EMERGE: Checking Implementation Equivalence for LLMs via Rule Synthesis

Qi Zhan
Zhejiang University
Hangzhou, China
qizhan@zju.edu.cn

Xing Hu*
Zhejiang University
Hangzhou, China
xinghu@zju.edu.cn

Xin Xia
Zhejiang University
Hangzhou, China
Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security
Hangzhou, China
xin.xia@acm.org

Shanping Li
Zhejiang University
Hangzhou, China
shan@zju.edu.cn

Abstract

Verifying whether two implementations of the same large language model are equivalent across frameworks is difficult in practice. Even when they implement the same computation, the graphs may differ in operator decomposition, tensor layout, and the use of fused or opaque kernels, making manual rewrite rules hard to build and maintain. We present Emerge, a framework for checking *Implementation Equivalence* over computation graphs extracted from model implementations. Instead of writing rules manually, Emerge infers candidate relations from execution values, and synthesizes rewrite rules on demand when existing rules are insufficient. Each synthesized rule is then validated using SMT solving when possible and constraint-aware randomized testing for opaque kernels. As these local equivalences accumulate, Emerge eventually establishes whether the two implementations are equivalent. Our implementation targets inference computation graphs captured from HuggingFace Transformers and vLLM. Our evaluation shows that Emerge establishes equivalence for correct implementation pairs at practical cost. It also detects 10 of 13 known implementation bugs, uncovers 8 previously unknown implementation issues that were later confirmed by developers, and localizes the first mismatch when equivalence cannot be established.

CCS Concepts

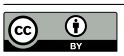
• **Software and its engineering → Software verification and validation**.

Keywords

Implementation Equivalence, Rule Synthesis, E-Graphs

ACM Reference Format:
Qi Zhan, Xing Hu, Xin Xia, and Shanping Li. 2026. EMERGE: Checking Implementation Equivalence for LLMs via Rule Synthesis. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3834405

*Corresponding author.



This work is licensed under a Creative Commons Attribution 4.0 International License.
ASE '26, Munich, Germany
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834405

1 Introduction

Deep learning deployment today relies on a diverse ecosystem of frameworks, compiler stacks, and high-performance serving systems, such as HuggingFace Transformers, XLA, TVM/Ansor, and vLLM [5, 14, 20, 39, 41, 45]. As a result, the same model architecture often has multiple implementations, optimized for serving efficiency or specific hardware targets. However, these implementations are not guaranteed to be equivalent. Even when derived from the same high-level model specification, they may differ substantially in computation graphs, operator decompositions, tensor layouts, and low-level kernels [10, 35, 40]. We refer to the problem of determining whether such implementations realize the same mathematical function as *Implementation Equivalence*.

When implementation equivalence does not hold, it often leads to *silent bugs*: they do not trigger crashes or exceptions, but instead produce incorrect intermediate tensors or outputs during normal execution [13]. Such failures have been observed in production ML frameworks, including incorrect loss scaling in Megatron-LM and output mismatches in vLLM caused by implementation bugs in components such as split-QKV handling and RoPE [4, 21, 33, 34].

Existing research has focused on equivalence checking for deep learning systems, particularly in the setting of *Parallel Equivalence* [12, 18, 35, 47]. In this setting, the goal is to verify that distributed execution strategies, such as tensor sharding or data parallelism, preserve the semantics of a reference implementation. However, verifying implementation equivalence beyond the parallel setting remains under-explored. For instance, the HuggingFace Transformers library [39] provides reference implementations of state-of-the-art models, while vLLM [14] offers highly optimized implementations. Although these frameworks target the same model architectures, their implementations can differ in computation graphs, tensor layouts, and kernel-level execution.

Recent work has explored equality saturation and e-graphs for proving equivalence in deep learning systems [35, 46, 47]. These approaches construct a unified representation of computation graphs and iteratively apply human-defined rewrite rules to establish equivalence. However, such manually authored rewrite rules are typically designed for parallel-equivalence settings and do not generalize

well to cross-framework model implementations. Unlike parallel-equivalence settings, where the main differences usually come from how tensors are partitioned across devices and later reduced or concatenated [28], cross-framework model implementations introduce two additional challenges:

Challenge 1: Divergent Implementation Paths. Even when mathematically equivalent, two implementations may use different operator decompositions, execution plans, and tensor layouts [10, 40]. For example, GPT-2 [24]'s linear projection appears as addmm in Transformers but as linear in vLLM, with the weight tensor stored in a transposed layout and the surrounding execution plan organized differently. As a result, the corresponding computation graphs can differ substantially in structure, making it impractical to manually encode rewrite rules for every such divergence.

Challenge 2: Opaque Semantics of Custom Kernels. To maximize inference-time performance, modern frameworks rely heavily on custom operators, often implemented as fused CUDA kernels or hardware-specific primitives [6, 26, 31]. These kernels make equivalence checking difficult because their semantics are often not explicit at the computation-graph level and are therefore hard to model precisely with existing verification techniques. In addition, they can introduce small numerical differences that accumulate along the computation graph [8], making direct output-based testing unreliable for establishing model equivalence. Manually writing rules for such operators requires significant domain expertise and does not generalize well across frameworks.

To address these challenges, we propose Emerge, a framework for checking Implementation Equivalence over computation graphs of different model implementations. The key insight is that we need not rely on a complete hand-written rule library. Instead, Emerge uses local correspondences inferred from executions to drive on-demand rule synthesis, allowing it to handle framework-specific differences and opaque kernels without extensive manual rule engineering.

Concretely, our approach represents both implementations in a joint e-graph [38] and incrementally establishes equivalence between their nodes. For each candidate relation, Emerge first attempts to establish it using available rules. If that fails, it synthesizes a new rule by abstracting over shared dependencies and validates the rule with SMT solving or constraint-aware randomized testing. Each newly established equivalence is then propagated through e-graph rebuilding, which can in turn expose further equivalences downstream. This design allows Emerge to handle divergent implementation paths, framework-specific fused operators, and opaque kernels within a single verification procedure.

We implement Emerge on computation graphs extracted by TorchDynamo [25] and evaluate it on model pairs drawn from Transformers and vLLM. Our evaluation shows that Emerge is effective along three dimensions. First, it detects real implementation bugs, including 10 of 13 known cases and 8 previously unknown issues, while providing useful fault localization signals. Second, it establishes equivalence for correct model pairs at practical cost. Third, it synthesizes block-level rewrite rules that compare favorably with manually authored ones.

Contributions. Our main contributions are as follows:

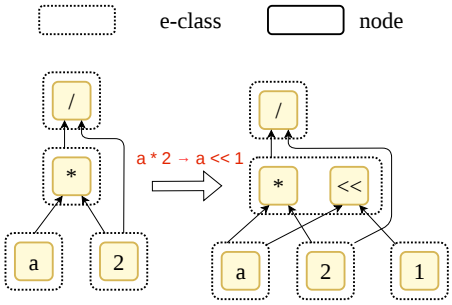


Figure 1: A small e-graph example. Dashed boxes are e-classes, and solid boxes are e-nodes. Applying the rewrite $a * 2 \rightarrow a \ll 1$ adds a shift e-node to the same e-class as the multiplication e-node, so the parent division can refer to either equivalent sub-expression through a single e-class.

- **Problem Formalization.** We formulate *Implementation Equivalence* as the problem of checking equivalence across multiple implementations of the same model.
- **Framework and Dynamic Rule Synthesis.** We propose Emerge, a framework over joint graphs, e-classes, and validated rewrite rules that synthesizes and validates rewrite rules on the fly.
- **Evaluation.** We evaluate Emerge on model pairs drawn from Transformers and vLLM, including bug reproduction, new-issue discovery, and equivalence checking on correct implementations.

2 Background

In this section, we review the concepts of e-graphs, computation graphs, and operators.

E-Graphs. An e-graph is a data structure for compactly representing many equivalent expressions at once [38]. It groups equivalent nodes into equivalence classes, or e-classes. Figure 1 illustrates the idea with the expression $(a * 2)/2$. Initially, each operator or constant appears in its own e-class. After applying the rewrite $a * 2 \rightarrow a \ll 1$, the multiplication node $*$ and the shift node $\ll$ are placed in the same e-class because they denote equivalent sub-expressions. The parent division node then points to this e-class rather than to one fixed child expression. Thus the same e-graph compactly represents both $(a * 2)/2$ and $(a \ll 1)/2$.

Computation Graphs and Operators. A deep learning implementation can be viewed as a computation graph, where nodes are tensor operators and edges carry tensors between operators. Common operators include matrix multiplication, reshape, layer normalization, and attention. Two frameworks may implement the same model with different graphs: one may decompose a computation into primitive operators, while another may use a fused kernel or a different tensor layout. In this paper, such operator nodes, input tensors, and parameters become e-nodes in a joint e-graph. Establishing implementation equivalence therefore amounts to showing that corresponding graph fragments, and eventually the two outputs, belong to the same e-classes.

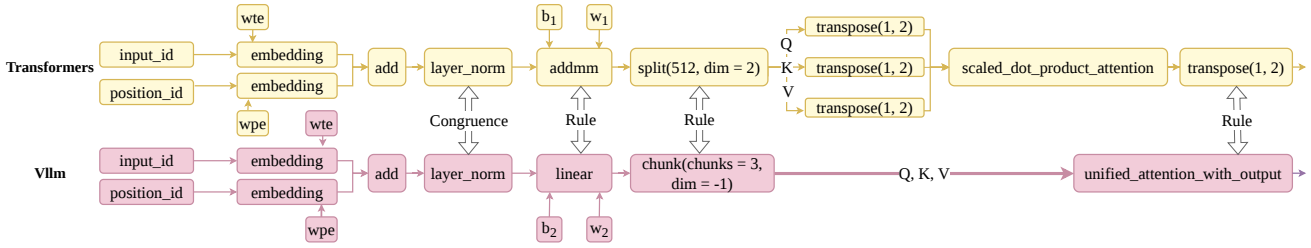


Figure 2: A fragment of a GPT-2 model used to illustrate equivalence verification between two implementations. Simplified and adjusted for clarity.

3 Overview

In this section, we provide a high-level overview of Emerge through an illustrative example, shown in Figure 2. We consider two implementations of the same GPT-2 subcomponent: one from HuggingFace Transformers, the other from vLLM. Although mathematically equivalent, they differ substantially in operator choices, computation graph structure, and tensor layouts.

Input Relation. We begin by establishing equivalences between the leaf nodes (inputs and parameters) of the two computation graphs. Since both implementations take the same token IDs and use semantically corresponding embedding weights, most input nodes can be matched directly. A more interesting case arises for the weight tensors w_1 and w_2. These represent the same logical matrix but are stored in transposed layouts by the two frameworks. By comparing the loaded weight values, Emerge automatically infers the relation $w_1 \iff \mathsf{transpose}(w_2)$.

Congruence and Graph Rebuilding. After aligning input nodes, Emerge propagates these relations to downstream operators through *congruence*: if $a \equiv b$, then $f(a) \equiv f(b)$ whenever the same operator f is applied in the same position. In this example, the two implementations share the same prefix computations, including the embedding, addition, and layer normalization blocks. Once their inputs are aligned, congruence automatically merges the corresponding intermediate nodes without requiring any rules. The layernorm and layernorm nodes in two implementations are merged directly through congruence. After each successful merge, Emerge rebuilds the e-graph to propagate the new equivalence.

Rule Synthesis and Application. After rebuilding, Emerge searches for node pairs that remain in different equivalence classes but have matching concrete values. Such pairs suggest candidate relations not yet captured by the current rule set. For the pair addmm and linear, Emerge synthesizes a candidate rewrite rule by abstracting over the concrete arguments:

$$\mathsf{linear}(a, \mathsf{transpose}(c), b) \iff \mathsf{addmm}(b, a, c).$$

The rule is then *validated* via randomized testing (§4.4). Once validated, it is added to the rule set and immediately applied to merge the two nodes into the same equivalence class. This merge is then propagated to downstream operators through rebuilding, allowing Emerge to lift the newly established equivalence through the rest of the graph (§4.6). Following the same procedure, Emerge synthesizes a rule relating split and chunk:

$$\mathsf{split}(a, 512, \mathrm{dim} = 2) \iff \mathsf{chunk}(a, \mathrm{chunks} = 3, \mathrm{dim} = -1),$$

which holds only when a has last dimension 3×512. This precondition is necessary because $\mathsf{split}(a, 512, \mathrm{dim} = 2)$ produces three outputs only in this case, matching the fixed three-way partition induced by $\mathsf{chunk}(a, \mathrm{chunks} = 3, \mathrm{dim} = -1)$. Emerge validates the rule formally via SMT solving under the shape constraints (§4.4).

Opaque Kernels. Finally, we consider the attention computation [32], where the two implementations differ most. Transformers decomposes attention into separate operators via scaled dot-product attention and output projection, whereas vLLM fuses the same computation into a single `unified_attention` kernel. Emerge infers the corresponding relation and synthesizes the following rule between scaled_dot_product_attention and unified_attention:

$$\mathsf{unified_attention}(Q, K, V) \iff$$

$$\mathsf{transpose}\left(\mathsf{scaled_dot_product_attention}\begin{pmatrix}\mathsf{transpose}(Q)\\ \mathsf{transpose}(K)\\ \mathsf{transpose}(V)\end{pmatrix}\right).$$

Here we abbreviate $\mathsf{transpose}(a, 1, 2)$ as $\mathsf{transpose}(a)$ for clarity. Since this kernel is opaque to symbolic reasoning, Emerge validates the rule via randomized testing (§4.4).

At this point, the output nodes belong to the same equivalence class, establishing the implementation equivalence of the two graphs. Although this walkthrough covers only a fragment of GPT-2, the synthesized rules are immediately reusable across later layers, amortizing the synthesis cost across the entire model. This example conveys the core intuition behind Emerge, but omits design details, such as relation inference, rule synthesis and validation, and e-graph maintenance. We now turn to the full framework.

4 Approach

Emerge checks the functional equivalence of two computation graphs through an iterative procedure on a joint e-graph. The overall workflow is illustrated in Algorithm 1. Given two computation graphs, Emerge repeatedly proposes candidate relations between nodes (§4.2) and attempts to establish them using existing rules (§4.5). When no existing rule applies, it synthesizes a candidate rule (§4.3) from the current e-graph state and validates the rule (§4.4) before merging the corresponding nodes. After each successful merge, the e-graph is rebuilt to maintain congruence and refresh attached execution data (§4.6). At the end of each iteration, Emerge checks

whether the output nodes o_1 and o_2 belong to the same equivalence class. If so, the two graphs are declared *equivalent*. If the predefined iteration limit is reached first, EMERGE reports failure and invokes failure localization to identify the first mismatched node pair that explains the remaining divergence (§4.7).

```
def compare(P, Q):
  G = initialize_egraph(P, Q) # build joint e-graph
  R = ∅
  for i in range(MAX_ITERATIONS):
    # may augment G with auxiliary nodes
    for (u, v) in find_possible_relations(G): # §4.2
      proved = apply_rules(R, u, v) # §4.5
      if not proved: # otherwise, synthesize new rule
        rule = synthesize_rule(G, u, v) # §4.3
        if rule and validate(rule):  # §4.4
          R = R ∪ {rule}
          proved = True
      if proved:
        G.merge(u, v)
        G.rebuild()  # §4.6
    if outputs_equivalent(G):
      return EQUIVALENT
  return FAILED, localize(G) # §4.7
```

Algorithm 1: High-level Workflow of EMERGE.

4.1 Formalization

We begin by formalizing the key components of our approach.

Computation Graph. A computation graph is represented as a directed acyclic graph $G = (V, E)$, where each node $v \in V$ represents either an operation or an input tensor and each edge $(u, v) \in E$ indicates that the output of node u is an input to node v. Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, we construct a joint graph $G = (V, E)$ where $V = V_1 \cup V_2$ and $E = E_1 \cup E_2$.

Implementation Equivalence. Two computation graphs are implementation equivalent if they realize the same mathematical function over their shared logical inputs and parameters under real-valued semantics. In particular, implementation equivalence abstracts away from graph-level structural differences and depends only on the induced function. In our setting, this equivalence is instantiated over the captured inference graphs and the logical inputs exercised by the checking run. Our algorithm attempts to establish this property by incrementally constructing a joint e-graph that places the designated outputs of the two graphs in the same e-class.

Equivalence Relation and E-Classes. During verification, EMERGE maintains a node-level equivalence relation $\equiv$ over the nodes of the joint graph G. This relation induces a partition of V into equivalence classes (e-classes). For a node $n \in V$, we write $\overline{n}$ for the e-class containing n. Intuitively, nodes in the same e-class are treated as equivalent by the e-graph. Each e-class is associated with concrete execution data obtained from dynamic execution. We denote this attachment by a partial function $f : \overline{V} \to D$, where D is the domain of runtime tensor values. Throughout the paper, we use plain symbols such as n, u, v for nodes and overlined symbols such as $\overline{n}, \overline{u}, \overline{v}$ for their corresponding e-classes.

Rewrite Rules. A rewrite rule is an equivalence of the form $l \iff r$, where l and r are operator terms built from the same operator vocabulary as the computation graph. An operator term is either a variable, a literal constant or an operator applied to a finite number of sub-terms. We write $\mathrm{FV}(t)$ for the free variables of a term t. A substitution σ maps the free variables of a rule to e-classes. Intuitively, a rewrite rule describes two parameterized computation subgraphs that produce the same result under all consistent substitutions of their free variables. In our system, rewrite rules are treated symmetrically, so either side may be matched and used to establish equivalence.

Maintained Structures. During verification, EMERGE maintains the joint graph G, the current equivalence relation $\equiv$, the attached execution data f, and the set $\mathcal{R}$ of validated rewrite rules. The rule set may be initialized either as empty or with a seed set of manually authored rules. These components are updated iteratively as new relations are inferred, rules are synthesized, and e-classes are merged and rebuilt. Our algorithm attempts to check the implementation-equivalence by placing the output nodes $o_1 \in V_1$ and $o_2 \in V_2$ in the same e-class[1].

The following sections use the running example in Figure 3. Dashed boxes denote e-classes and solid boxes denote nodes. In ①, the top subgraph computes a linear projection through the decomposed pattern $\mathrm{add}(\mathrm{mm}(i_1, w_1), b_1)$, while the bottom subgraph uses the fused operator $\mathrm{linear}(i_2, w_2, b_2)$. Although the two forms differ in operator structure and weight layout, they correspond to the same logical computation. We use this example to show how EMERGE first aligns the inputs in ②, then synthesizes a rewrite rule and merges the two outputs into the same e-class in ③.

4.2 Relation Inference

EMERGE infers candidate relations by comparing the concrete tensor values attached to e-classes. For two e-classes $\overline{u}$ and $\overline{v}$, if $f(\overline{u})$ and $f(\overline{v})$ match up to a configurable numerical tolerance ϵ, then $(\overline{u}, \overline{v})$ is treated as a candidate relation. This direct comparison is effective in many cases, but it is not sufficient on its own because semantically equivalent tensors may differ in layout, partitioning, or structural organization. For example, one may be the transpose of the other or a concatenation of its components. As illustrated in Figure 3, the weight matrices w_1 and w_2 are stored in different layouts. We therefore need to introduce an auxiliary node to relate them.

To handle such cases, EMERGE uses a small *transformation grammar* to describe value-level operations that relate semantically equivalent tensors with different layouts or structures. The search space is given by the following context-free grammar:

$$\begin{aligned} E \quad \to\ & \mathrm{transpose}(E, \mathrm{axis}_1, \mathrm{axis}_2) \mid \mathrm{concat}(E, \ldots, E, \mathrm{axis}) \\ & \mid \mathrm{reshape}(E, \mathrm{shapes}) \mid \mathrm{split}(E, \mathrm{size}, \mathrm{axis}) \end{aligned}$$

This grammar captures common layout and structural transformations in deep learning frameworks, including reordering, reshaping, and partitioning. EMERGE then performs bottom-up enumerative synthesis over this grammar to search for an expression that relates the two tensor values. In many common cases, the discovered

[1]Without loss of generality, we assume single output nodes. In practice, both graphs may have multiple outputs, and we require pairwise equivalence.

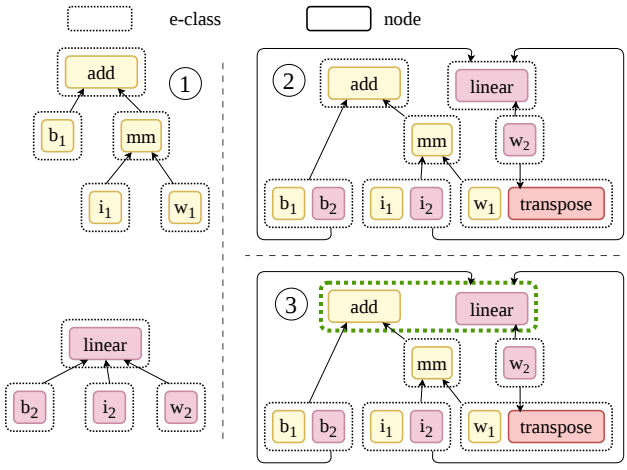


Figure 3: Rule synthesis from execution traces. ① Initial relation ② Relation inferred from input values ③ Relation inferred from rule synthesis.

transformation also induces a reverse relation under compatible parameters (e.g., concat and split along the same axis), so it is often sufficient to search in one direction. The current grammar is intentionally small and captures only a subset of value-level transformations observed in practice. When two implementations differ by a transformation outside this grammar, EMERGE may fail to infer the corresponding relation.

Once such a transformation is found, EMERGE adds the corresponding auxiliary nodes to the joint graph and uses them to relate the original nodes. If both original nodes are leaves, such as model inputs or parameters, the transformed node can be merged directly with its counterpart to establish an initial equivalence. In Figure 3, EMERGE introduces the auxiliary node transpose on top of w_2 and merges it with w_1 in ②. If the matched nodes are non-leaf nodes, however, the added auxiliary structure only exposes a candidate equivalence. EMERGE must still justify it through rule application (§4.5) or, when no existing rule applies, through rule synthesis (§4.3).

4.3 Rule Synthesis

When EMERGE identifies a candidate relation between e-classes $\overline{u}$ and $\overline{v}$, it first attempts to establish it using existing rules in $\mathcal{R}$. If no existing rule applies, EMERGE then synthesizes a rule using representative nodes $u \in \overline{u}$ and $v \in \overline{v}$. Concretely, rule synthesis seeks an equivalence $l \Longleftrightarrow r$ such that l matches u and r matches v in the current e-graph. However, not every pair of target e-classes admits such a rule. We therefore first determine whether the two sides can be abstracted over a common frontier.

We introduce the notion of an *abstraction frontier* of an equivalence class. Intuitively, a frontier is a set of e-classes whose values jointly determine the value of the target e-class. Formally, for an equivalence class $\overline{n}$, we write $C(\overline{n}) \subseteq \mathcal{P}(\overline{V})$ for its set of candidate abstraction frontiers. For a representative node $n \in \overline{n}$, we write $C(n)$ for the frontier family obtained by expanding through n. These two families are defined recursively as

$$C(n) = \begin{cases} \{\{\overline{n}\}\}, & n \text{ is a leaf} \\ \{\{\overline{n}\}\} \cup \left\{ \bigcup_{i=1}^{k} S_i \;\middle|\; \vec{S} \in \prod_{i=1}^{k} C(\overline{l_i}) \right\}, & \text{otherwise} \end{cases} \tag{1}$$

$$C(\overline{n}) = \bigcup_{m \in \overline{n}} C(m), \tag{2}$$

where $\overline{l_1}, \ldots, \overline{l_k}$ are the children of n and $\vec{S} = (S_1, \ldots, S_k)$. Intuitively, frontier enumeration may either stop at the current e-class or continue expanding through its children. Taking the union over all representatives allows enumeration to proceed through any equivalent form already present in the e-graph. For example, in Figure 3, $\{\overline{\text{add}}\} \in C(\overline{\text{add}})$, $\{\overline{\text{mm}}, \overline{b_1}\} \in C(\overline{\text{add}})$, and $\{\overline{b_1}, \overline{w_1}, \overline{i_1}\} \in C(\overline{\text{add}})$.

Two equivalence classes $\overline{u}$ and $\overline{v}$ can admit a rewrite rule only if they share a common frontier, i.e., $C(\overline{u}) \cap C(\overline{v}) \neq \emptyset$. If their frontier families are disjoint, then no meaningful symbolic relation can be synthesized between them. In the running example, after input alignment we have $\overline{b_1} = \overline{b_2}$ and $\overline{i_1} = \overline{i_2}$, so the add and linear nodes share the common frontier $\{\overline{b_1}, \overline{i_1}, \overline{w_2}\}$. By contrast, the mm node and the linear node do not share any common frontier, and hence no rule can be synthesized between them.

A key design goal is not merely to synthesize a valid rule, but to favor more general ones whenever possible. Coarser frontiers stop earlier and yield more general abstractions, while finer frontiers expand deeper and preserve more internal structure. When multiple common frontiers are available, they induce symbolic patterns at different abstraction levels. If frontier enumeration stops too early, the resulting rule preserves unnecessary internal structure and becomes overly specific. For example, a naive synthesis procedure might stop at the e-class containing $\text{layernorm}(a)$ and produce the overly specific rule $\text{linear}(\text{layernorm}(a), \text{transpose}(c), b) \Leftrightarrow \text{addmm}(b, \text{layernorm}(a), c)$, which unnecessarily constrains the first argument to be a layernorm output.

Equipped with the above definitions, we now present the algorithm for rule synthesis, shown in Algorithm 2. Here Π denotes Cartesian-product enumeration. For each target e-class, the algorithm enumerates a sequence of frontier-pattern pairs (line 2–3). Each pair (S, p) consists of a frontier S and a symbolic pattern p obtained by abstracting the target over S. The enumeration proceeds in a top-down manner, from the coarsest abstraction to progressively finer ones. Starting from a target e-class $\overline{n}$, the algorithm first yields the frontier $\{\overline{n}\}$ (line 9), treating the entire e-class as a single symbolic atom. It then recursively expands representative nodes toward their inputs (lines 11–26), refining the current frontier and constructing correspondingly more specific symbolic patterns. In this way, frontier-pattern pairs are generated in decreasing order of generality. Given the two enumerated sets P_u and P_v, the procedure searches over $\Pi(P_u, P_v)$ for a pair $(S_u, p_u) \in P_u$ and $(S_v, p_v) \in P_v$ such that $S_u = S_v$ (lines 4–6). Because frontier enumeration prioritizes coarser frontiers, the first shared frontier found under the current traversal order gives an early and typically more general candidate rule under the current e-graph: $p_u \Longleftrightarrow p_v$. If no such pair exists, the algorithm concludes that no rule can be synthesized for $\overline{u}$ and $\overline{v}$ under the current graph state.

In the running example, the extracted symbolic pattern summarizes the decomposed subgraph $\text{add}(\text{mm}(i_1, w_1), b_1)$ as the compound term $\text{addmm}(b, a, c)$. In the end, EMERGE synthesizes the

```
def synthesize_rule(egraph, ū, v̄):
  P_u = enumerate_frontier_patterns(egraph, ū)
  P_v = enumerate_frontier_patterns(egraph, v̄)
  for ((S_u, p_u), (S_v, p_v)) in Π(P_u, P_v):
    if S_u == S_v: # depend on the same frontier
        return Rule(p_u, p_v)

def enumerate_frontier_patterns(egraph, n̄):
  yield ({n̄}, Atom(n̄))
  # Expand through operators in the same e-class
  for n in n̄:
      if n.op == "input":
          continue
      child_patterns = [
          enumerate_frontier_patterns(egraph, arg)
          for arg in n.args
      ]
      # Cartesian product over children
      for combo in Π(*child_patterns):
          S = set()
          args = []
          for (S_i, p_i) in combo:
              S = S ∪ S_i
              args.append(p_i)
          p = Node(n.op, args)
          yield (S, p)
```

Algorithm 2: Rule synthesis by shared frontier enumeration.

equivalence $\mathsf{linear}(a, \mathsf{transpose}(c), b) \Longleftrightarrow \mathsf{addmm}(b, a, c)$, thereby merging the add root and the linear root in ③ of Figure 3.

4.4 Rule Validation

Synthesized rules are not admitted into $\mathcal{R}$ immediately. Before a rule can participate in equality saturation, Emerge validates it and records the strength of the resulting evidence. Validation first infers the rule's preconditions from operator semantics. These include shape constraints, operator-specific side conditions, and domain restrictions on sampled inputs. For example, the validator checks matrix-multiplication alignment, divisibility requirements for chunk and split, and index bounds for embedding. Rules whose preconditions are malformed or unsatisfiable are rejected directly. The remaining rules are validated using symbolic verification when the involved operators are supported by our SMT model, and constraint-aware randomized testing otherwise.

Symbolic Verification via SMT. For rules whose semantics can be modeled precisely, Emerge invokes an SMT solver to prove equivalence under the inferred constraints.

- Scalar Arithmetic. Expressions are lowered to SMT real, integer, and boolean terms. The solver checks the negation of the candidate equivalence. If the negation is unsatisfiable, the rule is proved correct and marked *formally verified*.
- Tensor Operation. For tensor rules, Emerge models each tensor as a symbolic array whose elements are distinct symbolic variables. Structural operators such as transpose and reshape are lowered to index transformations, while operators such as exp are modeled element-wise using uninterpreted functions. Under satisfiable shape constraints, the solver proves element-wise equality between the two sides.

In both cases, the rule and its inferred preconditions can be encoded exactly in SMT, so equivalence is proved symbolically under the inferred constraints. These SMT operator models are implemented manually for the supported subset, with tensors represented as NumPy arrays whose elements are Z3 symbolic values; this lets many tensor transformations reuse NumPy-style array indexing and reshaping.

Constraint-Aware Randomized Testing. For rules involving fused kernels or operators outside the supported SMT, Emerge falls back to randomized testing. The validator first solves the inferred preconditions and then samples shape- and domain-valid inputs that satisfy them. In particular, it solves constraints such as `shape_dim_eq` and `numel_eq` to generate valid tensor shapes, and it uses operator-aware domains for restricted inputs such as indices to embedding. In our evaluation, randomized validation reuses the shapes and dtypes observed in the captured subgraphs, because synthesized rules are applied only when the matched subgraphs have compatible shapes and dtypes. It generates 16 random test cases per rule, samples floating-point inputs from a normal distribution, and compares outputs with atol = 10^{-2} and rtol = 10^{-2}. A rule is accepted only if the two sides produce matching outputs, within the configured numerical tolerance.

Favoring general rules is also important for validation. Smaller and more structural rules are often easier to verify with SMT, whereas overly specific rules often include extra operators that make symbolic verification harder and therefore increase reliance on empirical validation.

Finally, Emerge assigns each accepted rule a validation level. Rules proved by SMT are labeled *formally verified*, while rules that pass constraint-aware randomized testing are labeled *empirically validated*. Only rules with one of these accepted validation levels are inserted into $\mathcal{R}$ and used in subsequent process. As a result, the soundness of our approach depends on the correctness of the accepted rules: formally verified rules carry formal guarantees, whereas empirically validated rules rely on the testing procedure.

4.5 Rule Application

Rule application in Emerge begins only after relation inference proposes a candidate pair of nodes. Emerge then checks whether an existing validated rule can justify that pair. Unlike conventional term rewriting, matching in Emerge is *e-class aware*: because an e-class may contain multiple nodes, a rule need not match a single fixed node form. Instead, it may match through any representative already contained in the e-classes.

Formally, a rule $l \Longleftrightarrow r$ matches a pair of e-classes $(\overline{u}, \overline{v})$ if there exists a substitution $\sigma : \mathrm{FV}(l) \cup \mathrm{FV}(r) \to \overline{V}$ such that l matches $\overline{u}$ and r matches $\overline{v}$ under the same substitution σ. Matching is existential over representatives: at every position in the pattern, the operator or atomic term may be matched by any representative already contained in the corresponding e-class, and a single consistent choice is sufficient. Literal constants and operator parameters are matched syntactically as part of the term structure, rather than

through σ. If a rule carries additional preconditions (e.g., shape divisibility constraints inferred during synthesis), the match is accepted only when these constraints are satisfied by the current substitution. Since rules are treated symmetrically in our system, either direction is accepted.

Each matched pair (u, v) provides a concrete instance that the rule relates the two e-classes under the current bindings. EMERGE merges the corresponding e-classes and invokes the rebuilding procedure (§4.6) to maintain congruence.

4.6 E-Class Merge and Graph Rebuild

When two e-classes $\overline{u}$ and $\overline{v}$ are merged, the new equivalence may induce further merges among their parent nodes through congruence. To maintain this invariant, EMERGE performs an iterative rebuilding phase until no new congruences arise.

(1) **Canonicalize Parent References.** For every parent node p that uses $\overline{u}$ or $\overline{v}$ as an argument, EMERGE replaces that argument with the canonical representative of the newly merged e-class.
(2) **Detect New Congruences.** After canonicalization, EMERGE checks whether any parent nodes have become structurally congruent, i.e., they have the same operator and their corresponding arguments now refer to the same e-classes. If so, they are merged and added to the worklist for further rebuilding.
(3) **Refresh Execution Data.** The merged e-class retains one attached value as its representative concrete value. Then, for each parent node whose input e-classes have changed, EMERGE re-executes the corresponding operator on the updated representative inputs to obtain a refreshed output value.

4.7 Fault Localization

When verification terminates without merging the output e-classes, EMERGE performs failure localization to identify the earliest substantive divergence between the two implementations. Rather than reporting only the final unmatched outputs, EMERGE traces the mismatch backward through the e-graph and returns a pair of frontier nodes that best explains why equivalence could not be established.

The localization procedure is heuristic-guided. Starting from the outputs, it traverses backward along data dependencies while skipping nodes treated as semantically transparent by the heuristic. It follows the primary data-carrying inputs of each operator until it reaches the first frontier whose e-class has no cross-graph counterpart. Among these frontiers, EMERGE reports the earliest unexplained mismatch.

5 Implementation

EMERGE is implemented on top of TorchDynamo [25], following prior work [35]. It captures computation graphs from production code, such as vLLM and HuggingFace, without manual instrumentation. To compactly represent and reason about many equivalent expressions, we use an e-graph with union-find [29], following the design of egg [38]. Rule validation uses Z3-backed [7] SMT checking together with constraint-aware randomized testing. This section focuses on two practical aspects of the implementation: specialized input matching and efficiency optimizations.

5.1 Practical Input Matching

Although input relations can in principle be inferred by the general relation inference procedure in §4.2, inputs only need to be matched against other inputs. Running the full machinery over all input pairs is therefore unnecessarily expensive in practice. We therefore implement a specialized input-matching pipeline. Given input sets I_1 and I_2 from graphs G_1 and G_2, the goal is to construct a small candidate set $E_{\text{input}} \subseteq I_1 \times I_2$ that is highly likely to contain the true correspondences.

Our current implementation proceeds in two stages. First, it partitions inputs by layer using layer-identifying substrings in their names, which are usually easy. Second, for each layer pair, it applies a lightweight matching heuristic to propose an initial set of possible input relations. This heuristic is used only to narrow the search space, and every pair must still be validated by the core relation inference procedure. Value-based matching uses absolute and relative tolerances of 10^{-2} by default.

5.2 Efficiency Optimizations

We implement several optimizations to improve the efficiency of EMERGE in practice:

Dependency-Based Blocking. Two e-classes can be equivalent only if they depend on the same logical inputs. We therefore compute a *dependency signature* for each e-class (the set of reachable input leaf e-classes) and compare candidate relations only when they have identical signatures. This prevents many impossible comparisons from being considered and reduces the search space.

Efficient Frontier Enumeration. Rule synthesis is driven by shared-frontier discovery, so we optimize frontier enumeration in two ways. First, for a candidate relation $(\overline{u}, \overline{v})$, we precompute the e-classes reachable from both endpoints and restrict enumeration to this intersection, pruning branches that cannot yield a shared frontier. Second, Cartesian products in frontier enumeration (Algorithm 2, line 19) are generated lazily with Python generators and traversed in diagonal order, so that coarser, more general candidates are explored earlier. Together, these optimizations avoid materializing exponentially many combinations and allow synthesis to terminate as soon as the first shared frontier is found without exhaustively enumerating the entire search space.

6 Evaluation

We evaluate EMERGE on the following research questions (RQs):

- **RQ1 (Bug Detection):** Can EMERGE effectively detect and localize real-world implementation bugs?
- **RQ2 (Equivalence & Scalability):** Can EMERGE establish equivalence for correct implementation pairs at practical cost?
- **RQ3 (Rule Characterization):** What rules does EMERGE synthesize, and how do they compare with manually authored rules?

6.1 Experimental Setup

Baselines. A direct end-to-end baseline is not available for our full setting: existing techniques mainly target tensor-parallel transformations within a shared framework and operator semantics, whereas our setting involves cross-framework implementations. We therefore use task-specific baselines: TTrace [12], a trace-based bug diagnosis method that we reproduce in our evaluation, serves

Table 1: Known bug cases used in our evaluation and detection results compared with TTrace.

ID	Emerge	TTrace	Description	Issue	Fix
1	◐	✗	InternLM2: wrong `split_qkv` execution semantics in tensor parallel	8017	pull/8055
2	○	✗	OLMo: wrong shard/full weight-loading semantics in tensor parallel	3775	pull/3869
3	◐	◐	Gemma: MLP GELU approximation mismatch (`tanh` vs exact)	3411	pull/3653
4	◐	◐	GPT-J: incorrect RoPE rotation scheme	590	pull/941
5	○	◐	GLM: rotary / attention path mismatch	16904	pull/16912
6	●	○	Command-R: wrong RoPE base parameter priority	3892	pull/3919
7	●	◐	Gemma2: context/window boundary exposed as rotary/cache inconsistency	6220	pull/10584
8	●	○	Qwen2-VL: weights mapper / parameter-path mismatch	18976	pull/19054
9	●	✗	Phi: declared max position exceeds materialized rotary cache size	9502	pull/9503
10	◐	◐	Llama4-Vision: wrong vision rotary cache representation / dtype path	25888	pull/25889
11	✗	✗	Mistral long prompt / sliding-window behavior; semantics are hidden	2059	pull/2088
12	✗	✗	Phi-3.5: LongRoPE branch / KV-cache semantics hidden by data-dependent branching	27414	pull/27431
13	✗	✗	GGUF: TP embedding/materialization bug not isolated under current method	7880	pull/7954

Legend (Detected): ● exact bug root, ◐ same functional block, ○ farther from the root, and ✗ missed.

as the baseline for bug detection in RQ1, and Entangle [35], a manually authored rule library for tensor-parallel equivalence, serves as the baseline for rule characterization in RQ3.

Inputs and Configurations. For each bug-reproduction case, we use the configuration that triggers the reported semantic mismatch, such as the relevant attention backend, scaling flag, tensor-parallel setting, or long-context option. For positive equivalence cases, we use default inference configurations unless the compared implementation feature requires a specific flag. Sequence lengths are small fixed constants chosen to exercise the target computation path. Automated exploration of input and configuration coverage is outside the scope of this evaluation.

Environment. All experiments are conducted on our local server with an Intel Xeon Platinum 8358P CPU, 2 TB of memory, and one NVIDIA A800 80 GB PCIe GPU, running Ubuntu 20.04.6 LTS. All experiments use a maximum of 3 iterations.

6.2 RQ1: Bug Detection

We answer RQ1 through two studies: reproducing known implementation bugs and discovering previously unknown bugs in current releases. We evaluate localization quality at three levels. ● denotes that the reported node is the root-cause operator. ◐ denotes that it falls in the same functional block. ○ denotes that it lies further downstream. Here, a functional block denotes the smallest model component implementing a coherent sub-computation, such as rotary embedding, attention, or MLP, as identified from the captured computation graph and the corresponding implementation code.

6.2.1 Reproducing Known Bugs. We collected 13 confirmed, independently fixed bugs by searching the issue trackers of vLLM [14] and HuggingFace Transformers [39] with output-mismatch keywords such as "wrong output", "incorrect generation", and "mismatch". We kept reproducible inference-semantics bugs that could be reduced into graph pairs, excluded cases whose relevant code paths could no longer be reconstructed, and constructed a model pair for each remaining bug to exercise the affected code path. We then run Emerge and record (i) whether it reports non-equivalence, and (ii) the first reported mismatch returned by the localization procedure. For baseline comparison, we run TTrace on the same model pairs under the same configurations and record whether it reports a mismatch and its localization result when available. We report these results in Table 1.

Table 2: Unknown issues identified by Emerge.

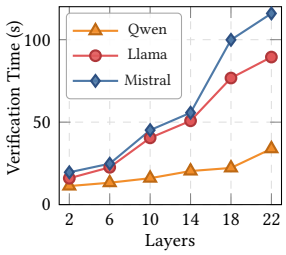
ID	Our	Description	Issue	Fix
14	●	GPT-2: `scale_attn_weights` ignored in SDPA path	44380	pull/44397
15	●	GPT-2 family: `scale_attn_weights` mismatch in vLLM	35402	pull/35436
16	●	GPT-BigCode: `scale_attn_weights` mismatch in vLLM	36565	pull/36637
17	○	Qwen1: long-context `logn_attn` missing in vLLM	36880	pull/37089
18	●	PhiMoE: `router_jitter_noise` hard-coded	35494	pull/35499
19	◐	OLMoE: `clip_qkv` ignored	35513	pull/35523
20	○	JAIS: learned positional embeddings still trigger ALiBi in vLLM attention	37400	pull/37820
21	●	Phi: `qk_layernorm` missing in vLLM	37852	pull/37870

Emerge detects **10** of the 13 known bugs. Among the detected cases, 6 are localized exactly (●), 3 are localized to the same functional block (◐), and 1 is localized further downstream (○). For example, Bug #4 (GPT-J incorrect RoPE) is localized directly to the `rotary_embedding` operator, while Bug #3 (Gemma GELU mismatch) is pinpointed at the `gelu` node. In these cases, Emerge reduces debugging effort from inspecting the entire model execution to inspecting a single operator and its immediate context. Three bugs remain undetected: two lie behind data-dependent branches not exercised by our current inputs, and one arises from weight-loading semantics outside the current graph-extraction boundary.

Compared with TTrace, Emerge achieves both higher detection coverage and more precise localization on the known-bug

Model	Params	Time (s)
GPT-2	70m	11.03
Qwen	0.5b	22.59
Phi	3.8b	147.99
Llama3	8b	148.33
Mistral	12b	350.09

(a) End-to-end verification.



(b) Scalability trends.

Figure 4: RQ2 results on correctness and scalability.

benchmark. TTrace detects 7 of the 13 bugs and does not achieve exact localization on any case, whereas EMERGE detects 10 and localizes 6 exactly. TTrace localizes bugs by comparing execution traces against a trusted reference under carefully designed numerical tolerances, whereas EMERGE incrementally establishes local equivalences, merges the corresponding e-classes, and propagates these merges through rebuilding. As a result, already explained differences are removed from following analysis, which improves both detection coverage and localization precision by focusing on the first frontier that remains unmatched.

6.2.2 Discovering New Bugs. Beyond reproducing historical bugs, we ran EMERGE on recent releases of all evaluated model families under both default and non-default configurations such as alternative attention backends. This process uncovered 8 previously unknown implementation issues (Table 2). We manually inspected the reports before submission to remove duplicates and non-semantic differences. All 8 issues have been confirmed by developers, 2 fixes have already been merged, and the remaining 6 are under review. We highlight two representative cases below.

GPT-2 `scale_attn_weights` *(Bugs #14–#15).* EMERGE found that the `scale_factor` parameter is silently ignored under the SDPA and FlashAttention backends in both Transformers and vLLM. In our framework, the candidate rule for the corresponding attention computation fails validation because the scaling factor is absent from one side. This bug is configuration-sensitive and would not surface under the default backend. Fixes have been merged in Transformers (PR#44397) and are under review in vLLM (PR#35436).

Qwen1 `logn_attn` *(Bug #17).* Qwen1 applies a $\log n$ attention scaling during long-context inference, but this logic was entirely missing in vLLM. EMERGE reported non-equivalence along the attention path and localized the discrepancy to the missing scaling behavior. Unlike configuration-specific bugs that only surface under non-default settings, this bug can already be triggered under the default configuration once the input length exceeds the base context window.

6.3 RQ2: Equivalence Proving and Scalability

In this section, we evaluate EMERGE's ability to establish equivalence for correct implementation pairs and analyze how verification time scales with model size.

Our benchmark covers five model families: GPT2 [24], Qwen [3], Llama [9], Mistral [11], and Phi [1], which are among the most widely adopted open-weight language models in the research community. Figure 4a reports the end-to-end verification time for each model family, together with the corresponding model size in parameters. Across these model families, the measured verification time ranges from 11.03 seconds for GPT-2 70M to 350.09 seconds for Mistral 12B. For every model in the benchmark, EMERGE establishes that the output nodes of the two implementations belong to the same equivalence class, confirming equivalence. We did not observe any spurious non-equivalence on these correct implementation pairs. All reported runtimes are obtained with the full optimized implementation described in §5.2. Disabling either dependency-based blocking or efficient frontier enumeration, the two optimizations described in §5.2, caused verification to exceed one hour on the complete benchmark. This indicates that these optimizations are essential to making the approach practical.

Figure 4b shows the scalability trends on three representative model families: Qwen, Llama, and Mistral. As shown in the figure, verification time increases with model depth and overall architectural complexity. Qwen grows relatively steadily over the evaluated range, whereas Llama and Mistral increase much more sharply at larger depths, with Mistral exceeding 100 seconds at 22 layers. A likely explanation is the growth of the candidate search space: after value-based filtering, EMERGE still needs to consider many possible relations among surviving nodes, and the number of such candidate pairs can grow quadratically in the worst case. As a result, deeper and more structurally diverse models require more verification work as depth increases. Despite this increase, EMERGE remains practical on the evaluated models.

6.4 RQ3: Rule Characterization

We characterize the rules synthesized by EMERGE from two perspectives: the distinct rule patterns that emerge across all experiments, and the granularity of these synthesized rules relative to manually authored rules.

6.4.1 Synthesized Rule Catalogue. Across the five full-model positive examples in our benchmark, EMERGE synthesizes 86 rule instances, corresponding to 54 unique rules. Each model requires a small learned rule set, ranging from 14 to 21 rules. This indicates that the checking process relies on a compact reusable core supplemented by only a few model-specific rules, rather than a large rule library. The learned rules exhibit a clear reuse pattern. 12 of the 54 distinct rules appear in at least two models, and several recur in all models, including wrapper elimination, dtype normalization, and simple algebraic normalization. The most common rule families are general layout and normalization rewrites. In contrast, model-specific differences are concentrated in a small number of bridge rules that capture architectural variation, especially different attention implementations (e.g., decomposed scaled-dot-product attention versus fused attention kernels), as well as model-specific rotary/cache and fused-MLP paths.

Of the 86 rule instances, 64 are *formally verified* and 22 are *empirically validated*. Most rules over explicit tensor transformations, such as layout, slicing, and normalization rewrites, can be discharged by SMT. By contrast, rules involving opaque or fused operators, especially attention-related ones, are validated through randomized testing. This division reflects the design of EMERGE:

Emerge (Ours): Automatically Synthesized Rules

(R1) $\text{linear}(\text{gelu}(\text{linear}(x,\ \text{concat}(W_1, W_2, 0),\ \text{concat}(b_1, b_2, 0))),\ \text{concat}(f_1, f_2, 1),\ e)$
$\Longleftrightarrow \text{reduce_add}(\text{linear}(\text{gelu}(\text{linear}(x, W_1, b_1)),\ f_1,\ e),\ \text{linear}(\text{gelu}(\text{linear}(x, W_2, b_2)),\ f_2,\ 0))$

(R2) $\text{linear}(\text{attention}_{\text{head}}(q, k, v),\ \text{concat}(w_o^{(1)}, w_o^{(2)}, 1),\ b)$
$\Longleftrightarrow \text{reduce_add}(\text{linear}(\text{attention}_{\text{head}}(q_1, k_{\pi(1)}, v_{\pi(1)}),\ w_o^{(1)},\ b),\ \text{linear}(\text{attention}_{\text{head}}(q_2, k_{\pi(2)}, v_{\pi(2)}),\ w_o^{(2)},\ 0))$

Entangle (Baseline): Manually Defined Rules

(E1) $\text{addmm}(\text{concat}(b_1, b_2, 0),\ x,\ \text{concat}(W_1, W_2, 1),\ \beta,\ \alpha) \Longleftrightarrow \text{concat}(\text{addmm}(b_1, x, W_1, \beta, \alpha),\ \text{addmm}(b_2, x, W_2, \beta, \alpha),\ 1)$

(E2) $\text{gelu}(\text{concat}(t_1, t_2, d)) \Longleftrightarrow \text{concat}(\text{gelu}(t_1),\ \text{gelu}(t_2),\ d)$

(E3) $\text{matmul}(\text{concat}(x_1, x_2, 1),\ \text{concat}(W_1, W_2, 0)) \Longleftrightarrow \text{reduce_add}(\text{matmul}(x_1, W_1),\ \text{matmul}(x_2, W_2))$

(E4) $\text{attention}_{\text{head}}(\text{concat}(q_1, \ldots, q_p, 1),\ \text{concat}(k_1, \ldots, k_m, 1),\ \text{concat}(v_1, \ldots, v_m, 1))$
$\Longleftrightarrow \text{concat}(\text{attention}_{\text{head}}(q_1, k_{\pi(1)}, v_{\pi(1)}),\ \ldots,\ \text{attention}_{\text{head}}(q_p, k_{\pi(p)}, v_{\pi(p)}),\ 1)$

Figure 5: Comparison of tensor-parallel equivalence rules. Top: normalized block-level rules synthesized by Emerge from execution traces. Bottom: manually authored component rules used by Entangle. All rules assume compatible tensor shapes and partition layouts. R2 and E4 additionally assume that query, key, and value tensors are partitioned into disjoint head groups under a fixed query-to-KV head mapping π.

use formal proof whenever the operator semantics are explicit, and fall back to empirical validation for semantically opaque kernels. In our benchmark, we did not observe any empirically validated rule that later led to an incorrect merge or an incorrect equivalence result. However, these empirically validated rules should still be interpreted as testing-backed rather than formally proved.

6.4.2 Comparison with Manually Authored Rules. To assess rule quality, we compare the rules synthesized by Emerge with manually authored rules from Entangle [35] on tensor-parallel equivalence patterns. Although Emerge is designed for cross-framework equivalence, this setting provides a direct way to contrast automatically inferred rules with hand-engineered ones.

Figure 5 shows that Emerge synthesizes rules (R1–R2) that directly capture the semantic decomposition of feed-forward and head-partitioned attention blocks across tensor-parallel partitions. R1–R2 present normalized block-level summaries of the learned rules, which connect operators such as linear, gelu, concat, and reduce_add. Entangle, by contrast, expresses the same transformations through manually authored primitive rules (E1–E4) for affine decomposition, activation distribution, reduction, and attention partitioning. Entangle's primitive rules are more general in that they apply across a wider range of patterns. Emerge can automatically recover the higher-level equivalences needed in practice, without requiring the underlying primitive steps to be enumerated by hand.

7 Discussion

In this section, we discuss the threats to validity and limitations of our approach.

7.1 Threats to Validity

Internal Validity. Our results depend on the correctness and completeness of the captured computation graphs. Emerge reasons about captured inference graphs along the execution paths exercised by the provided inputs, while behavior outside graph capture or unexercised branches remains out of scope. A second threat comes from rule validation. While some synthesized rules are formally proved by SMT, others are accepted through constraint-aware randomized testing. Such empirically validated rules may still be unsound, potentially leading to false equivalence. In addition, the results may also be sensitive to hyper-parameters such as numerical tolerances and validation budgets.

External Validity. Our evaluation focuses on decoder-only transformer-based language models implemented in HuggingFace Transformers and vLLM [14, 39]. We do not evaluate encoder-decoder or multimodal architectures in this paper. The results may not directly generalize to other serving or compilation ecosystems such as TensorRT, XLA, or ONNX Runtime [2, 15, 41], where operator lowering and execution behavior can differ. Although the underlying formulation is not specific to these two frameworks, our current implementation relies on TorchDynamo [25] for computational graph capture, which may not be available or equally effective in other deployment settings.

7.2 Limitations

Coverage and Dynamic Control Flow. The major limitation of our approach is that it only reasons about execution paths exercised by the input set. Because equivalence relations and rewrite rules are synthesized only from observed executions, if a data-dependent branch is never triggered, the corresponding semantics remain unexplored. This limitation is especially pronounced for Mixture-of-Experts architectures [27], where expert routing depends on input values, and for long-context mechanisms such as LongRoPE, which are activated only beyond certain sequence-length thresholds.

Validation Strength. When both sides of a rule consist of symbolically tractable operators, Emerge can discharge equivalence

through SMT solving. However, opaque or fused kernels often lack accessible symbolic semantics and therefore fall back to constraint-aware randomized testing. In such cases, confidence depends on the quality of inferred preconditions and the testing budget, rather than on a formal proof.

Scalability. Repeated graph execution, e-graph rebuilding, and rule enumeration introduce overhead that grows with model size. Although we employ pruning and bounded enumeration, synthesis can still become costly on very large graphs with high branching factors. We have not yet evaluated Emerge at the 200B-parameter scale, and reducing this overhead remains an important direction for future engineering work.

8 Related Work

8.1 Equivalence Checking for DL Systems

The closest line of work studies equivalence checking for distributed training and parallel implementations of deep learning systems [12, 18, 35, 46, 47], which can be broadly grouped into three categories.

Differential Testing and Diagnosis. Testing-based approaches validate correctness by comparing executions against a trusted reference [12]. Rather than comparing only final outputs, TTrace aligns intermediate tensors from a distributed execution with those from a reference implementation and uses tolerance analysis to distinguish bug-induced deviations from numerical error. However, such methods still provide empirical rather than formal evidence of equivalence. They are also sensitive to implementation differences that make numerical alignment difficult, even when the two implementations are semantically equivalent.

Formal Verification of Parallel Training. TrainVerify [18] targets formal verification of distributed LLM training. Given a logical model specification as the ground truth, it verifies that a distributed parallel execution plan is mathematically equivalent to that specification. To scale to frontier models, TrainVerify introduces a stage-wise verification algorithm together with shape-reduction techniques. Compared with our setting, TrainVerify focuses on parallelization equivalence within distributed training, whereas we target cross-framework model implementations whose divergences often involve different operator vocabularies, graph structures, layout transformations, and opaque kernels.

Rewrite-based Equivalence Checking. Recent work [35, 46, 47] has explored the use of equality saturation and e-graphs to check equivalence between computation graphs. Aerify [47], Entangle [35], and Scalify [46] are representative systems in this direction. In particular, Scalify uses bijection inference to reason about layout transformations, while our work also targets equivalences arising from different operator implementations and execution structures.

Compared with these approaches, our work focuses on discovering rules dynamically during verification. This makes the approach better suited to modern model implementations, where the transformations are often framework-specific and difficult to enumerate.

8.2 Testing and Verification for DL Libraries

A broader line of work studies correctness of the deep learning stack through operator verification, differential testing, and fuzzing. Verified lifting techniques recover high-level semantics from low-level deep learning operators and check the recovered summaries against the original implementation [43]. Testing systems such as CRADLE, LEMON, and FreeFuzz detect bugs in deep learning libraries by comparing backend results, mutating models, or generating API-level tests [23, 36, 37]. Compiler-focused fuzzers such as NNSmith and Tzer generate valid models or compiler IR programs to stress deep learning compilers [16, 17]. These techniques are effective for exposing implementation bugs in libraries and compilers. Emerge instead targets equivalence between two concrete LLM implementations and explains mismatches through graph-level relations and synthesized rules.

8.3 Equality Saturation and Rule Synthesis

Our approach builds directly on e-graphs and equality saturation [30], which provide a compact representation of many equivalent expressions and support efficient congruence maintenance during rewriting. Egg [38] introduces an efficient equality-saturation engine together with the rebuild procedure that restores congruence after merges. Our rebuilding procedure is inspired by this design. Our attachment of concrete execution values to e-classes can be viewed as an e-graph analysis in the sense of Egg [38]. Egglog [44] further combines equality saturation with Datalog-style reasoning to improve rule application efficiency. Beyond applying a fixed rewrite set, several prior works study the automatic inference of rewrite rules. Ruler [19] and Enumo [22], for example, synthesize rules from a grammar and an interpreter by enumerating candidate expressions and validating semantic equivalence. TASO [10] and Mirage [40] enumerate graph substitutions and verify their correctness under known operator semantics, but their goal is graph optimization rather than implementation equivalence.

Our work differs in two important ways. First, we synthesize rules in the context of large deep learning computation graphs rather than in a standalone algebraic domain. Second, our synthesis procedure is guided by the shared-frontier structure induced by the current e-graph, which constrains the search space and enables practical rule discovery on real models.

9 Conclusion

In this paper, we presented Emerge, a verification engine for *Implementation Equivalence*. Instead of relying on manually written rewrite rules, Emerge infers candidate relations from execution values, synthesizes rules on demand, validates them, and propagates through a joint e-graph. Our current implementation and evaluation focus on inference computation graphs, and the results show that this execution-driven design is effective in practice: Emerge detects and localizes real implementation bugs, establishes equivalence for correct model pairs at practical cost, and synthesizes rules that compare with manually authored ones.

Acknowledgments

This research is supported by National Key R&D Program of China (No. 2024YFB4506400) and the CCF-Huawei Populus Grove Fund.

Data Availability Statement

Artifacts are available via figshare [42].

References

[1] Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, Ammar Ahmad Awan, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, Harkirat Behl, Alon Benhaim, Misha Bilenko, Johan Bjorck, Sébastien Bubeck, Martin Cai, Qin Cai, Vishrav Chaudhary, Dong Chen, Dongdong Chen, Weizhu Chen, Yen-Chun Chen, Yi-Ling Chen, Hao Cheng, Parul Chopra, Xiyang Dai, Matthew Dixon, Ronen Eldan, Victor Fragoso, Jianfeng Gao, Mei Gao, Min Gao, Amit Garg, Allie Del Giorno, Abhishek Goswami, Suriya Gunasekar, Emman Haider, Junheng Hao, Russell J. Hewett, Wenxiang Hu, Jamie Huynh, Dan Iter, Sam Ade Jacobs, Mojan Javaheripi, Xin Jin, Nikos Karampatziakis, Piero Kauffmann, Mahoud Khademi, Dongwoo Kim, Young Jin Kim, Lev Kurilenko, James R. Lee, Yin Tat Lee, Yuanzhi Li, Yunsheng Li, Chen Liang, Lars Liden, Xihui Lin, Zeqi Lin, Ce Liu, Liyuan Liu, Mengchen Liu, Weishung Liu, Xiaodong Liu, Chong Luo, Piyush Madan, Ali Mahmoudzadeh, David Majercak, Matt Mazzola, Caio César Teodoro Mendes, Arindam Mitra, Hardik Modi, Anh Nguyen, Brandon Norick, Barun Patra, Daniel Perez-Becker, Thomas Portet, Reid Pryzant, Heyang Qin, Marko Radmilac, Liliang Ren, Gustavo de Rosa, Corby Rosset, Sambudha Roy, Olatunji Ruwase, Olli Saarikivi, Amin Saied, Adil Salim, Michael Santacroce, Shital Shah, Ning Shang, Hiteshi Sharma, Yelong Shen, Swadheen Shukla, Xia Song, Masahiro Tanaka, Andrea Tupini, Praneetha Vaddamanu, Chunyu Wang, Guanhua Wang, Lijuan Wang, Shuohang Wang, Xin Wang, Yu Wang, Rachel Ward, Wen Wen, Philipp Witte, Haiping Wu, Xiaoxia Wu, Michael Wyatt, Bin Xiao, Can Xu, Jiahang Xu, Weijian Xu, Jilong Xue, Sonali Yadav, Fan Yang, Jianwei Yang, Yifan Yang, Ziyi Yang, Donghan Yu, Lu Yuan, Chenruidong Zhang, Cyril Zhang, Jianwen Zhang, Li Lyna Zhang, Yi Zhang, Yue Zhang, Yunan Zhang, and Xiren Zhou. 2024. Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone. arXiv:2404.14219 [cs.CL] https://arxiv.org/abs/2404.14219

[2] Shakkeel Ahmed, Prakash Bisht, Ravi Mula, and Soma S Dhavala. 2021. A Deep Learning framework for Interoperable Machine Learning. In *Proceedings of the First International Conference on AI-ML Systems* (Bangalore, India) *(AIMLSystems '21)*. Association for Computing Machinery, New York, NY, USA, Article 23, 7 pages. doi:10.1145/3486001.3486243

[3] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen Technical Report. arXiv:2309.16609 [cs.CL] https://arxiv.org/abs/2309.16609

[4] Stas Bekman. 2022. BLOOM: Megatron-DeepSpeed. https://huggingface.co/blog/bloom-megatron-deepspeed

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 579–594.

[6] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FLASHATTENTION: fast and memory-efficient exact attention with IO-awareness. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1189, 16 pages.

[7] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. doi:10.1007/978-3-540-78800-3_24

[8] David Goldberg. 1991. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.* 23, 1 (March 1991), 5–48. doi:10.1145/103162.103163

[9] Aaron Grattafiori et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[10] 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

[11] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B. arXiv:2310.06825 [cs.CL] https://arxiv.org/abs/2310.06825

[12] Haitian Jiang, Shaowei Zhu, Zhen Zhang, Zhenyu Song, Xinwei Fu, Zhen Jia, Yida Wang, and Jinyang Li. 2026. TTrace: Lightweight Error Checking and Diagnosis for Distributed Training. arXiv:2506.09280 [cs.DC] https://arxiv.org/abs/2506.09280

[13] Yuxuan Jiang, Ziming Zhou, Boyu Xu, Beijie Liu, Runhui Xu, and Peng Huang. 2025. Training with Confidence: Catching Silent Errors in Deep Learning Training with Automated Proactive Checks. arXiv:2506.14813 [cs.LG] https://arxiv.org/abs/2506.14813

[14] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. doi:10.1145/3600006.3613165

[15] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. doi:10.1109/CGO51591.2021.9370308

[16] 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

[17] Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. 2022. Coverage-guided tensor compiler fuzzing with joint IR-pass mutation. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 73 (April 2022), 26 pages. doi:10.1145/3527317

[18] Yunchi Lu, Youshan Miao, Cheng Tan, Peng Huang, Yi Zhu, Xian Zhang, and Fan Yang. 2025. TrainVerify: Equivalence-Based Verification for Distributed LLM Training. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles* (Lotte Hotel World, Seoul, Republic of Korea) *(SOSP '25)*. Association for Computing Machinery, New York, NY, USA, 237–253. doi:10.1145/3731569.3764850

[19] Chandrakana Nandi, Max Willsey, Amy Zhu, Yisu Remy Wang, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. 2021. Rewrite rule inference using equality saturation. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 119 (Oct. 2021), 28 pages. doi:10.1145/3485496

[20] NVIDIA Corporation. 2025. Megatron Core. https://developer.nvidia.com/megatron-core.

[21] NVIDIA Megatron-LM Contributors. 2024. Incorrect loss scaling in context parallel code logic. https://github.com/NVIDIA/Megatron-LM/issues/673. GitHub Issue #673.

[22] Anjali Pal, Brett Saiki, Ryan Tjoa, Cynthia Richey, Amy Zhu, Oliver Flatt, Max Willsey, Zachary Tatlock, and Chandrakana Nandi. 2023. Equality Saturation Theory Exploration à la Carte. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 258 (Oct. 2023), 29 pages. doi:10.1145/3622834

[23] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1027–1038. doi:10.1109/ICSE.2019.00107

[24] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. *OpenAI Blog* (2019). https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

[25] James Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. 2022. torch. fx: Practical program capture and transformation for deep learning in python. *Proceedings of Machine Learning and Systems* 4 (2022), 638–651.

[26] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. FlashAttention-3: fast and accurate attention with asynchrony and low-precision. In *Proceedings of the 38th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '24)*. Curran Associates Inc., Red Hook, NY, USA, Article 2193, 28 pages.

[27] 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

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

[29] Robert Endre Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM* 22, 2 (April 1975), 215–225. doi:10.1145/321879.321884

[30] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. *SIGPLAN Not.* 44, 1 (Jan. 2009), 264–276. doi:10.1145/1594834.1480915

[31] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing

Machinery, New York, NY, USA, 10–19. doi:10.1145/3315508.3329973

[32] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[33] vLLM Contributors. 2023. GPT-J: Incorrect RoPE rotation scheme. https://github.com/vllm-project/vllm/issues/590. GitHub Issue #590.

[34] vLLM Contributors. 2024. Wrong `split_qkv` execution semantics in tensor parallel (InternLM2). https://github.com/vllm-project/vllm/issues/8017. GitHub Issue #8017.

[35] Zhanghan Wang, Ding Ding, Hang Zhu, Haibin Lin, and Aurojit Panda. 2026. It Takes Two to Entangle. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (USA) *(ASPLOS '26)*. Association for Computing Machinery, New York, NY, USA, 1022–1039. doi:10.1145/3779212.3790178

[36] 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

[37] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free lunch for testing: fuzzing deep-learning libraries from open source. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 995–1007. doi:10.1145/3510003.3510041

[38] 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

[39] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Qun Liu and David Schlangen (Eds.). Association for Computational Linguistics, Online, 38–45. doi:10.18653/v1/2020.emnlp-demos.6

[40] Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Man Kit Ao, Praveen Velliengiri, Xupeng Miao, Oded Padon, and Zhihao Jia. 2025. Mirage: a multi-level superoptimizer for tensor programs. In *Proceedings of the 19th USENIX Conference on Operating Systems Design and Implementation* (Boston, MA, USA) *(OSDI '25)*. USENIX Association, USA, Article 13, 18 pages.

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

[42] Qi Zhan. 2026. EMERGE: Checking Implementation Equivalence for LLMs via Rule Synthesis. doi:10.6084/m9.figshare.31852063.v1

[43] Qi Zhan, Xing Hu, Xin Xia, and Shanping Li. 2024. Verified Lifting of Deep Learning Operators. arXiv:2412.20992 [cs.LG] https://arxiv.org/abs/2412.20992

[44] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better Together: Unifying Datalog and Equality Saturation. *Proc. ACM Program. Lang.* 7, PLDI, Article 125 (June 2023), 25 pages. doi:10.1145/3591239

[45] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: generating high-performance tensor programs for deep learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 49, 17 pages.

[46] Kahfi S. Zulkifli, Wenbo Qian, Shaowei Zhu, Yuan Zhou, Zhen Zhang, and Chang Lou. 2025. Verifying Computational Graphs in Production-Grade Distributed Machine Learning Frameworks. arXiv:2509.10694 [cs.LG] https://arxiv.org/abs/2509.10694

[47] Kahfi S. Zulkifli, Wenbo Qian, Shaowei Zhu, Yuan Zhou, Zhen Zhang, and Chang Lou. 2025. Verifying Semantic Equivalence of Large Models with Equality Saturation. In *Proceedings of the 5th Workshop on Machine Learning and Systems* (World Trade Center, Rotterdam, Netherlands) *(EuroMLSys '25)*. Association for Computing Machinery, New York, NY, USA, 82–89. doi:10.1145/3721146.3721943