g., incorrectly specifying padding or offsets, or using the wrong scaling factor, see §6.2), when adding these additional operations, or worse forget some, resulting in bugs.

Indeed, our work was motivated by the observation that at ByteDance several bugs (discussed in §6.2) had been introduced when implementing a distributed version of a recent model architecture. But our experience was not unique: recent work [14, 20, 50, 52] found similar bugs in open-source distributed ML implementations.

In this paper, our goal is to identify bugs introduced when implementing distributed ML models, before they are deployed. To do so, we propose a static approach for checking *model refinement* (§3.2): that is, can a sequential model G_s's outputs be reconstructed from the outputs of a distributed model implementation G_d? In developing this approach, we

[*] Both authors contributed equally to this research.
[†] Corresponding authors.

ASPLOS '26, Pittsburgh, PA, USA.

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790178

had to address two core challenges: *scalability*: ML models are growing in size and the number of GPUs used by implementations is also growing, and we aim for approaches that can be applied to today's models and implementations; and *usability*: we want to provide the users of our approach with actionable information that can help address bugs.

Our approach, which we have implemented in a tool called Entangle (§3), uses iterative term rewriting to generate a relation (§3.2) to map the outputs produced by the implementation G_d to G_s's outputs. G_d refines G_s if Entangle can find a *complete clean relation*, i.e., a relation that can be used to reconstruct all outputs from G_s without requiring additional computation (beyond what is required to gather and combine outputs from multiple GPUs). The lack of a clean relation indicates a bug. Entangle can work with models and implementations written in PyTorch and other popular frameworks without requiring significant effort, and is thus easily applied to existing distributed ML model implementations.

We address the scalability and usability challenge by adopting an iterative approach where each operator in G_s is processed individually (§4.1). Processing a single operator limits the number of rewritten terms that Entangle has to consider, and ensures that the runtime grows linearly with model complexity. Our evaluation §6.4 shows that this approach allows us to check implementations of state-of-the-art models (e.g., GPT, Qwen2, Llama-3). It takes Entangle between 10—245 seconds (or less than 5 minutes) to check these models. In terms of usability, because Entangle processes a single operator at a time, its output aids programmers in localizing and addressing the bugs it identifies (§6.2).

Furthermore, as we discuss in §3.3, processing individual G_s operators does not affect Entangle's soundness: Entangle will always report bugs if they exist. However, our approach is based on observations about how programmers (and compilers) translate model specifications to distributed implementations, and Entangle cannot ensure completeness if these observations do not hold, i.e., in some cases Entangle might raise a false alarm and report that a correct implementation is buggy. However, we did not encounter false alarms when using Entangle and evaluating it on open-source and proprietary models.

2 Background

2.1 Distribution Strategies

We start by reviewing the strategies used to distribute (or parallelize) machine learning models. The strategies dictate how the sequential model's inputs and computation are partitioned across GPUs, and how outputs from multiple GPUs should be combined to recover the original sequential model's output.

A correct distribution strategy ensures that if inputs are partitioned correctly (the input relation holds) then combining the outputs using the approach provided by the strategy will recover the original result. This observation motivated our formulation of the model refinement problem: we check that there is a mapping from the distributed model's output to the sequential model's output assuming that a user-provided input mapping is correct.

Data Parallelism (DP). was an early distribution strategy to improve training performance. When using this strategy, each GPU (or rank) runs the same model implementation, and independently computes gradients. This strategy requires that training data be partitioned across GPUs (or ranks), and ensures that aggregating gradients from these machines (using all-reduce) produces the same training result as training on a single machine.

Tensor Parallelism (TP) [18, 27], Sequence Parallelism (SP) [17] and Context Parallelism (CP) [8, 21, 38]. partition one or more operators across multiple GPUs. These strategies require that input tensors (TP), sequences (SP) or context (CP) be partitioned across GPUs, and they specify what operations should be used to combine their outputs. They ensure that assuming inputs are partitioned correctly, the combined output is the same (or produces the same results) as the operator(s) running on a single GPU.

Expert Parallelism (EP). [11, 12] is a distribution strategy targeting mixture-of-experts (MoE) models. These models consist of multiple experts, and expert parallelism distributes experts across GPUs. This strategy requires that inputs be routed to the distributed experts using the same routing mechanism as would be used in a sequential implementation, and uses the same operators as the sequential implementation to combine outputs, ensuring that distributed and sequential models have the same output.

Pipeline Parallelism (PP). is a parallelism approach with several variants [7, 10, 13], all of which partition the model's layers across multiple GPUs. PP requires input batches to be partitioned into microbatches, and combines the outputs using gradient accumulation. Similar to TP, it ensures that the accumulated result is the same as would be expected from running the model on a single GPU.

As can be observed, all six distribution strategies provide similar correctness guarantees: if the strategy is correctly applied to a sequential model G_s to produce a distributed implementation G_d, and if the strategy's input relation is used to map sequential inputs to G_d's inputs, then G_d's outputs can be used to produce G_s's outputs. Finally, we note that Entangle makes no assumptions about what distribution strategy is used, and can be used with any of them (or with a combination).

2.2 Example Bugs

Next, we briefly illustrate the types of bugs that can be introduced when using a distribution strategy to implement a sequential model G_s. Later in §6.2 we discuss a larger set of bugs and evaluate Entangle's ability to find them.

Incorrectly scaling auxiliary loss. In MoE training, auxiliary loss [19, 31] is used to better balance load among experts by penalizing hot experts. However, when using tensor parallelism, the auxiliary loss needs to be scaled down by the number of TP ranks T (that is, be divided by T) to balance out a subsequent reduce-scatter operation that sums up all gradients. We observed a bug at ByteDance where an implementation did not scale down the auxiliary loss, leading to the distributed implementation producing an auxiliary loss that was T times larger than expected.

Incompatible configurations for model components. We also observed a bug at ByteDance when switching an MoE model implementation from using TP to shard experts to SP. In this case, the expert weights need to be replicated across SP ranks rather than sharded, but the bug was that MoE weights continued to be sharded rather than replicated when using SP, resulting in incorrect output. To illustrate why, consider a sequential model that computes $X \times A$, where X is an input and A are expert weights. SP requires partitioning X into X_1 and X_2, while sharding partitions A into A_1, A_2. The resulting distributed implementation computes $X_1 \times A_1$ and $X_2 \times A_2$, but these cannot be combined to produce $X \times A$ since the off-diagonal blocks ($X_1 \times A_2$ and $X_2 \times A_1$) were never computed. As we explain in §6.2 this bug did not change the size of the intermediate data, and thus cannot be caught by checking types and shapes in the model implementation.

3 Model Refinement

In this section, we define the model refinement problem that Entangle solves. To do so, we first motivate the model refinement problem by giving an overview of how users can use Entangle (§3.1) to find bugs in distributed model implementations. Next, in (§3.2), we introduce the notation used in the rest of the paper and formally define the problem. Finally, in §3.3, we discuss the guarantees that Entangle provides when checking model refinement.

3.1 Overview

Our goal is to check whether a distributed ML model G_d refines a sequential model G_s that is designed to run on a single machine (i.e., uses compute and memory from a single GPU and processor). Note that in most cases, using G_s for training or inference is impractical because no single machine may have sufficient resources. However, writing a correct G_s is easier because the model designer does not need to consider communication or coordination across GPUs and ranks.

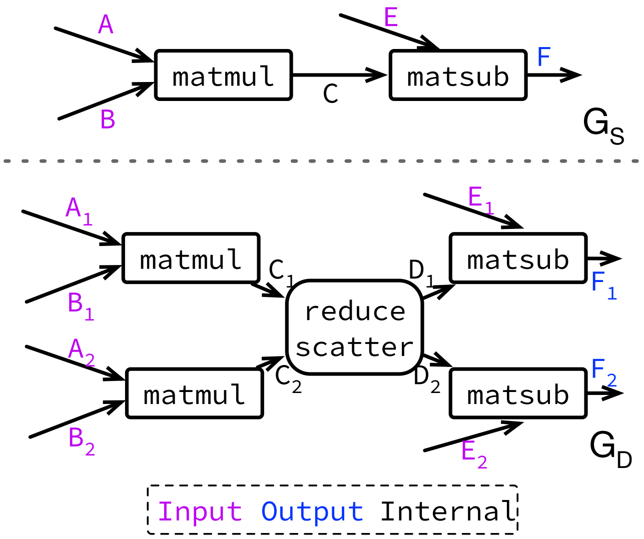


Figure 1: An example of a sequential model G_s and its distributed implementation G_d that is distributed on 2 ranks. G_s produces one output F, while G_d produces two outputs F_1 and F_2. Proving that G_d refines G_s requires finding an expression ρ such that $F = \rho(F_1, F_2)$.

To use Entangle, a user provides G_s and G_d, which are specified as computation graphs (Figure 1). Additionally, the user also provides a *clean input relation* (defined below) R_i that maps G_s's input tensors (A, B and E in Figure 1) to G_d's input tensors ($A_0, A_1, B_0, B_1, E_0, E_1$). From these inputs, Entangle produces a *clean output relation* R_o that maps G_s's outputs (F) to G_d's outputs (F_0, F_1).

From the output relation R_o, the user can determine whether G_d refines G_s by checking whether R_o is *complete*, that is, does R_o contain mappings for all of G_s's outputs. A complete R_o implies that all of G_s's outputs can be derived from G_d's output without significant computation. On the other hand, an *incomplete* R_o means that G_d cannot be used to compute at least one of G_s's outputs.

Finally, the user can use a complete R_o to translate outputs from a deployed G_d to G_s's output.

3.2 Formal Definition and Terminology

Notation	Explanation
$\mathrm{I}(G)$, $\mathrm{O}(G)$	Set of inputs, outputs of graph G.
$\mathrm{I}(v)$, $\mathrm{O}(v)$	Set of inputs, outputs of node v.
$X \longmapsto Y$	An expression maps elements in set X to Y.
$X \xmapsto{clean} Y$	An expression cleanly maps from set X to Y.
R	Relation as a set of tensor-expression pairs.
R_i	Clean relation of inputs.
R_o	Clean relation of outputs.
R_v	Clean relation of outputs of node v.

Table 1: Notations used throughout the paper.

The *model refinement problem* requires computing how the sequential model G_s's outputs can be reconstructed from the outputs of the distributed implementation G_d: given model G_s and G_d and a *clean input relation* (defined below) R_i mapping G_s's inputs to G_d's inputs, solving the model refinement problem requires finding a complete clean output relation R_o that maps all of G_s's output tensors to tensors in G_d. If no R_o can be found, model refinement fails, indicating a bug.

In this paper (and in Entangle), we represent the model G_s and G_d as computation graphs. A computation graph is a directed acyclic graph whose vertices are operators (i.e., computation or communication kernels) and whose edges are tensors. Further, each computation graph G has a set of inputs $I(G)$ and a set of outputs $O(G)$[1]. We use $T(G)$ to refer to all tensors in a computation graph. Our algorithm requires considering the inputs and outputs of each operator, and we use $I(v)$ (or $O(v)$) to refer to v's input (or output). In §5, we discuss how Entangle can extract computation graphs from implementations in popular frameworks including PyTorch.

A relation R from computational graph G to G' is a set of tensor-expression pairs: $R = \{(t, \rho) \mid t \in T(G) \text{ and } t = \rho(T(G'))\}$. An expression is a symbolic description of a computation, and applying expression ρ to an input x evaluates the expression by substituting x for inputs where appropriate. We use the notation $X \longmapsto Y$ to represent an expression mapping elements in the set X to Y.

The *input relation* R_i (provided by users) and *output relation* (required as an output) are relations from G_s to G_d, $R_i = \{(t, \rho) \mid t \in I(G_s) \text{ and } t = \rho(I(G_d))\}$ and $R_o = \{(t, \rho) \mid t \in O(G_s) \text{ and } t = \rho(O(G_d))\}$. Each element in R_i (or R_o) provides a mapping from G_s's inputs (or outputs) to G_d' inputs (or outputs). Note that a relation might provide several mappings for the same tensor t, allowing us to model distributed implementations that replicate inputs.

We define an output relation R_o as *complete* if it contains mappings for all outputs from G_s, that is, R_o is complete iff $\forall o \in O(G_s)\, \exists (o, \rho) \in R_o$.

A *clean expression* ρ is one that consists of two types of operations: (i) operations including slice, concatenate and transpose that rearrange tensor elements, e.g., by permuting them or masking elements in certain positions; (ii) reduction operations including reduce-sum that perform collective communication and combine tensors distributed across nodes. We use the notation $X \overset{clean}{\longmapsto} Y$ to represent a clean expression from set X to Y. A *clean output relation* R_o is a relation that contains only clean expressions, i.e., $\forall (t, \rho) \in R_o,\ \rho$ is clean.

We restrict R_o so that it contains only clean expressions because needing complex computation to reconstruct G_s's outputs from G_d indicates a bug: Programmers apply parallelism strategy (§2) to create an implementation G_d that is equivalent to the sequential model G_s. Combining distributed outputs requires communication and aggregation operations (which clean operations are allowed to perform), but any computation beyond this indicates that G_d is either incomplete or buggy.

3.3 Assumptions and Guarantees

Finally, we list the assumptions made by Entangle when solving the model refinement problem, and discuss the guarantees that it provides.

Assumptions: We make two assumptions about G_s and G_d: First, we assume that the same set of optimizations (e.g., kernel fusion or using optimized kernels such as FlashAttention [6]) is applied to both the specification G_s and the implementation G_d. Second, we assume that if G_d correctly refines G_s, then G_s's outputs can be reconstructed by rearranging or combining G_d's outputs (this assumption is precisely captured by our definition of clean relations). The second assumption is based on our use case: programmers build G_d to implement G_s, and thus additional computation to map G_d's outputs to G_s's indicate a bug.

Guarantees: *Entangle is sound*: that is, if Entangle says that G_d refines G_s, then there exists a clean output relation R_o using which G_s's outputs can be reconstructed from tensors in G_d. This is because Entangle explicitly searches for such an R_o, and returns the relation it finds. Thus, Entangle's output acts as a certificate of soundness.

Entangle is not complete: it can falsely report a bug for a correct G_d. This is because Entangle depends on several assumptions to scale its performance, and it might not find a clean relation if the model or implementation violate these assumptions. Specifically, Entangle assumes that:

1. The same optimizations are applied to G_s and G_d. This might be violated if a programmer or tool optimizes G_d directly, e.g., by replacing multiple operators by a fused kernel.
2. G_d and G_s perform operations in the same order. As we explain in the next section, this assumption allows us to iteratively verify model refinement, and thus scale to large models.
3. If an operator $v_d \in G_d$ refines operator $v_s \in G_s$ (i.e., v_s's outputs can be computed using v_d's outputs), then v_d's inputs can be mapped to v_s's inputs or outputs. This assumption, which we state more formally in §4.3.1, enables an optimization (§4.3.1) that allows Entangle to iteratively consider subgraphs of G_d when searching for R_o, the clean output relation mapping G_d's outputs to G_s's outputs.

These assumptions are motivated by our goals: Entangle aims to find bugs introduced when applying parallelization strategies to create distributed model implementations. Thus, bugs introduced by other optimization strategies, e.g., operator fusion or reordering are out of scope. However, note

[1] For convenience, in our diagrams we represent inputs and outputs as edges that only connect to one vertex.

```
1  def compute_out_rel(G_s, G_d, R_i):
2      sort_vs = topological_sort(G_s)
3      R = R_i
4      for v_s in sort_vs:
5          R_v = compute_node_out_rel(v_s, G_d, R)
6          if not R_v.contains(O(v_s)):
7              raise RefinementError("Could not map
                      outputs for operator", v_s)
8          R = R ∪ R_v
9      R_o = {(t_s, expr(T)) | (t_s, expr(T)) ∈ R, t_s
            ∈ O(G_s), T ⊆ O(G_d)}
10     return R_o
```

Listing 1: Algorithm to compute the relations between output tensors of G_s and G_d inductively. We describe the underlined function in §4.1.

that this does not limit our utility: as we describe in §5, we use the same approach to capture both the G_s and G_d provided as input to Entangle. Thus, our implementation can be used with any compilers or framework that applies the same optimizations to both. Thus far, we have not run into cases where these assumptions were violated.

4 Entangle's Approach

Entangle's approach to computing the clean output relation (Listing 1) R_o is iterative: it processes each operator $v \in G_s$ (we use $v \in G$ to refer to an operation v in computation graph G) in topological order (line 2) and computes a clean output relation R_v (line 5) containing expressions $O(v) \xmapsto{clean} T(G_d)$ that map v's outputs to G_d's tensors. If R_v is not a complete relation, i.e., it does not contain mappings for all of v_s's outputs, then Entangle raises an error indicating that G_d does not refine G_s (line 6). The error includes the identity of the operator v_s whose outputs could not be mapped cleanly to G_d, enabling bug localization.

On the other hand, if R_v is a complete relation, Entangle updates R, a relation containing clean maps $T(G_s) \xmapsto{clean} T(G_d)$ found thus far, by adding the mappings contained in R_v (line 8). Finally, once all operators have been processed, it filters R to produce the clean relation R_o of $O(G_s) \xmapsto{clean} O(G_d)$ (line 9).

The relation R is also provided as an input to the function (Line 5) that computes operator v's output relation R_v. We describe this function in the next section (§4.1), and the algorithm needs to map $I(v)$ $(I(v) \subseteq T(G_s))$ into $T(G_d)$. It uses the relation R for this purpose. Processing operators in topological order ensures that this mapping is always feasible: it ensures that the operator being processed either uses tensors in $I(G_d)$ which can be mapped using the user-provided expression R_i that we use as R's initial value (line 3) or using the output of previously processed operators whose outputs have been mapped by previous calls to `compute_node_out_rel`.

The correctness of our approach requires that any tensor in G_s (whether input, intermediate, or output) can be mapped to one or more tensors in G_d by a clean expression: If this requirement is violated, the algorithm would not find a R_v for some operator $v \in G_s$. Our assumption that optimizations (including kernel fusion) applied to G_d must also be applied to G_s ensures that this requirement holds for our inputs.

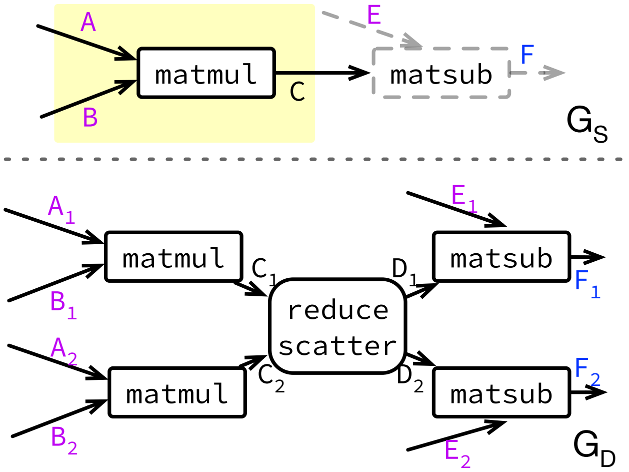


Figure 2: We want to compute the relation between C and all tensors $(A_1, B_1, C_1, D, E_1, F_1, A_2, B_2, C_2, E_2, F_2)$ in G_d. The edges are marked with what output tensor is passed to the next node as the input. All tensors here have two dimensions. Assume we have the input relation R_i: $(A, \alpha_0 : \texttt{concat}(A_1, A_2, \texttt{dim=1}))$, $(B, \beta_0 : \texttt{concat}(B_1, B_2, \texttt{dim=0}))$.

We illustrate this process using the computation graphs in Figure 1: initially the algorithm sets $R = R_i$ and processes the `matmul` operation by finding the relations $R_C = \{(C, \text{ReduceSum}(C_1, C_2)), (C, \texttt{Concat}(D_1, D_2))\}$ (line 5 returns all mappings for C). that maps the intermediate tensor C to tensors in G_d. Next, Entangle updates $R = R_i \bigcup R_C$ and processes the `matsub` operation, and finds the relation $R_F = \{(F, \text{Concat}(F_1, F_2, dim = 0))\}$. Because F is G_s's only output, Entangle emits R_F as its output.

4.1 Computing the Output Relation for an Operation

Next, we detail how Entangle computes the clean output relation R_v for an operator $v \in G_s$ given an input relation R that contains mappings $I(v) \xmapsto{clean} T(G_d)$.

Informally, `compute_node_out_rel` works as follows: (i) it uses the input relation R (provided as input) to produce the expression ρ_v that can be used to compute v's outputs using tensors in $T(G_d)$; (ii) it uses lemmas (§4.2.1) to rewrite and find the set of all expressions (§4.2.2) P_v that are equivalent to ρ_v (including ρ_v); (iii) it uses information from G_d to rewrite and find all equivalent expressions to those appearing in P_v, and adds those to P_v; and (iv) it uses the set of clean expressions in P_v to construct and return the desired output relation R_v.

We illustrate this process by applying it to the `matmul` operator (Figure 2) in our running example. Because `matmul` is the first operator in G_s, `compute_node_out_rel` receives $R = R_i$ as input. The computation proceeds as follows:

```
1  def compute_node_out_rel(v, G_d, R):
2      P_v = set()
3      base_expr = v(I(v))
4
5      # Step 1: replace tensors of I(v_s) with the
           expressions in R.
6      P_v = rewrite_t_to_expr(base_expr, R)
7      exprs_step1 = P_v
8
9      # Step 2: rewrite expressions in exprs_step1
           based on the given lemmas.
10     for expr in exprs_step1:
11         P_v = P_v ∪ rewrite_using_lemma(expr)
12     exprs_step2 = P_v
13
14     # Step 3: rewrite expressions in exprs_step2 by
           replacing the sub-expressions with tensors.
15     T_rel = {$t \in T(G_d)$ |
           $t$ is an input of an expression appearing in R}
16
17     # R_G_d is a relation of all tensors in $T(G_d)$
           that can be computed using tensors in T_rel
           (including by applying multiple operators).
18
19     R_G_d = expressible_using_tensors(G_d, T_rel)
20     for expr in exprs_step2:
21         P_v = P_v ∪ rewrite_expr_to_t(expr, R_G_d)
22
23     # Step 4: filter the expressions and only keep
           clean ones
24     R_v={(t_s,expr)|t_s∈O(v_s),expr∈P_v,expr is clean}
25     return R_v
```

Listing 2: Algorithm to compute the relations between output tensors of v_s and all the input/output tensors of a node set G_d. R is the relations between the input tensors of v and G_d. The underlined functions are described in §4.2.2.

(i) The algorithm uses input mappings for A and B to produce ρ_v : $\texttt{matmul}(\alpha_0, \beta_0)$ (where α_0 : $\texttt{concat}(A_1, A_2, \texttt{dim} = 1)$ and β_0 : $\texttt{concat}(B_1, B_2, \texttt{dim} = 0)$)).

(ii) Next, the algorithm applies the block matrix lemma to rewrite ρ_v and find the equivalent expression ρ_v^1 : $\texttt{sum}(\alpha_1, \beta_1)$ (α_1 : $\texttt{matmul}(A_1, B_1)$ and β_1 : $\texttt{matmul}(A_2, B_2)$)). Another lemma also applies to reduce scatter, but we omit it for clarity.

(iii) The algorithm uses G_d to find additional rewritings for each $\rho \in P_v$. In this example, it uses the observation that $C_1 = \texttt{matmul}(A_1, B_1)$ and $C_2 = \texttt{matmul}(A_2, B_2)$ to rewrite ρ_v^1 to the equivalent expression ρ_v^2 : $\texttt{sum}(C_1, C_2)$. After this step, P_v includes $\{\rho_v, \rho_v^1, \rho_v^2\}$ (along with additional terms from considering reduce-scatter).

(iv) Finally, the algorithm filters P_v to find clean mappings. In this case ρ_v^2 is a clean mapping, and the algorithm returns $R_v = \{(C, \texttt{sum}(C_1, C_2)), (C, \texttt{Concat}(D_1, D_2))\}$, the later of which was computed by considering the reduce-scatter operation.

4.1.1 Computing R_v.

Listing 2 shows the algorithm used to compute R_v for an operator $v \in G_s$ given the relation R of mappings computed by previous calls to this function. As we discussed above, our approach uses iterative expression rewriting to compute R_v. We describe the rewrite functions (which are underlined in the listing) in §4.2.2, and present the overall algorithm below:

First, Entangle computes v's output (line 3) in terms of G_s's tensors, and uses the relation R to express this output in terms of tensors in $T(G_d)$ (lines 5—7), and initializes the set P_v with these expressions. It then applies lemmas to find equivalent expressions, and adds them to P_v (lines 9—12).

Next, it adds to P_v any expressions produced by rewriting the elements in P_v in terms of tensors in $T(G_d)$ (lines 14—21). Observe that the only tensors in G_d that can appear in P_v must also appear in R because applying lemmas cannot produce an expression accessing additional tensors from G_d. On line 15, we compute this set as `T_rel`. Therefore, to improve efficiency, Entangle creates a relation `R_G_d` (line 19) that map tensors in `T_rel` to $T(G_d)$, and then use `R_G_d` to rewrite the relations in P_v (line 21). Finally, it filters P_v to produce its output R_v (line 24).

4.2 Rewriting Expressions and Terms

4.2.1 Lemma.

Expression rewriting is a core part of Entangle's approach. We depend on rewrite rules that we refer to as *lemmas* to identify ways to rewrite an expression. As we discuss in §5 Entangle includes lemmas for common operations in PyTorch's ATen library. Some models rely on optimized kernels or uncommon operators, and we also require users to provide lemmas for this. We evaluate the number of additional lemmas required and the associated effort in §6.5.

An Entangle lemma states under what conditions an expression can be rewritten to another. In our exposition, we represent a lemma as $\rho_m(T_m) \xrightarrow{C_m(T_m)} \rho_n(T_n)$ This lemma states that the expression $\rho_m(T_m)$ and $\rho_n(T_n)$ are equivalent if $C_m(T_m)$ is true. Consequently, if $C_m(T_m)$ holds, then our algorithm will treat $\rho_n(T_n)$ as a valid rewriting of $\rho_m(T_m)$. It is easy to see if under condition C_m ρ_m can be rewritten to ρ_n, there must be some condition C_n under which expression ρ_n can be rewritten as ρ_m. Our algorithm assumes that both conversions are available for each lemma, in practice one can generally be derived from the other.

4.2.2 Rewriting using `EGraphs`.

Given an operator $v \in G_s$, Entangle computes the clean output relation R_v by rewriting expressions using lemmas and mapping in the input relation R. We use `EGraphs` (and the egg [46] library) to implement rewriting. Our use of egg is standard: we represent expressions (ρ above) as `ENodes` and lemmas as rewrite rules; we run saturation, and then use the resulting `EClasses` (containing equivalent relations) in our rewriting functions.

Entangle uses the following three rewriting functions (listing 2), all of which return a set of expressions:

- `rewrite_on_lemma`, to find all expressions that can be produced by using lemmas to rewrite the expression ρ. When processing ρ, this function looks at the

```
14  # Step 3: rewrite expressions in exprs_step2 by
        replacing the sub-expressions with tensors.
15  T_rel = {$t \in T(G_d)$ |
        $t$ is an input of an expression appearing in R}
16  R_explored = set()
17  while true:
18      # Distinct from R_G_d, R_d only contains tensors
            in $T(G_d)$ that can
19      # be computed using a single operator, all of
            whose inputs are in T_rel.
20      R_d = {$(t, \rho : t \longmapsto$ T_rel) |
            $t$ is direct children of T_rel, $\rho \in G_d$}
21      if R_d ⊆ R_explored:
22          break
23      R_d = R_d - R_explored
24      R_explored = R_explored ∪ R_d
25      for expr in exprs_step2:
26          P_v = P_v ∪ rewrite_expr_to_t(expr, R_d)
27      T_rel = T_rel ∪ {t|t is the input of a clean
            expression in P_v}
```

Listing 3: The optimized version of algorithm that should replace the step 3 (line 18-27) in Listing 2.

EClass corresponding to ρ and all its subexpressions, and return all expressions equivalent to ρ.

- rewrite_t_to_expr, which takes as input a relation R and an expression ρ, and rewrites variables in ρ using the expressions in R. If a tensor t is present in ρ and $(t, \rho_t) \in R$, then this function generates a new expression by replacing every occurrence of t in ρ with ρ_t. In our running example, this function is called with the expression $matmul(A, B)$, and a relation containing the tuples $(A, \alpha_0 : \texttt{concat}(A_1, A_2, \texttt{dim} = 1))$ and $(B, \beta_0 : \texttt{concat}(B_1, B_2, \texttt{dim} = 1))$, and produces the expression $\texttt{matmul}(\alpha_0, \beta_0)$.
  Note this function finds and returns all rewriting, so if two tensors t and u occur in ρ, and both $(t, \rho_t) \in R$ and $(u, \rho_u) \in R$ then this function will produce three new expressions from ρ: (i) one where t is replaced by ρ_t; (ii) one where v is replaced by ρ_v; and (iii) finally one where both t and v are replaced. The function will return a set containing all three rewritings and ρ.
- rewrite_expr_to_t, which takes as input a relation R and an expression ρ, and replaces sub-expressions of ρ with tensors appearing in R when possible. If ρ_s is a subexpression of ρ and $(s, \rho_s) \in \rho$ then this function creates a new expression by rewriting all occurrences of ρ_s in ρ with s. In our running example this function is responsible for rewriting the expression $\texttt{sum}(\alpha_1, \beta_1)$ $(\alpha_1 : \texttt{matmul}(A_1, B_1)$ and $\beta_1 : \texttt{matmul}(A_2, B_2)))$ to $\texttt{sum}(C_1, C_2)$ when given a relation R containing the mappings (C_1, α_1) and (C_2, α_2).
  Similar to rewrite_t_to_expr, this function finds and returns all possible rewritings.

4.3 Optimizations

4.3.1 Optimizing Exploration.

The size of R_G_d (line 21) has a significant effect on the time taken to process a single operator: Entangle needs to construct this relation when processing an operator, and it relates to the size of the EGraph used for rewrites. Therefore, we use two observations to further reduce its size.

Observations. Consider operators $v_s \in G_s$ and $v_d \in G_d$. We observe that in most cases, if v_s's outputs can be cleanly mapped to v_d's outputs, then one of the following two conditions holds for all tensors $t \in I(v_d)$: (i) There exists a clean expression that operates on t and maps to a tensor in $I(v_s)$; or (ii) There exists a clean expression that operates on t and maps to a tensor in $O(v_s)$.

The first condition covers the case where all of v_d's inputs can be mapped to inputs of v_s, which indicates that they likely compute related values. The second condition covers the case where a previous operator (v_p) already produces output that can mapped to v_s's output, but additional operators are used to further process (e.g., using reduce-scatter it or padding) this output and produce other equivalent outputs. Our goal is to collect all equivalent outputs, necessitating this condition.

Our core observation is that if v_d has inputs that are not related to v_s then v_d's outputs are unlikely to cleanly map to v_s's outputs. Note that these observations hold because of our assumption that the same optimizations were applied to both G_s and G_d (§3.1). Furthermore, if an input violates this observation, Entangle will lose completeness (i.e., it might falsely report a bug) but soundness will not be affected (i.e., it will not incorrectly report that model refinement holds).

Using these observations. Listing 3 shows how we modify line 14—line 21 in Listing 2 to reduce the size of the G_d subgraph considered.

The optimization maintains the set T_rel of tensors in G_d related to v's inputs or outputs. This set initially contains all tensors $t \in T(G_d)$ that appear in the input relation R (line 15). Because we explore the computational graph in topological order, this initial set contains all tensors $t \in G_d$ such that there is an expression that cleanly maps them to v's inputs.

Entangle then uses an iterative process to find rewritings within the subgraph of G_d that meets the observations:

Entangle uses R_d to rewrite expressions found in step 2 (line 26) and adds them to P_v. Entangle also adds any tensors $t \in G_d$ that appear in newly added clean expressions in P_v, and proceeds to the next iteration.

During this process, Entangle tracks the relations it has considered in each iteration (line 24), and terminates the process when no additional tensors that meet our observations are identified.

To illustrate how the optimized algorithm works in practice, we revisit the example in Figure 2. Initially, T_rel contains A_1, A_2, B_1, and B_2, which are captured in R.

In the first iteration, we consider C_1 and C_2 form G_d, and add $(C_1, \texttt{matmul}(A_1, B_1))$ and $(C_2, \texttt{matmul}(A_2, B_2))$ to R_d. After rewriting, we observe that both C_1 and C_2 appear as inputs to the clean expression $C = \texttt{sum}(C_1, C_2)$, and are thus related to v's outputs and are added to `T_rel`.

In the next iteration, using the updated `T_rel`, which includes C_1 and C_2, we identify the new tensors D_1 and D_2 that satisfy our conditions, and we check again whether the expressions can be rewritten using D_1 and D_2. This second check yields the clean expression $C = \texttt{concat}(D_1, D_2)$. However, because E_1, E_2 are not related to either A, B or C, they are not in `T_rel`. Therefore, they, and tensors computed using them (F_1, F_2) will not be included in `R_d`, and thus not be considered in this case.

4.3.2 Optimizing Term and Expression Rewriting. We also found that naively using EGraphs would produce a large number of rewritten expressions, most of which did not aid in proving model refinement. For example, a lemma of the form $x \rightarrow \texttt{reshape}(\texttt{reshape}(x))$ can be applied to every tensor t, and thus produce a large number of rewritten expressions. However, these expressions are generally not useful: `reshape` is its own inverse, and programmers would not needlessly add extra computation. Thus, we rely on two optimizations to reduce the number of unnecessary rewritten expressions that are produced:

Constrained Lemmas. Some lemmas, e.g., the lemma $X[a : c] \rightarrow \texttt{concat}(X[a : b], X[b : c])$ can produce many rewritten expressions because any integer $a < b < c$ is valid. The same is true for the `reshape` lemma discussed above. However, both lemmas are necessary, these rewrites might be required to prove model refinement, and we cannot remove them. Instead, we add an additional constraint to these lemmas: we require that the target expression or a subexpression (e.g., $\texttt{reshape}(x)$, or both $X[a : b]$ and $X[b : c]$) already appear as `ENodes`, i.e., they already appear as expression in the computational graph.

Pruning Equivalent Expressions. Our second optimization does not limit the set of rewritten expressions, but instead limits the set added to a relation R. In particular, applying lemmas can produce several equivalent expressions, e.g., $\text{concat}(X[16 : 32], X[32 : 48])$ and $X[16 : 48]$. The equivalence of these expressions is dictated entirely by the rewrite lemmas, and does not depend on any mappings provided by the user. We observe that given one such expression, `rewrite_on_lemma` (§4.2.2) can generate all others. Therefore, when maintaining relations (e.g., P_v and R_v) we only add the simplest version of each set of equivalent expressions: we pick the expression with the smallest number of nested expressions. The reduction in the size of the relation reduces memory requirement and improves performance, without any impact on our tools soundness or completeness.

4.4 Checking User Expectations on Refinement

We found that in some cases, Entangle users did not just want to check that a refinement existed between G_s and G_d but also that a particular refinement function sufficed. We support this usage by reducing the problem to the model refinement problem and then using Entangle as normal.

In particular, users specify their expectation by providing functions f_s and f_d that express the desired refinement using tensor expressions, e.g., $f_d \equiv concat(t_1, t_2)$ would indicate that tensors t_1 and t_2 in G_d can be concatenated to produce G_s's output. Given this input, Entangle needs to determine whether $f_s(O(G_s)) \overset{?}{=} f_d(O(G_d))$. It does so using the following process. First, it adds $f_s(O(G_s))$ and $f_d(O(G_d))$ to the input graphs G_s and G_d, producing G'_s and G'_d. It then uses the refinement-checking algorithm to compute the output relation R'_o between $O(G'_s)$ and $O(G'_d)$. Finally, it checks whether R'_o contains the identity relation, i.e., if R'_o shows that $O(G'_s) = O(G'_d)$. If so, user expectations are met; otherwise, we have identified an error. Three of the bugs (bugs 5, 8 and 9) we report on in the evaluation (§6) involve cases where a model did not meet user expectations.

5 Implementation and Usage Experience

We implemented Entangle in 9000 lines of Python, and the relation inference algorithm in about 7800 lines of Rust code. Of the 7800 lines of Rust, 4100 are used to specify lemmas for PyTorch's ATen library [39] and to validate the lemmas (e.g., by checking correct shapes and types). Our implementation includes a combination of new lemmas, and ones ported from TASO [15] and Tensat [49]. The lemmas we implemented de novo were based on input constraints specified in the PyTorch documentation and on mathematical definitions.

Capturing the Computational Graph. . Our implementation relies on existing tools to capture a model's computational graph. We require that the resulting graph be represented as torch.fx style graph representations, and use ATen IR [39] for common operators. As we mentioned previously (§3.3), we capture both G_s and G_d using the same model setup (including program arguments and environment variables) and only varying parallelism size. This ensures that the same set of compiler optimizations are applied to both, and our assumptions (§3.3) hold for the inputs.

For most of our evaluation, we used models written using PyTorch, and used `TorchDynamo` [2] to capture computational graphs. TorchDynamo outputs graphs in the required format.

One of our evaluations used a model implemented using AWS's NeuronX framework that builds on HLO and is compatible with XLA. In this case, we used XLA to generate the computation graph, and then wrote a utility (in 377 lines of Python code) that translated the output to our intermediate format. A similar approach can be adopted when analyzing graphs written in other frameworks including TensorFlow [1] and JAX [5].

```
1  // An example of universal lemmas, in the format:
2  // "<lemma name>" => "<$\rho_m(T_m)$>" => "<$\rho_n(T_n)$>"
3  "<matmul-first-concat-commutative>" =>
4  "(matmul (concat ?A0 ?A1 0) ?B)"
5  => "(concat (matmul ?A0 ?B) (matmul ?A1 ?B) 0)"
6
7  // An example of conditioned lemmas, in the format:
8  // "<lemma name>" => "<$\rho_m(T_m)$>" => |egraph,subst| {
9  //    <customized functions returning $\rho_n(T_n)$> }
10 "<slice-concat-commutative>" =>
11 "(slice (concat ?t1 ?t2 ?dim1) ?dim2 ?begin ?end)"
12 => |egraph, subst| {
13   let [dim1, dim2] = get_vals!(
14       egraph,subst,["?dim1","?dim2"]);
15   if dim1 != dim2 {
16     format!("(concat (slice ?t1 ?dim1 ?begin ?end) (
           slice ?t2 ?dim1 ?begin ?end 1) ?dim2)")
17   } else {
18     // Other branches when dim1 == dim2.
19     ... // omitted
20   }
21 }
```

Listing 4: Lemma Examples (simplified for readability). The expressions are defined using nested tuples, with first element as operator name and rest as parameters.

Writing Lemmas. . Our implementation supports two types of lemmas: *universal lemmas* that can always be used (i.e., lemmas for which condition $C_m(T_m)$ = true), and *conditioned lemmas* that have a non-trivial condition. Both types of lemmas are written using an embedded DSL, but differentiating between the two allows us to reduce developer effort: universal lemmas can be expressed in one or two lines of code, while conditioned lemmas requires more lines. We show examples of both in Listing 4: Lines 3—5 shows the universal lemma stating that `matmul` is commutative. Lines 10—21 show a conditioned lemma about when a combination of the `slice` and `concat` operators are commutative: the rules in this case need to consider the input dimensions, thus necessitating the use of a conditioned lemma.

Handling Symbolic Scalars. . In the computational graphs we capture, tensors do not carry actual data values; instead, they contain only metadata such as shape and data type information. However, certain operators, such as `select`, can extract individual elements from a tensor, and these extracted scalars can be used to compute the tensor shapes. In TorchDynamo, such scalars are represented as symbolic scalars.

To correctly handle these scalars and apply the rewrites, we sometimes need to reason not only about equality, but also about inequality: for example, some lemmas like the commutativity of $\texttt{concat}(X_1, X_2, dim)[a:b]$ can have different rewriting results depending on the equality (or inequality) of the shapes of X_1 and X_2, and the values of a and b. Consequently, we need to be able to compare these even if they are symbolic. However, symbolic scalars cannot be directly compared, so we cannot use `EGraph` for this. Therefore, we encode these scalars using SMT-LIB [4].

Specifically, each scalar in the `EGraph` is associated with metadata that is either a concrete value or a symbolic identifier. Whenever symbolic comparisons are required, we use SMT-LIB to resolve them using user-specified constraints. This allows Entangle to verify end-to-end verification over computational graphs that include symbolic values. In the models we have used Entangle with only simple operations (e.g., addition) are used on symbolic scalars. Consequently, we have found that using SMT solvers to reason about equality or inequality in this cases is feasible, and does not raise concerns about undecidability or performance.

6 Evaluation

Our evaluation focuses on addressing four questions:

- Can Entangle report bugs efficiently in an informative way when they occur (§6.2)?
- How fast can Entangle complete an end-to-end verification (§6.3)? And how well does Entangle scale with respect to the number of parallelism sizes and the layers of models (§6.4)?
- When a user invokes new operators, how much effort is required to complete the operator definition and corresponding lemmas (§6.5)?
- What lemmas are used when checking model refinement for different models (§6.6)?

6.1 Experiment Setup

Hardware Setup. We evaluated Entangle on c6525-25g nodes in CloudLab [9]. Each machine has a 16-core AMD EPYC 7302P CPU running at 3GHz, and 128 GB memory. We ran our experiments on Ubuntu-22.04.

Workload Setup. We evaluate Entangle using the models shown in Table 2. Most of our evaluations only consider the forward pass of the model. The exception is the ByteDance's internal model, where we consider both the forward and the backward pass. This is not because of Entangle's limitation (the approach and our implementation can check both passes) but rather due to limitations in capturing sufficient detail from both passes. In particular, when checking a complete graph, i.e., both the forward and backward pass, a user must provide a single graph that includes both passes or input relations that can be used to map the inputs of one with the other graph. However, limitations in the TorchDynamo make it so that getting such an input requires manual effort: TorchDynamo produces a separate forward and backward pass, and does not relate their inputs. We added input relations or the ByteDance's internal model, but did not do so for the other models in the interest of time. Furthermore, for two of the models, Qwen2 and Llama-3, we only had access to inference scripts, and could not use these to instantiate a version designed for training.

We evaluated four commonly used distribution strategies: TP, SP, EP and gradient accumulation. However, as noted

Framework	Model	Optimization
ByteDance Framework	ByteDance Model	TP, SP, EP
Megatron-LM	GPT[2]	TP, SP
vLLM	Qwen2	TP
Huggingface's transformers	Regression model with MSE[3]	gradient accumulation
Transformers-Neuron	Llama-3	TP

Table 2: A summary of frameworks, models, and optimization strategies.

previously, our approach does not make assumptions about the distribution strategy and can be applied to others.

We did not evaluate DP and PP, both of which are popular, because of limitations of the graph capturing tool. For example, in Megatron-LM, DP is optimized with contiguous buffers, which are initialized before the model runs and are not exposed to TorchDynamo. Similarly, PP relies on intermediate leaf tensors for which it computes gradients, and this is forbidden by TorchDynamo and results in a disconnected graph [40, 41].

6.2 Case Study

One of our goals in designing Entangle was to provide users with actionable information when model refinement cannot be proved. We assess this by reproducing 9 real-world bugs and showing how they aid in localizing the problem. Of the bugs we report on, 5 are from ByteDance and 4 are from open source projects. Of these bugs, one in the ByteDance model was found by our tool, and the others had been previously identified.

As we explained in §4, if Entangle cannot find R_o, it returns the operator $v \in G_s$ where its search terminated. Users can inspect v, its input relations, earlier operators, and any user expectations (§4.4) to understand and identify the source of the problem. We summarize the bugs we found in table 3. Due to space constraints, we present details about each bug in appendix A.

6.3 Verification Time for Different Models

Next, we evaluate time taken by Entangle to compute the output relation and check model refinement. We used Entangle with the models listed in Table 2. The distributed implementation we used had parallelism size set to 2 (i.e., if the model used TP and SP, we would use 2 TP ranks and 2 SP ranks), and we checked a single model layer. Checking a single model layer suffices, since all layers have the same operations. Further, we have empirically found that a parallelism size of 2 suffices for finding most bugs. Finally, as we discussed above, because of limitations when capturing computational graphs, for models other than ByteDance's internal model, we only checked the forward pass.

We report times in Figure 3 for all models other than the HuggingFace's regression model. The Huggingface model was small, and took less than a second. For the remaining models, we observe that verification takes less than 2 minutes for any model (we add the times for ByteDance-Fwd and ByteDance-Bwd because they represent two passes of the same model), demonstrating that Entangle can be used while implementing distributed models. Second, we observe that as expected, the verification times are positively correlated with the number of operators used by the model.

6.4 Scalability

Next, we evaluated Entangle's scalability by measuring verification time as we vary the degree of parallelism and the number of layers (which increases the number of operators). For this evaluation, we used the GPT model and Llama-3. We distributed the GPT model using tensor parallelism (TP), sequence parallelism (SP) and vocabulary parallelism (VP, which is similar to TP) and the Llama-3 model with TP. We used the same degree of parallelism for all types of parallelism.

Figure 4 shows the result in terms of verification time as a function of parallelism size and number of layers. We find that using Entangle remains practical even as the number of operators and parallelism degree increases, showing that it is practical to use our approach in current and emerging deployments. We found that increasing the degree of parallelism has a bigger impact on verification time than increasing the number of layers, but we found that the times remained reasonable up to degree 8. This is because increasing degree of parallelism increases the width of the graph. While verification time is linear in graph depth, the increased width increases the cost of each step, leading to superlinear increase in time.

Further, as we observed above, we have also found that for the distribution strategies we used, increasing the degree of parallelism does not produce additional bugs.

6.5 Adding Operators and Lemmas

Our implementation (§5) already includes lemmas for commonly used operators in PyTorch's ATen library. As we show in §6.6, these operators are commonly used. But models also use optimized kernels that fuse operators or are optimized for particular hardware. Furthermore, other IRs, e.g., HLO, provide operators whose semantics might differ from ATen's operators.

When verifying models that use operators outside the ATen library, Entangle requires users to provide lemmas that capture the operators semantics. In Figure 5 we quantify the number of new lemmas required to verify the models

[2] This is the example GPT training script[36] in the Megatron-LM repository.

[3] This is a test case from Huggingface's transformers repository [35].

Framework	Description
ByteDance Framework	1: Incorrect offset in RoPE with SP
	2: Incorrect scaling for auxiliary loss with TP
	3: Mismatched padding and slicing in data processing
	4: Incompatible configurations for model components
	5: Missing aggregation for a layernorm weight
Huggingface transformers	6: Wrong scaling in gradient accumulation [25, 48]
Megatron-LM	7: Missing all-reduce in parallel linear layer due to mis-configuration [42]
	8: Missing all-reduce in optimizer for MoE router with TP+SP [30]
Transformer-Engine	9: Missing all-reduce in optimizer for layernorm with SP [26]

Table 3: Bugs Summary

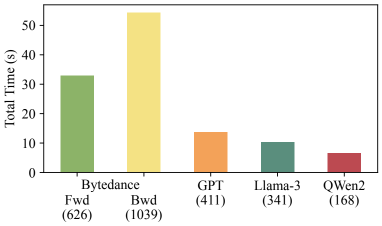
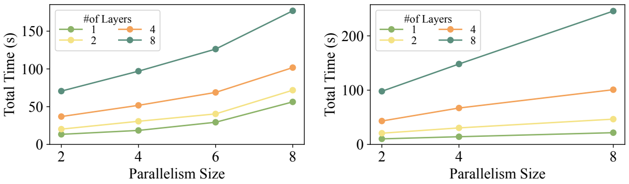
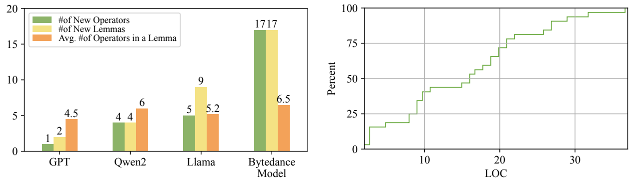


Figure 3: End-to-end verification time across different models. The number in parentheses are the total number of operators in G_s and G_d graphs. The "Fwd" and "Bwd" are the forward and backward graphs of ByteDance proprietary model.



(a) GPT **(b)** Llama-3

Figure 4: Scalability on verifying parallelized models. For Llama-3, there is no data for parallelism size as 6, because some component cannot be evenly partitioned by 6.



(a) #of operators and lemmas and average of #of operators per lemma. **(b)** CDF of LOC per lemma.

Figure 5: The efforts to support customized operators.

we evaluated against (Table 2) and effort required to create these lemmas.

We quantify effort in two ways: lines of code (shown in the CDF Figure 5b), and *lemma complexity*. We measured lemma complexity by counting the number of operators appearing in the lemma. For example, consider the lemma:

$$\texttt{RMSNorm}(\texttt{concat}(X_1, X_2, dim = 0), W) \xrightarrow{Cond(X_1,X_2,W)} \texttt{concat}(\texttt{RMSNorm}(X_1, W), \texttt{RMSNorm}(X_2, W), dim = 0)$$

Two operators appear on the left hand side (RMSNorm and concat) while three appear on the right, and we would assign this lemma a complexity of 5. For each model, we report the average complexity of all lemmas that were added.

These measurements show that when using Entangle, users need to add a small number of lemmas. Lemmas can be written in a few (< 40) lines of code, and most lemmas are simple. In practice, we found that adding lemmas was not a burden, and thus conclude that the need to add lemmas for optimized operators does not impede Entangle's usability.

6.6 Lemma Application Analysis

Finally, we analyzed the frequency with which different lemmas were used when using Entangle to check different models. Figure 6 shows a heatmap of the number of times each lemma is used. We observe the following from the heatmap:

- Lemmas about operators that can appear in clean expressions, including `slice`, and `concat` are the most commonly used.
- While models that use HLO (Llama-3) require some additional lemmas, they allow us to reuse many of the popular lemmas, including `slice`, `reshape` and `concat` that are developed in the context of ATen.
- The different GPT rows show that increasing degree of parallelism increase the number of times that lemmas must be applied. This matches our observation about scalability (§6.4): increasing parallelism has a significant impact on verification time.

7 Related Work

Verification for ML model transformations. Prior work has looked at verifying the equivalence between two ML models. Much of this work has been done in the context of superoptimizing ML compilers: Tensat [49] uses `EGraphs` to prove equivalence; TASO [15] encodes the equivalence problem in first order logic and uses an SMT solver to check equivalence between models after optimization; TensorRight [3] uses rewriting rules to generate bounded proof obligations that can be discharged more efficiently using SMT solvers; and Mirage [47] and PET [43] use a probabilistic approach

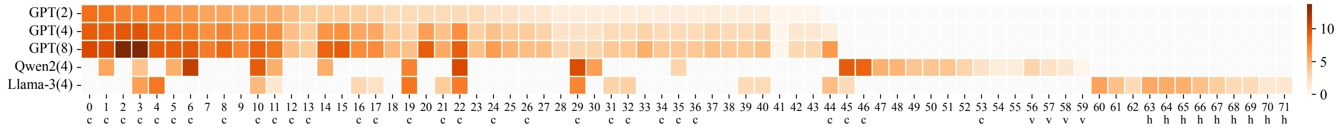


Figure 6: The heatmap shows (in log scale) the number of times each lemma is used for different models, under different parallelism settings. The numbers in parenthesis on the Y-axis represent the degree of parallelism. The x-axis shows lemma IDs, and lemmas marked with *c* concern operators that can appear in clean expressions, those marked with *v* concern operators from vLLM, and those marked with *h* concern HLO operators.

that evaluates the two models on different inputs to check equivalence. The approaches adopted by superoptimizing compilers are generally less scalable than Entangle's approach, and most compilers only consider rewritings of small subgraphs (e.g., consisting of at most a few tens of nodes). The lack of scalability is because of the different setting that they target: Our scalability builds on the assumption that the output of each operator $v \in G_s$ can be mapped to one or more tensors in G_d. But many optimizations, including kernel fusion, that superoptimizing compilers are designed to automate violate this requirement. Consequently, they cannot use Entangle's iterative approach to scale.

Two recent projects, TrainVerify [23] and Aerify [52] have focused on identifying bugs introduced when parallelizing ML models. Source codes (and details about all optimizations) are available for neither, and therefore we *did not* compare to either in our evaluation.

Of these, TrainVerify [23] uses an SMT-based approach to verify element-wise equivalence between output tensors. Operating at the element level introduces scalability concerns, which TrainVerify addresses by using shape-reduction technique to reduce the size of tensors, and by partitioning the input graph. However, to partition the graph, TrainVerify needs to identify equivalent intermediate tensors. For most frameworks, including Megatron-LM and DeepSpeed, these equivalences must be identified manually, and thus using TrainVerify requires additional user effort beyond what is required when using Entangle.

Aerify [52] is more closely related: similar to Entangle, it uses EGraphs to verify semantic equivalence between models. Our work differs in two crucial ways: (a) Aerify's definition of semantic equivalence requires that both models' outputs belong to the same EClass, i.e., they must be equal. This is a stronger condition than what is required by model refinement: we do not require G_d the produce output that is equal to G_s's output (indeed, this is not the case for many of the implementations we found), but rather that one's output can be mapped to the other. (b) Aerify tries to verify both model optimization and distribution, and similar to verifiers for superoptimizing compilers, our iterative approach cannot be used when reasoning about optimization. Finally, Aerify suggests heuristic model partitioning as a way to scale to larger models, but the use of heuristics often impacts soundness. By contrast, Entangle is sound.

Fuzz Testing. Prior work has also proposed using fuzz testing [16, 22, 24, 44, 45] to evaluate model equivalence. Unlike static analysis based approaches, fuzz testing can scale to large models. However, fuzz testing cannot guarantee soundness. By contrast, verification approaches such as Entangle can provide soundness guarantees, albeit at the cost of scalability.

Optimizing ML Compilers. Our approach relies on term and expression rewriting. Expression rewriting is central to most ML compilers, including TASO [15], Mirage [47], TensorRT [37], PET [43], and Mirage [47]. Most of these compilers either develop their own graph substitution algorithms for expression rewriting (TASO, Mirage, etc.), use sketches [33] (TVM, etc.), use expression templates (Taso, TVM, Mirage, etc.), or use a combination. These approaches do not generate *all rewritings* for an expression, and thus cannot be readily used by Entangle. Tensat [49] is a rewrite of Taso using EGraphs, and thus uses a similar expression rewriting strategy as Entangle.

8 Conclusion

Distributing ML model state and computation across multiple GPUs is a necessity today: model sizes continue to increase, as does the amount of data and compute necessary to train them and use them. We started work on this project because we observed that implementing a distributed model often involved many missteps: bugs would be introduced but go unnoticed in the implementation phase, and would only be noticed during training or later. Formal verification has been used to address similar problems in other domains, e.g., cryptography and networking. But ML differs from these domains in scale: most existing ML models are very large. This led us to design Entangle, an approach that uses iterative expression rewriting to check model refinement, and identify bugs introduced when implementing distributed models. The use of iterative expression rewriting allows Entangle to scale to today's models.

Acknowledgments

We thank Jinkun Lin, Jinyang Li and Shivaram Venkatraman for comments on early drafts. We thank the anonymous ASPLOS reviewers and our shepherd for helping us improve this work and how it is presented in this paper. This work was funded in part by the National Science Foundation under award CNS-2145471, and gifts from Google and the Stellar Foundation.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, et al. 2016. TensorFlow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265–283.

[2] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 929–947. doi:10.1145/3620665.3640366

[3] 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. 2025. TensorRight: Automated Verification of Tensor Graph Rewrites. *Proc. ACM Program. Lang.* 9, POPL, Article 29 (Jan. 2025), 32 pages. doi:10.1145/3704865

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org.

[5] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs.* http://github.com/jax-ml/jax

[6] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and memory-efficient exact attention with io-awareness. *NeurIPS* 35 (2022), 16344–16359.

[7] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, et al. 2025. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437

[8] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[9] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. 2019. The design and operation of CloudLab. In *USENIX ATC*.

[10] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. 2021. DAPPLE: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 431–445.

[11] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: scaling to trillion parameter models with simple and efficient sparsity. *J. Mach. Learn. Res.* 23, 1, Article 120 (Jan. 2022), 39 pages.

[12] Jiaao He, Jiezhong Qiu, Aohan Zeng, Zhilin Yang, Jidong Zhai, and Jie Tang. 2021. FastMoE: A Fast Mixture-of-Expert Training System. arXiv:2103.13262 [cs.LG] https://arxiv.org/abs/2103.13262

[13] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems* 32 (2019).

[14] Bytedance Inc. 2024. https://volcengine.github.io/veScaleWeb/blog/mlsys2024.html

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

[16] Haitian Jiang, Shaowei Zhu, Zhen Zhang, Zhenyu Song, Xinwei Fu, Zhen Jia, Yida Wang, and Jinyang Li. 2025. TTrace: Lightweight Error Checking and Diagnosis for Distributed Training. arXiv:2506.09280 [cs.DC] https://arxiv.org/abs/2506.09280

[17] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2022. Reducing Activation Recomputation in Large Transformer Models. arXiv:2205.05198 [cs.LG] https://arxiv.org/abs/2205.05198

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

[19] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. arXiv:2006.16668 [cs.CL] https://arxiv.org/abs/2006.16668

[20] 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. 2024. nnScaler: Constraint-Guided Parallelization Plan Generation for Deep Learning Training. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 347–363.

[21] Hao Liu, Matei Zaharia, and Pieter Abbeel. 2023. Ring Attention with Blockwise Transformers for Near-Infinite Context. arXiv:2310.01889 [cs.CL] https://arxiv.org/abs/2310.01889

[22] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 530–543. doi:10.1145/3575693.3575707

[23] Yunchi Lu, Youshan Miao, Cheng Tan, Peng Huang, Yi Zhu, Xian Zhang, and Fan Yang. 2025. TrainVerify: Equivalence-Based Verification for Distributed LLM Training. arXiv:2506.15961 [cs.DC] https://arxiv.org/abs/2506.15961

[24] Weisi Luo, Dong Chai, Xiaoyue Run, Jiang Wang, Chunrong Fang, and Zhenyu Chen. 2021. Graph-based Fuzz Testing for Deep Learning Inference Engines. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 288–299. doi:10.1109/ICSE43902.2021.00037

[25] Benjamin Marie. 2024. https://github.com/huggingface/trl/issues/2175

[26] Tim Moon and Megatron-LM Team. 2025. https://github.com/NVIDIA/TransformerEngine/pull/1528

[27] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient

large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[28] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. arXiv:2104.04473 [cs.CL] https://arxiv.org/abs/2104.04473

[29] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. arXiv:1910.02054 [cs.LG] https://arxiv.org/abs/1910.02054

[30] RookieHong and Megatron-LM Team. 2023. https://github.com/NVIDIA/Megatron-LM/issues/599

[31] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. arXiv:1701.06538 [cs.LG] https://arxiv.org/abs/1701.06538

[32] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 [cs.CL] https://arxiv.org/abs/1909.08053

[33] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *ASPLOS*. 404–415.

[34] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. 2023. RoFormer: Enhanced Transformer with Rotary Position Embedding. arXiv:2104.09864 [cs.CL] https://arxiv.org/abs/2104.09864

[35] The Huggingface Team. 2025. https://github.com/huggingface/transformers/blob/main/tests/trainer/test_trainer.py

[36] The Megatron-LM Team. 2025. https://github.com/NVIDIA/Megatron-LM/blob/main/examples/run_simple_mcore_train_loop.py

[37] The NVIDIA Team. 2016. https://developer.nvidia.com/tensorrt

[38] The NVIDIA Team. 2024. https://docs.nvidia.com/megatron-core/developer-guide/latest/api-guide/context_parallel.html

[39] The PyTorch Team. 2023. https://pytorch.org/docs/stable/torch.compiler_ir.html

[40] The PyTorch Team. 2023. https://github.com/pytorch/pytorch/issues/109505

[41] The PyTorch Team. 2023. https://github.com/pytorch/pytorch/issues/107861#issuecomment-1696058500

[42] trintamaki and Megatron-LM Team. 2024. https://github.com/NVIDIA/Megatron-LM/commit/5fffdfc737f14297bc3781dfc9e273199d1df52e#diff-855adbcea94c997a151e12312a282117853f541a11989febe40db2ad12fa38c6

[43] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. 2021. PET: Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 37–54. https://www.usenix.org/conference/osdi21/presentation/wang

[44] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep learning library testing via effective model generation. In *ESEC/SIGSOFT FSE*. ACM, 788–799.

[45] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep learning library testing via effective model generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 788–799. doi:10.1145/3368089.3409761

[46] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (Jan. 2021), 29 pages. doi:10.1145/3434304

[47] Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Kit Ao, Praveen Velliengiri, Xupeng Miao, Oded Padon, and Zhihao Jia. 2024. Mirage: A Multi-Level Superoptimizer for Tensor Programs. arXiv:2405.05751 [cs.LG] https://arxiv.org/abs/2405.05751

[48] Zhaofeng Wu. 2021. https://github.com/huggingface/transformers/issues/14638

[49] Yichen Yang, Phitchaya Mangpo Phothilimtha, Yisu Remy Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. arXiv:2101.01332 [cs.AI] https://arxiv.org/abs/2101.01332

[50] Xiao Yu, Haoxuan Chen, Feifei Niu, Xing Hu, Jacky Wai Keung, and Xin Xia. 2025. Towards Understanding Bugs in Distributed Training and Inference Frameworks for Large Language Models. arXiv:2506.10426 [cs.SE] https://arxiv.org/abs/2506.10426

[51] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. arXiv:2201.12023 [cs.LG] https://arxiv.org/abs/2201.12023

[52] Kahfi Soobhan Zulkifli, Wenbo Qian, Shaowei Zhu, Yuan Zhou, Zhen Zhang, and Chang Lou. 2025. Verifying Semantic Equivalence of Large Models with Equality Saturation. In *The 5th Workshop on Machine Learning and Systems (EuroMLSys'25)* (Rotterdam, Netherlands). New York, NY, USA.

A Bug Descriptions

We describe the bugs we evaluated in (§6.2) including their root cause and how Entangle reports them.

A.1 Bugs observed in ByteDance

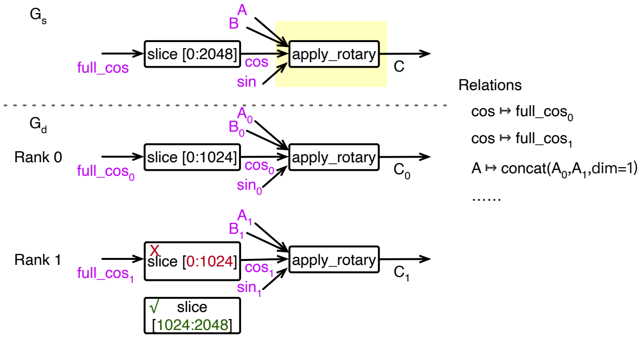


Figure 7: The sub-graphs for the RoPE Bug. Some key relations listed on the right side.

Bug 1: Incorrect offset in RoPE with SP. When sequence parallelism is enabled, the RoPE embedding [34] takes a partition of the original sequence as its input. Consequently, each SP rank should take a different part of the pre-computed cos and sin tensors. When developing this model, a developer had correctly set the corresponding offset in the code used for the forward pass. However, because the backward pass was implemented using `torch.autograd.Function` and the developer forgot to also set the corresponding offset in its backward method (Figure 7), resulting in an implementation bug.

Entangle detect this bug when trying to infer a clean relation for the RoPE operator's output $O(v)$. The user debugs this by checking the input relation $I(v)$ of the operator, which shows that the cos tensor in $\mathrm{I}(v)$ can only be related to the tensor before they are sliced ($\mathrm{full\_cos}_0$ and $\mathrm{full\_cos}_1$ in Figure 7). This is unexpected, because sequence parallelism requires partitioning these tensors and the cos should also map to $\texttt{concat}(\cos_0, \cos_1, dim = 0)$. Tracing back a step further to understand why the mapping between the cos tensor and the sliced tensors is missing, the user can see that the slice offsets are incorrect, thus localizing the problem.

Bug 2: Incorrect scaling for auxiliary loss with TP. We mentioned this bug in §2.2: During MoE training, auxiliary loss [19, 31] is used to penalize hot experts and improve load balancing. When using TP, the loss should be divided by TP size T to balance out a subsequent reduce-scatter operation that sums up the gradient. Otherwise, the final aggregated gradients can be T times the expected one.

In this case, Entangle can map outputs for the auxiliary loss update. However, it fails to find a mapping for a subsequent `matmul` operation that multiples the gradient with another tensor. The bug would require dividing the gradient by TP size to be equivalent, but division is not a clean expression. In this case, the user works backwards from the `matmul` operator to identify the missing division.

Bug 3: Mismatched padding and slicing in data processing. The `AllGather` operation that we use at ByteDance requires that the input tensor from senders have the same shape. Thus, when adopting SP, a developer needs to pad tensors to meet this requirement, and subsequently use the `slice` operator to drop the padding. A bug was introduced when a developer used inconsistent parameters for the padding and slice operators, which resulted in some non-padding elements being dropped and padded elements being retained.

Entangle detects this bug while inferring clean relation for a subsequent `baddbmm` operation. In particular, given the operator v's input relation $\mathrm{I}(v)$, Entangle could not find a clean output relation $\mathrm{O}(v)$. The Entangle user can inspect both the `baddbmm` operation and the previous `slice` operation that produces its inputs to discover that the slice operation had dropped required element. This allows the user to compare parameters for the slice and pad operators, and thus address the problem.

Bug 4: Incompatible configurations for model components. This bug in ByteDance's model was identified by our tool during this evaluation, and we previous discussed it in §2.2. In brief, a developer used SP to parallelize a MoE model, which requires replication the experts' weights. Unfortunately, the developer did not correctly configure some model components, and the expert weights were sharded. This led to a bug: if the sequential model computed $X \times A \times B$, the buggy implementation would compute $X_1 \times A_1 \times B_1$ and $X_2 \times A_2 \times B_2$, where X_1, X_2, A_1, A_2, B_1, B_2 are partitions of X, A and B, respectively. Furthermore, the resulting output still matches the input's hidden dimension size (because the output is still the same size as $X \times A \times B$), and thus the resulting model can be trained. However, the implementation behaves differently from the sequential specification: it never computes the off-diagonal blocks $X_1 \times A_2$ and $X_2 \times A_1$, and they do not contribute to the final output.

Entangle detects this bug when trying to map the first `matmul`'s output (i.e., $X \times A$) because its output cannot be mapped to any tensor in the implementation. Given this information, the user investigates the operator's input relation and find that input A is incorrectly partitioned.

Bug 5: Missing aggregation for a layernorm weight. We observed a bug when deploying a custom distributed optimizer. The developer added a layernorm operation before computing the key tensor in an attention layer, but did not register the layernorm operation's weight with the SP group optimizer. This meant that the layernorm weights were not considered during all-reduce, and thus the gradients computed by this implementation differed from those computed by the sequential model.

This is a case where user expectations (§4.4) played a role: it is possible to map G_d's outputs to G_s's, but these mappings are unexpected. We provided Entangle with an appropriate f_s and f_d, and Entangle reported an error when checking refinement given these expectations.

A.2 Bugs in Open-source Frameworks

Bug 6: Wrong scaling in gradient accumulation. This bug was first reported in 2021 [48] but was misattributed to numeric errors. It was re-reported and finally addressed in 2024 [25]. The bug manifests when gradient accumulation is enabled: gradient accumulation is an approach that splits a batch into multiple min-batches, thus allowing the use larger batch sizes. This approach is similar to the distribution strategies considered above, though the goal is to increase the batch size rather than the number of GPUs. We can also easily obtain a model without gradient accumulation (corresponding to G_s) and one with (corresponding to G_d), allowing us to use Entangle.

When using gradient accumulation, the programmer must scale the loss computation for correctness. Otherwise, the computed loss is much larger than would be expected. We evaluated Entangle's ability to find this bug by creating a simple regression that uses MSE loss.

Entangle detected this bug because the accumulated loss in G_d cannot cleanly represent the loss in G_s without computation because the loss needs to be scaled by number of accumulation steps in each batch.

Bug 7: Missing all-reduce in parallel linear layer due to mis-configuration. This is a bug previously reported in Megatron-LM [42]: when the TP size is larger than 1 the framework did not synchronize gradients from a parallel linear layer with an all-reduce. This leads to the wrong gradient being compute. Mathematically, the output of the parallel linear layers are $X_1 \times A_1$ and $X_2 \times A_2$ (where tensors X and A have been partitioned into X_1, X_2 and A_1, A_2) instead of the desired value $X \times A$ (which would require computing $X_1 \times A_1 + X_2 \times A_2$).

In Entangle, this bug manifests in a subsequent parallel `matmul` operator: the operator multiplies the linear layers output with a tensor B that has been partitioned into two tensors B_1 and B_2. However, the bug in the linear layer means that some elements, e.g., $X_2 \times A_2 \times B_1$ and $X_1 \times A_1 \times B_2$ are not computed. Consequently, the `matmul` output does not contain elements required to map to the output from G_s, and Entangle cannot find a clean relation.

Bug 8: Missing all-reduce in optimizer for tensor and sequence parallelized MOE router. This was another bug in Megatron-LM [30] that occurred when both TP and SP are enabled fro a MOE model, and weights for the router module were not synchronized due to a configuration model when finalizing the gradients.

This was another case where user expectations (§4.4) played a role: we could find a refinement from G_d's outputs to G_d, but the refinement relations in R_o differ from the approach adopted by Megatron-LM to combine weights for the router module.

Bug 9: Missing all-reduce in optimizer for sequence parallelized layernorm. The final bug occurs in TransformerEngine, and was caused when the developers of that framework implemented a new API for LayerNorm and RMSNorm [26]. The developers accidentally forgot to use all-reduce to aggregate weights when sequence parallelism was enabled.

This bug was another case where Entangle observed a violation of user intent: Entangle can find a refinement R_o (which uses an all-reduce) but the user expects that no additional operations are necessary, allowing us to identify the problem.

B Artifact Appendix

B.1 Abstract

This artifact provides the complete codes to infer tensor relations and verify equivalence. We also provides input computation graphs for the open-source frameworks, but we choose not to publish the graphs of the ByteDance's proprietary models.

With the artifact, one will be able to reproduce for the open-source models:

- (Figure 3) End-to-end verification time across different models (GPT, Qwen2 and Llama-3)
- (Figure 4) Scalability on verifying parallelized models
- (Figure 5) Lemmas complexity statistics
- (Figure 6) Heatmap showing the number of times each lemmas is used for different models

Please always refer to the README.md in the Github repository for updated information.

B.2 Artifact check-list (meta-information)

- **Compilation:** Requires Rust (cargo) to compile.
- **Run-time environment:** Ubuntu-22.04
- **Hardware:** We use a c6525-25g node in CloudLab [9] for evaluation, which has EPYC 7302P CPU and 128GB memory. If you have access to CloudLab, you can directly use the profile https://www.cloudlab.us/p/rdma-prefetch/Entangle. But the codes should also work on any other CPU machines with at least 32GB memory.
- **Execution:** Only need to run a single script to setup and several Python commands to run the experiments.
- **Metrics:** Running time cost, LOC.
- **Output:** Result data along with visualization similar to Figure 3, Figure 4, Figure 5 and Figure 6, where results of models from ByteDance are excluded.
- **Experiments:**

  - (Figure 3) End-to-end verification time across different models
  - (Figure 4) Scalability on verifying parallelized models.
  - (Figure 5) Lemmas complexity statistics (which requires manual counting and the results are put in the visualization script in advance).
  - (Figure 6) Heatmap showing the number of times each lemmas is used for different models.
- **How much disk space required (approximately)?:** 32GB
- **How much time is needed to prepare workflow (approximately)?:** Environment setup requires about 5 minutes.
- **How much time is needed to complete experiments (approximately)?:** 60 minutes if using the same hardware.
- **Publicly available?:** The repository is available on Github (https://github.com/nyu-systems/Entangle).
- **Code licenses:** Apache-2.0
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.17924206

B.3 Description

B.3.1 How to access. The repository is available on Github (https://github.com/nyu-systems/Entangle).

B.3.2 Hardware dependencies. We recommend using CloudLab to reproduce the results. And we provide a profile (https://www.cloudlab.us/p/rdma-prefetch/Entangle) for it. But the artifact should run on any CPU machine with at least 32GB memory.

B.3.3 Software dependencies. To get best performance, we recommend Ubuntu 22.04, with Python 3.12 and newest Rust installed. Any other Linux distribution, WSL or MacOS should also work.

B.4 Installation

B.4.1 Option 1 (Recommended): Using CloudLab. If you are using CloudLab, we provide a CloudLab profile and a single script to automatically set up the environment. You can find the profile here (https://www.cloudlab.us/p/rdma-prefetch/Entangle).

After starting an experiment with this profile, login to the shell and setup it following steps below:

1. Download the setup script https://github.com/nyu-systems/Entangle/blob/main/setup.sh to the "$HOME" directory.
2. Run "**source $HOME/setup.sh**" to activate the Rust and Python environment in current shell session.

This script will

1. install Rust (cargo) and uv
2. clone the repository to "/opt/tiger/Entangle"
3. set up the Python environment with uv
4. build and installs Entangle

B.4.2 Option 2: Manual Setup. Assuming Python 3.12 and Rust are installed. Then clone the repository and run the commands in Listing 5

```
# Assuming Python>=3.12 and Rust (cargo) installed.
# Assuming you are at root directory of the repository.
pip install -e .  # here is a dot at the end.
cd egger && cargo build --release
```

Listing 5: Manual Installation

B.5 Experiment workflow

The workflow is also described in README.md. Please run the experiments in the "examples" directory.

B.5.1 Step 1. Model Verification Experiments. We provided bug-free model verification experiments for all the models except the ByteDance's proprietary one. Run the commands in Listing 6 sequentially below to start the experiments:

```
# Assume you are in directory `examples`
python run_all.py gpt --all
python run_all.py qwen2 --all
python run_all.py aws_llama --all
```

Listing 6: Model Verification Experiments

If you see output messages like "Refinement verification succeeded for ..." in your terminal after each run, it means the verification is successful. Details about the output directory can be found in README.md.

B.5.2 Step 2. Bug Detection Experiments. We provided all the graphs for bug detection experiments except those from the ByteDance's proprietary model. Run the following commands sequentially below to start the experiments:

```
# Assume you are in directory `examples`
# Bug 6 in paper
python ./run_all.py grad_accumulation
# Bug 7 in paper
python ./run_all.py missing_allreduce_under_wrong_config
# Bug 8 in paper
python ./run_all.py missing_switchmlp_allreduce
# Bug 9 in paper
python ./run_all.py missing_layernorm_allreduce
```

Listing 7: Bug Detection Experiments

These runs are **expected to raise errors**. If you see either of the errors raised below, then you reproduce the result:

- entangle.sgraph.egraph.CannotFindPostconditions: Failed, check the conditions above.
- entangle.tools.egg.FailedImplyingEquivalence: User expectation violated.

B.5.3 Step 3. Visualization. To visualize the result from Appendix B.5.1, run the command in Listing 8

```
# Assume you are in directory `examples`
python visualization.py
```

Listing 8: Visualization Command

Appendix B.6 introduces how to evaluate and compare the result figures.

B.6 Evaluation and expected results

Ideally, you should be able to reproduce all the evaluation results except those involved ByteDance's models. The visualization result figures will be saved to the directory "examples/figures", including

- **one_layer_time.pdf**: the end-to-end verification time results (Figure 3). The performance results can vary when using different hardwares.
- **GPT_scalability.pdf**, **Llama-3_scalibility.pdf**: the scalability results (Figure 4), where you should see a bit super-linear time cost increasing with the parallel degrees and linear time cost increasing with the model sizes.
- **number_of_ops_and_lemmas.pdf**: the number of operators and lemmas used (Figure 5a), where you should see the same number as the one in the paper (except that data of ByteDance's proprietary models are removed).
- **lemma_loc.pdf**: the CDF of LOC of the lemmas (Figure 5b), where you should see a similar trend as the one in the paper (you will see some differences since we only include open-sourced models/framworks here).
- **lemma_applied_count_heatmap.pdf**: the heatmap of lemma application counts (Figure 6), where you should see something very similar to the one in the paper.