Lift-and-Verify: Kernel-Transparent Equivalence-Based Verification in LLM Compilation

João Paulo Cardoso de Lima
joao.lima@tu-dresden.de
Technische Universität Dresden
Dresden, Germany

Anderson Faustino da Silva
afsilva@uem.br
Universidade Estadual de Maringá
Maringá, Brazil

Guilherme de Oliveira Silva
guilherme.olivsilva01@gmail.com
Technische Universität Dresden
Dresden, Germany

Jeronimo Castrillon
jeronimo.castrillon@tu-dresden.de
Technische Universität Dresden
Dresden, Germany

Abstract

Large language model (LLM) compilation and serving stacks rely on aggressive graph optimizations, distributed parallelization, and hardware-specialized custom kernels. These optimizations cause low-level execution graphs to diverge sharply in operator decomposition, tensor layouts, and device partitioning, often introducing silent miscompilation bugs. Existing equivalence verifiers fail to scale because graph-level equality saturation suffers from e-graph rule explosion, while custom vendor kernels remain opaque black boxes. We present Lift-and-Verify (LnV), a kernel-transparent, hierarchical verification framework that enables scalable semantic equivalence checking across LLM compilation stacks. LnV's key insight is that raising both execution graphs and vendor kernels to a single canonical tensor abstraction, `einsum+`, normalizes away framework-specific divergences, so that canonicalization discharges most equivalence checks and equality saturation is needed only for the rest. By fully integrating low-level kernels into a hierarchical verification pipeline, LnV achieves scalable, end-to-end equivalence checking without e-graph rule explosion.

1 Introduction

Deep learning models are deployed across a diverse ecosystem of tensor frameworks [23], compiler stacks [4, 5, 19], serving systems [34], and heterogeneous accelerators [10]. The same model is often realized by multiple implementations, each specialized to a serving system and hardware target [5]. Even when they originate from one high-level program (e.g., PyTorch [23, 38]) and share a lowering path (e.g., an MLIR middle-end [4, 15, 19]), the resulting implementations might diverge sharply in how the linear algebra is decomposed [5, 39], how tensors are laid out in memory [5], and how the computation is partitioned across devices [29]. Such divergences can lead to errors that go unnoticed during the costly LLM training [18], creating the need for equivalence-based verification systems [27, 40, 41].

Implementation inequivalences are a growing source of silent bugs. For instance, production stacks replace the reference $\texttt{softmax}(QK^T/\sqrt{d}\cdot V)$ with hand-optimized fused kernels, such as FlashAttention and Triton variants, which a graph-level checker sees as a single opaque node. A misplaced $\sqrt{d}$ or a numerical-precision mismatch silently corrupts the output.[1] Other bugs can appear due to layout and permutation divergence: two rotary-embedding (RoPE) implementations that agree on every tensor shape and operator name still compute different tensors when they interleave channels under different conventions.[2] Additional bugs in production have also arisen from making a sequential model into a distributed one [29]. As LLMs increasingly generate kernels [16, 22, 35], the rate at which such divergences enter production will rise, and the functional equivalence with respect to the reference model description becomes a first-order reliability problem.

Equality saturation [27] is a natural engine for addressing this problem. It represents all rewrites of both graphs (i.e., reference and optimized) in one equivalence graph (e-graph) [31]. When both graph outputs land in the same equivalence class (e-class), that is the proof of equivalence. Recent checkers have built on it: Entangle [29] refines a distributed model back to its sequential reference form; Scalify [40] adds partitioning, layer memoization, relational reasoning, and symbolic bijection inference. However e-graphs grows explosively as rules compound, which is the barrier to models with hundreds of layers. More fundamentally, these systems reason at a single, fixed abstraction (typically HLO [8]), where a difference in operator decomposition or layout has no compact rewrite rules and depends on framework-specific, hand-written rules.

Crossing abstractions is harder still. Modern frameworks lean heavily on custom kernels, such as C++/CUDA fused kernels or hardware-specific primitives [6, 28], whose semantics are not exposed to the computation-graph level. Even where semantics are visible, two implementations may diverge in operator decompositions and tensor layouts. For example, GPT-2's linear projection [25] uses `addmm`(w1) in HuggingFace's Transformer [32] and `linear`(w2) in vLLM [14], with transposed weight tensor and the surrounding execution plan organized differently. These challenges, namely, *opaque semantics* [24] and *divergent path* [29, 39], are symptoms of the same root cause: the semantics needed for equivalence checking are only available below the graph-level abstraction. A verifier that stays at graph-level can only match operators synthetically, so it needs a hand-written rule for every kernel and every decomposition, and does not generalize across frameworks. Prior verifiers [29, 40, 41] therefore have left framework-specific kernels as opaque operators that a graph-level verifier cannot see through. Emerge [36], by contrast, discovers

CHASM 2026, Athens, Greece
2026. ACM ISBN 978-X-XXXX-XXXX-X/XX/XX
https://doi.org/XXXXXXX.XXXXXXX

[1] https://github.com/triton-lang/triton/issues/1671
[2] https://github.com/huggingface/transformers/issues/39687

relations of custom kernels from executions and synthesizes rules on demand, hence inferring w1 ≡ `transpose`(w2) without ever making the kernel transparent.

Our *observation* is that rewrite rules fail to transfer across frameworks because verification happens at too low an abstraction, the level at which implementation is divergent by design. If we lift both implementations to a common, higher, canonical tensor abstraction, those divergences are abstracted away. With that, `addmm` and `linear` become the same tensor contraction and a transposed weight becomes an index relabeling, a sharded `matmul` plus `all-reduce` becomes an ordinary reduction, and so on. At that level, a small, fixed, framework-agnostic rule-set is enough and most of the equivalence verification is discharged by canonicalization rather than search.

We present Lift-and-Verify, LnV for short, a hierarchical, lifting-based methodology for verifying tensor-program equivalences across frameworks. Concretely, our contributions are:

- A **semantic-preserving lifting** method (§ 3) that raises medium/low-level kernel code (e.g., C++) to a high-level canonical tensor abstraction while preserving silent bugs, hence making opaque kernels transparent to the verifier.
- A **hierarchical verification methodology** (§ 4) that decomposes an equivalence query and routes each sub-difference to the abstraction level where it is soundly and cheaply resolved, using canonical fingerprints to prune subgraphs already proven equal.

2 Preliminaries

2.1 Distributed Tensor Algebra

Following the tensor-algebra formulation used in tensor compilers [11], we treat tensors together with the operations of product, addition, and contraction. Of these, contraction is the operation that reduces rank by summing a shared index between two tensors, or within one:

$$C_{i_1,\ldots,i_k} = \sum_{j_1,\ldots,j_m} A_{i_1,\ldots,i_p,j_1,\ldots,j_m} \cdot B_{j_1,\ldots,j_m,i_{p+1},\ldots,i_k} \tag{1}$$

Matrix multiplication, dot products, and trace, are all instances of this rule with different choices of k, m, and p.

Einstein summation notation (`einsum`) is the syntax for specifying contractions popularized by array libraries, such as NumPy and JAX [9]. Its general form is $\texttt{einsum}(I_1, \ldots, I_n \rightarrow O, T_1, \ldots, T_n)$, where each T_i is an input tensor with multi-index I_i, O is the output multi-index, and the multi-indices are composed of tensor dimensions whose count gives the tensor's order. An index letter repeated across operands and absent from the output implies a contraction. When a repeated index is instead kept in O, it represents broadcast as a batch axis (e.g., b in `bij,bjk` → `bik`). `einsum` thus expresses the class of tensor operations built from contraction and batching, which constitutes the dominant linear-algebra pattern in DL computation. However, it does not cover decomposition/inversion operations or the non-algebraic operations that motivate our extension (§ 4.2).

Distributed execution partitions this algebra across a physical mesh of devices. Modern parallel training and inference paradigms, such Megatron-style Tensor Parallelism [26], Fully Sharded Data Parallel [38], and Sequence Parallelism [12], decompose a high-dimensional contraction into sharded sub-contractions. We write a device mesh of shape $r_1 \times \cdots \times r_d$ as M, and a sharding specification that assigns a tensor's output axes to mesh dimensions as a postfix `shard=`*axes*`@`M annotation. For example, `shard=ik@2x4` marks axes i, k of an einsum's output as partitioned across a 2×4 mesh. Splitting a *contracted* axis (e.g., j in `ij,jk` → `ik`) across M leaves each device holding only a partial sum, so restoring the value in (1) requires a collective operation (i.e., `all-reduce`) over that mesh dimension; splitting a *non-contracting* axis instead requires `all-gather` or `reduce-scatter` only when a shard must be later materialized or redistributed across ranks.

2.2 Tensor Program Lifting

Lifting is the process of translating lower-level or framework-specific program representations (e.g., PyTorch computations or imperative C++/CUDA code) into high-level mathematical execution graphs or abstract tensor algebra primitives [3]. In the context of tensor compilers, lifting paradigms fall into two categories: (i) verified lifting uses inductive program synthesis, constraint solving, and interactive theorem provers alongside a formal proof of semantic preservation [3, 24], and (ii) neural and heuristic lifting leverages deep learning models or structural pattern-matching heuristics [4, 7, 17]. By raising framework-specific kernels to a target-agnostic tensor abstraction, lifting lets us reuse the compiler optimizations already available at that higher abstraction [24]. We leverage this idea to simplify kernel code so it can be included in the graph-level verification, stripping away vendor runtime calls, explicit memory layouts, and imperative loop structures in the process.

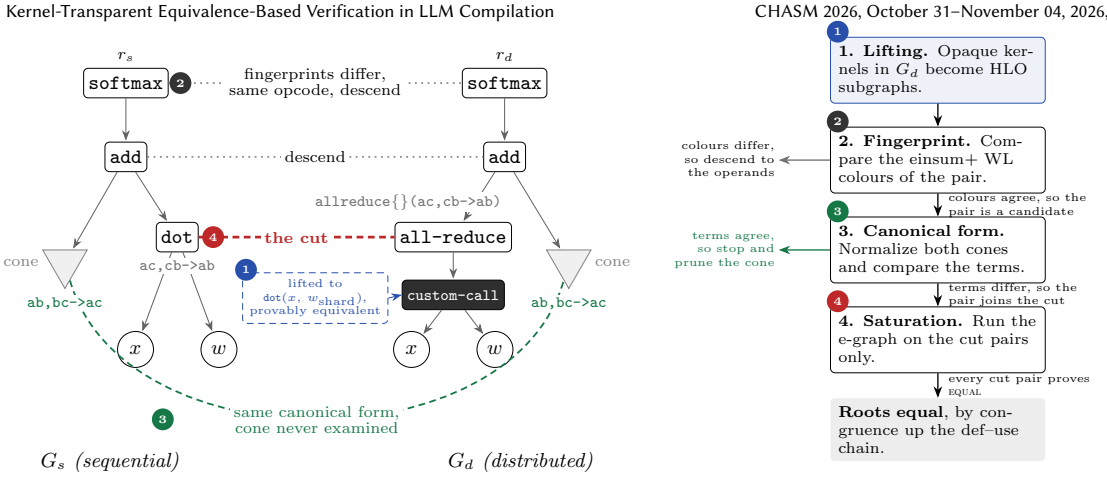
2.3 Equality Saturation

Equality saturation [20, 27] is a non-destructive term rewriting technique that represents all equivalent expressions explored during optimization within a compact data structure called an equivalence graph (e-graph). An e-graph $G = (E, N)$ consists of a set of equivalence classes (e-classes) E, where each e-class $e \in E$ contains semantically equivalent e-nodes $n \in N$. An e-node represents an operator applied to child e-classes.

Verification via equality saturation operates by inserting two intermediate representation (IR) expressions, i.e., a reference sequential graph G_s and an optimized distributed graph G_d, into the e-graph [40]. A set of directed rewrite rules $l \rightarrow r$ is applied saturatedly until a fixed point is reached or a resource limit is exceeded. Functional equivalence between G_s and G_d is formally proven if the root nodes of both programs are merged into the same e-class ($e_{G_s} = e_{G_d}$).

2.4 Overview of Lift-and-Verify

LnV decides whether a distributed, optimized model G_d computes the same function as the sequential model G_s, using the running example in Fig. 1 for illustration. Since full equality saturation does not scale, and compared graphs are usually mostly similar (differing only where a compiler pass rewrote one region), LnV cheaply identifies identical parts and reserves expensive e-graph reasoning for the rest. The approach proceeds hierarchically through four tiers, as shown in Fig. 1b: Tier 1 lifts custom kernels from C++ to HLO to make them transparent; Tier 2 canonicalizes both graphs into `einsum+` form and reduces subgraphs to fingerprints for candidate matching; Tier 3 verifies candidates by canonicalizing and comparing cones,



(a) lifting and lockstep walking both graphs **(b) tiers of Lift-and-Verify**

Figure 1: Overview of Lift-and-Verify hierarchical verification approach.

pruning proven pairs; and Tier 4 runs equality saturation only on the remaining unproven cut, with the roots' equivalence following by congruence.

3 Semantic-Preserving Lifting

To cross the abstraction barrier between opaque, framework-specific kernels, and transparent, verifiable code, the Tier 1 of LnV lifts the kernels' implementations to obtain one from another. The *correct-by-construction* lifting schema generates functional tensor subgraphs that are semantically equivalent to the imperative kernel code, and replaces the latter with the former without relying on hand-written, framework-specific rules or synthetic operator matching.

3.1 Problem Formulation and Bug-Preserving Guarantees

Given an imperative kernel implementation K_{code} operating on concrete memory buffers across an input domain $\mathcal{D}$, the goal of *correct-by-construction* lifting is to synthesize a functional tensor expression T_{tensor} within a high-level tensor algebra that precisely models the input-output behavior of K_{code}. Formally, lifting establishes functional algebraic equivalence between $\mathcal{E}_{\text{imperative}}(K_{\text{code}}, \mathbf{X})$ and $\mathcal{E}_{\text{functional}}(T_{\text{tensor}}, \mathbf{X})$, where $\equiv_\eta$ denotes equivalence under floating-point precision tolerance η during numeric witness refutation (§ 4.5).

Crucially, unlike standard optimizing compilers that assume bug-free reference implementations, a verification-oriented lifter must be *bug-preserving*. If K_{code} contains a subtle implementation divergence, such as an inverted channel permutation, a misplaced normalization constant, or an incorrect reduction index, the synthesized term T_{tensor} must faithfully capture that exact defect rather than repairing it into a correct reference form. Preserving these defects at the tensor abstraction layer ensures that downstream canonicalization and e-graph equivalence checking reliably catch silent miscompilations.

3.2 Verification-Oriented Synthesis Pipeline

The translation from imperative code (e.g., C++/CUDA or Triton ASTs) to transparent functional subgraphs proceeds through a three-stage synthesis pipeline. It leverages Polygeist [19] for stage 1, and Tensorize [3] for stages 2 and 3.

1. From C++/CUDA to structured MLIR. The lifter first translate the imperative implementation into structured MLIR, using the `arith`, `memref`, and `scf` dialects. CUDA directives are raised and expressed with the gpu dialect. It then analyzes the structure of the loops across the program and check whether they are well formed and strictly linear. If those requirements are met, `scf` loops are further raised to the `affine` dialect.

2. Specification Extraction and Sketch Generation. The lifter executes the structured MLIR symbolically, rather than concretely, to track the implicit mapping between input and output arrays. This allows it to extract the algebraic specification of the input program, which serves as the basis for generating symbolic sketches. Sketches are partial programs with placeholder inputs that can be recursively replaced with other sketches to compose expressions and symbolically executed. They are expressed in the same DSL as the target program (i.e., HLO), enabling step-wise simplifications.

3. Program Synthesis. The functional representation T_{tensor} of the kernel K_{code} is synthesized using algebraic solving to select sketches that compose an expression that is equivalent to the specification. This process is conducted in top-down manner, by choosing the partial programs that simplify the specification the most. It is also recursive, as the simplified sketch is then used as the basis for the next synthesis step. This continues until a complete equivalent program is generated, which is correct by construction.

3.3 Kernel Substitution and Graph Integration

Once the functional term T_{tensor} is correctly synthesized, Tier 1 performs in-place graph substitution on the distributed execution graph G_d. The opaque `custom-call` instruction in G_d is spliced out, and the lifted HLO subgraph T_{tensor} is inserted into its position, re-linking parameter input edges and consumer def-use chains.

This substitution transforms G_d from a partially opaque graph containing black-box kernel calls into a completely transparent computation DAG. By raising imperative kernel code to target-agnostic functional tensor operations, Tier 1 eliminates framework-specific divergences at the lowest abstraction layer. The fully transparent graph is then handed directly to Tier 2.

4 Hierarchical Verifier

After Tier 1, the distributed graph G_d contains no opaque nodes, so both inputs to the verifier are transparent HLO modules. This section describes what happens next.

4.1 Graph Foundations and Cut Formalism

An HLO module is a DAG of instructions. Every instruction carries an opcode (`dot`, `reshape`, `add`, and so on), an output shape, and an ordered list of operands, that is, the instructions that produce its inputs. Edges follow def–use, so an instruction points at the instructions it consumes. Recall, we first want to identify the identical parts of a graph in a cheaper manner before handing it to equality saturation. Two notions make this possible. The *cone* of an instruction is the set of instructions that feed it, obtained by following operand edges down to the parameters. The *cut* is the frontier of instruction pairs where the cheap comparison stops succeeding, so everything above the cut is already matched and everything below it still needs a decision.

4.2 High-Level Representation via einsum+

Two implementations of the same computation rarely produce the same HLO. A linear projection with a bias may appear as a `dot` followed by a broadcast and an `add`, as a single fused `addmm`, or as a vendor kernel. Each of these is a structurally different DAG, so a verifier that compares HLO directly needs one rewrite rule per divergence, and the rule set grows with every framework and every compiler pass it must tolerate [29, 39, 40].

We mitigate this growth by mapping both graphs to a shared abstraction in which such divergences cannot be represented, namely `einsum+`, a canonical textual IR for tensor subgraphs and a strict superset of `einsum`. `einsum+` encodes a tensor subgraph as a single whitespace-free string, referred to as the *signature* of that subgraph. Standard `einsum` already supports tensor contractions, elementwise multiplication, and batch axes, which suffices to represent the linear computational core of a neural network but does not capture operations beyond this scope. `einsum+` adds three constructs on top of it:

(1) **Wrapped operands.** An operation that changes shape or is not multilinear wraps the axes it consumes, as in `relu{}(ij)` for an elementwise activation or `maxpool{k=2}(ij->im)` for a windowed reduction. The internal arrow records how local axes are transformed.
(2) **Post-operation pipes.** Operations applied in sequence to the result of the contraction are appended after a pipe, as in `ab,bc->ac|relu{}`. Pipes keep chained elementwise work out of the contraction, where it would otherwise force a separate subgraph.
(3) **Attribute blocks.** Static hyperparameters are represented by `{key=value}` blocks, so a reduction axis, a stride, or a numeric constant cannot be confused with tensor structure.

Operands may carry an element type, written `[bf16]ab`, and an output whose extents are not known statically is marked with `?`. The projection in our running example raises to:

```
ab,bc->ac|add{}|softmax{axis=c}
```

whether it was emitted as three HLO instructions or as one fused instruction, because the notation records which axes are contracted rather than which instructions were used to contract them.

Raising HLO. A signature describes one subgraph producing one output, so a HLO module, which has many outputs and constructs that `einsum+` does not model, cannot be raised as a whole. Raising a module requires splitting it into subgraphs of the right shape, which we call *regions*, and each region is then rendered as one signature. The split cuts at side-effecting instructions, e.g., host I/O, control flow (`while`, `conditional`, and asynchronous start and done pairs), since ordering constraints and trip count are not expressible in a signature. Distributed collectives are not cut, since §4.3.1 relies on them appearing inside a region. What remains inside a region is one contraction together with the chain of elementwise operations applied to its result, which is the form `ab,bc->ac|relu{}` expresses.

4.3 Weisfeiler-Leman Term Canonicalization

Raising to `einsum+` alone does not make two equivalent subgraphs textually equal, because axis letters and operand order are still arbitrary. Canonicalizing them is circular problem that a naive scanning on graphs with structural symmetries produces different strings for equivalent subgraphs [5]. We avoid this circularity by employing Weisfeiler–Leman color refinement [30] applied to the dimensions of a signature.

To do so, each dimension starts with a color built from facts that do not depend on writing order, namely its extent, its role as contracted, batch, or free, and the operations touching it. Each refinement round then replaces the color of a dimension with the pair formed by its current color and the sorted list of the colors of the dimensions it appears beside, whether in an operand or in the output. Refinement stops at the fixed point where color classes no longer split, and letters are assigned to the dimensions by color class. Only then are operands and commuting pipes sorted by their rendered strings.

Refinement can stop with several dimensions still sharing a color, either because the expression is symmetric, as in `ij,ji->`, or because refinement is an incomplete test for symmetry [33]. We break such ties in two ways and keep both results, as a *fingerprint* that takes the source order, which costs nothing and is not canonical, and as a *canonical form* that keeps whichever option gives the alphabetically smallest string, which no longer depends on the input order.

In summary, equal fingerprints alone do not imply equivalence, so a fingerprint match is only a candidate that the canonical form has to confirm. Equal canonical forms imply that the two subgraphs compute the same function, which is used to prune search. Equivalences that need search, such as reassociating a chain of contractions, are out of the canonalization scope, so they are left to § 4.5.

4.3.1 Distributed Collectives. Distributed graphs carry communication primitives (§ 2.1) that move and recombine shards of a tensor without changing its mathematical value. We therefore raise every collective to `einsum+` as an explicit wrapped operand, for example `all_reduce{op=sum,replica_groups=[[0,1]]}(ac)->ac`, which records the reduction operator and which devices combine. Canonicalization renames and orders these operands like any other, but the collective itself is not touched. The equality is established in § 4.5, where an `all-reduce` over a per-device fragment is identified

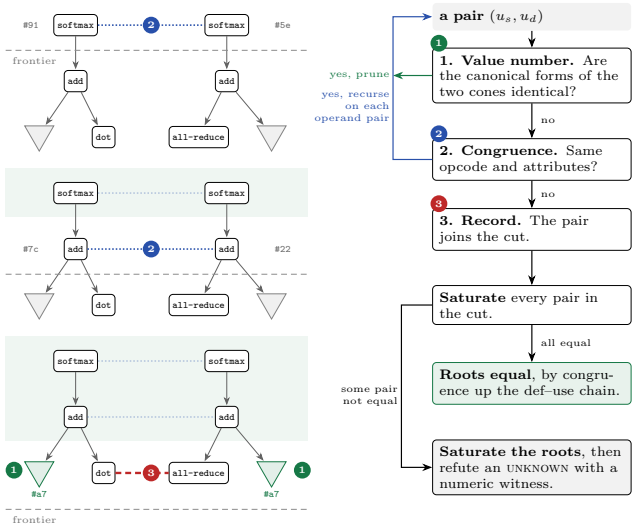


Figure 2: Left: (a) the walk descending to the cut, top to bottom. Right: (b) the decision taken at a single pair.

with the corresponding reduction, guarded by the condition that its operand really is a fragment along the mesh axis being reduced.

4.4 Lockstep Traversal and Frontier Isolation

The HLO-to-`einsum+` raising gives each instruction a term denoting its entire cone, written over the module parameters, together with its opcode, attributes, and def–use edges. Canonicalizing that term and serializing it yields one string per instruction, which we now use as a value number. Two proof rules apply to a pair of instructions: value numbering and congruence. Equal value numbers mean the two cones compute the same value, which is global value numbering [1] over a canonicalized term, so two cones that differ only in index names or in the order of commuting operands receive the same number. Congruence means that two instructions with the same opcode and attributes whose operands are already known equal pairwise produce equal results [21].

LNV pairs root r_s with r_d and walks both DAGs downward, pairing operands by position, as illustrated in Fig. 2. As shown in Fig. 2b, each pair (u_s, u_d) is resolved by the first rule that applies:

(1) **Value numbering.** The two value numbers are equal. The cones are equal, so stop without descending.
(2) **Congruence.** The opcodes and attributes match. Recurse on every operand pair. The pair is equal once all of them resolve as equal.
(3) **Record.** Otherwise record the pair and stop.

On the running example of Fig. 2a the walk resolves four pairs. At the roots the two value numbers differ but the opcodes match, so rule 2 descends. The same holds one level down. At the third step the two cones match by rule 1 and are never examined, while the pair that neither rule settles becomes the cut. The dashed line is the frontier, and the shaded region is the region already resolved.

Rule 1 compares whole cones and fails on any difference canonicalization does not absorb, however deep. Rule 2 compares one instruction and delegates the rest to the recursion, which is what carries the frontier down to the pairs responsible for the difference. Without it, two graphs that differ anywhere would put their roots in the cut. The pairs recorded by rule 3 form the *cut*. Decomposing an equivalence query at a frontier of internal points is the standard structure of cut-based equivalence checking [13], which we apply to tensor graphs rather than circuits. Rule 2 only ever asks that operands be equal, so proving the cut pairs equal makes every pair above them equal as well, up to the roots. This is the congruence needed to make skipping the rest sound. Because `einsum+` canonicalization canonicalizes commuting operands into a deterministic lexicographical order (§ 4.3), operand pairing by position remains sound even when compiler passes permute commutative arguments.

The cut has two purposes. First, the cut localizes the difference, namely the instructions at which the graphs stop agreeing, which is reported when we cannot prove equivalence. Second, a cut bouds the work sent to saturation, since its size depends on how much of the graph a compiler pass previously rewrote rather than from how large the graph is. If every cut pair proves equal, the roots follow immediately. If any pair does not, then LNV discards the localization and issues a fresh query at the roots, because a difference at an intermediate node does not imply one at the output.

4.5 Localized Cut Saturation

Equality saturation (§ 2.3) represents many rewritings of a term in one equivalence graph and reports equivalence when two terms land in the same equivalence class. LNV runs it on the cut pairs only. Each pair enters as the two canonicalized terms of its cones, so the initial e-graph holds one e-node per distinct normalized subterm rather than one per syntactic variant, and saturation starts from a state that canonicalization has already reduced. Crucially, a cut pair (u_s, u_d) is not inserted into the e-graph as isolated opcodes. Instead, each node enters as the fully expanded, canonicalized `einsum+` term representing its entire computation cone down to parameter leaves. This enables egglog rules [31] to reason globally about sharding annotations and operator contractions across the localized frontier.

Not all of LNV's rules are rewrites. Some rules instead derive *facts* rather than equalities, such as whether an index occurs free (fv) in a term and whether a tensor is a per-device fragment along a mesh axis, and the remaining rules apply only when those facts hold. For that reason, we use egglog [37], which combines Datalog-style fact derivation with equality saturation. Two of our four rule groups derive facts that the others depend on, and those facts must be maintained as e-classes merge rather than checked once. The rule set contains 34 rules in four groups covering multilinear, additive, uninterpreted, and collective operators. Two of them carry most of the weight. Pushing a contraction through a product,

$$\sum_i (x \cdot y) \;\equiv\; x \cdot \sum_i y \qquad \text{when } i \notin \mathrm{fv}(x),$$

makes the contraction path and the choice of intermediates irrelevant, so $(AB)C$ and $A(BC)$ meet without a rule naming either. Identifying a collective with a reduction, $\mathrm{allreduce}_p(x) \equiv \sum_p x$ when x is a per-device fragment along p, discharges the cut pair of Fig. 2a. The condition is essential, since an `all-reduce` applied to a tensor that is not a fragment is not a reduction, and applying the rule anyway would prove a distributed graph correct that is not. If these rules were written at HLO instead, each rule would need one variant per decomposition and per layout a framework happens to emit, which is the rule growth that `einsum+` IR is meant to avoid. For scale, XLA's algebraic simplifier alone comprises 175 such rules [2].

Table 1: Architectural comparison of LnV against state-of-the-art equivalence verifiers and tensor compilation tools. Abbrev.: Can.=canonical; FT=fully transparent; OP=opaque; BB=black-box; EqSat=equality saturation; Hier.=hierarchical; CutSat=cut saturation; Nat.=native; Lim.=limited; Y/N/P=yes/no/partial.

System	Target IR	Kernel Transparency	Verification Engine	E-graph explosion mitigation	Distributed Parallelism	Bug Preservation
LnV	**Can. einsum+ IR**	**FT (lifting)**	**Hier. (WL+GVN+EqSat)**	**High (cut-based; 34r)**	**Nat. (TP/FSDP/SP)**	**Y**
Entangle [29]	HLO graph	OP (BB)	EqSat (`egg`)	Low (rule expl.)	Nat. (Dist→Seq)	N
Scalify [40]	HLO graph	OP (BB)	EqSat (`egglog`)	Med. (part./memo.)	Nat. (rel. rules)	N
TrainVerify [18]	Symbolic DFG	FT (hand-rewritten)	SMT + shape red.	N/A	Nat.	P
Emerge [36]	PyTorch FX graph	BB (dyn. discovery)	SMT + EqSat + dyn. rules	Med. (on-demand)	Lim. (1 node)	P
Polyjuice [39]	HLO/ONNX	OP (BB)	EqSat (`egg`)	Low (comb. exp.)	N/A	N/A
TensorRight [2]	TensorRight DSL	N/A	Unbdd. SMT (agg. axis)	N/A	—	N/A
Tenspiler [24]	C/C++→TensIR	N/A	Prog. synth. + SMT	N/A	—	P
Tensorize [3]	C/C++→PyTorch	N/A	Prog. synth. + SMT	N/A	—	P

We then run the rules to saturation, and ask whether the two roots share an e-class. Proving every cut pair equal proves the roots equal, since each pair above the cut has already matched one instruction at a time, so agreement propagates upward and no instruction above the cut enters the e-graph. The reverse does not hold. A cut pair that is not equal leaves the roots undecided, because a difference in the middle can disappear later, e.g., when two intermediates differing only in sign are both squared. The verifier therefore discards the cut in that case and asks the original question about the roots (Fig. 2b).

Saturation always reaches a fixpoint, but a fixpoint without a merge is not always a disproof. The verdict is therefore three-valued. It is EQUAL when the classes meet, NOT_EQUAL when both terms lie in the complete multilinear fragment and the classes do not meet, and UNKNOWN otherwise. We resolve an UNKNOWN by searching for an input on which the two programs disagree. A counterexample turns UNKNOWN into NOT_EQUAL and never into EQUAL, since agreeing on sampled inputs is not a proof.

5 Preliminary Results

5.1 Architectural Comparison

Tab. 1 summarizes the architectural capabilities of LnV in comparison with state-of-the-art equivalence verifiers and tensor compilation tools across six critical design dimensions.

Existing graph-level verifiers, such as Entangle [29], Scalify [40], and PolyJuice [39], operate directly on fixed IRs (e.g., HLO or TorchFX). As a result, they treat custom vendor kernels as opaque black boxes and suffer from severe e-graph rule explosion when encountering framework-specific operator and layout divergences. While framework-level systems like TrainVerify [18] and Emerge [36] attempt to handle dynamic or high-level operators via SMT reduction or execution tracing, they either lack full kernel transparency or rely on unverified runtime heuristics that yield only partial bug preservation. Conversely, program synthesis frameworks such as Tenspiler [24] and Tensorize [3] focus on raising code for compilation rather than bug-preserving verification.

In contrast, LnV uniquely combines *full kernel transparency* with *bug preservation* by lifting low-level imperative code into a canonical `einsum+` representation. By routing equivalence queries through a hierarchical pipeline, LnV achieves high mitigation against e-graph explosion, requiring a compact set of only 34 framework-agnostic rewrite rules. Furthermore, LnV natively supports distributed parallelism paradigms, establishing a scalable and sound end-to-end verification pipeline.

5.2 LnV & Emerge

We compared LnV against the state-of-the-art Emerge [36] on its 19 cases. Three contrasts stand out.

Proof strength. Of Emerge's 199 synthesized equivalence rules, only 110 (55%) are SMT-verified. The remaining 89 rest on randomized testing alone, i.e., rules synthesized on demand. Every EQUAL verdict LnV returns is a sound equality-saturation proof, with no randomized testing involved. Every NOT_EQUAL is either a complete-fragment disproof or a concrete numeric counterexample.

Saturation and pruning. Emerge performs no fixpoint saturation. As rules are synthesized on demand, it has no e-graph growth to control. Even so, its `gpt-2` case still materializes 585 e-classes and 1293 e-nodes and runs in 8.2 s, and its `internlm2` tensor-parallel case takes 889 s, dominated by synthesis time. Our hierarchical verifier instead saturates a fixed rule set, but fingerprint-prunes structurally identical subgraphs to zero e-nodes, reserving saturation for the divergence frontier.

Soundness. Emerge merges e-classes by comparing executed values at a specified tolerance, e.g. 1e-2. Under the b = h axis collision, a transposed, non-equivalent tensor keeps its shape and can pass this check. On the same constructed pair, our symbolic canonicalization correctly returns NOT_EQUAL. On the matched cases we can lift, our checker attains 100% proof completion and 100% bug detection, with per-case latency in milliseconds.

6 Research Agenda

LnV rests on a claim we can verify in our experiments but not settle, i.e., that cross-framework divergence normalizes at a sufficiently high tensor abstraction. The claim invites falsification, and establishing its limits would be progress regardless of which way it falls.

Three questions strike us as open. First, *is there a canonical level at all?* `einsum+` normalizes layout, operator decomposition, and sharding, but this is not an exhaustive list. A characterization of which classes of divergence admit canonicalization, and which are provably irreducible, would tell every verifier in this space how high it is worth lifting. Second, *what does an equivalence claim mean numerically?* Most verifiers in this literature establish equivalence only up to a tolerance η that is chosen by hand; no one can yet

compose per-rewrite error bounds into a statement that survives a hundred layers, and until that exists, "equivalent within a given tolerance ϵ" is a convention rather than a guarantee. Third, *who verifies the lifter?* A lifted specification is itself a model of the kernel, and a lifter that silently repairs the defect it was meant to expose defeats the whole pipeline.

Progress here is limited less by algorithms than by absent shared infrastructure. Each system is evaluated on a bug set of its authors' own construction, so results are not comparable and pass criteria diverge in ways that are sometimes accidental rather than principled. We see three concrete community tasks. A corpus of *silent* implementation defects would do for this area what fuzzing corpora did for compiler testing. An interchange contract for lifted kernels would let a lifter from one group feed a verifier from another, which is currently impossible. Finally, the field needs a reporting convention for *proof strength*. Current systems mix rules discharged by a solver with rules generated on randomized testing and report a single "verified," which makes soundness claims not comparable across papers.

References

[1] Bowen Alpern, Mark N Wegman, and F Kenneth Zadeck. 1988. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1–11.

[2] Jai Arora, Sirui Lu, Devansh Jain, Tianfan Xu, Farzin Houshmand, Phitchaya Mangpo Phothilimthana, Mohsen Lesani, Praveen Narayanan, Karthik Srinivasa Murthy, Rastislav Bodik, et al. 2025. Tensorright: Automated verification of tensor graph rewrites. *Proceedings of the ACM on Programming Languages* 9, POPL (2025), 832–863.

[3] Alexander Brauckmann, Luc Jaulmes, José W. de Souza Magalhães, Elizabeth Polgreen, and Michael F. P. O'Boyle. 2025. Tensorize: Fast Synthesis of Tensor Programs from Legacy Code using Symbolic Tracing, Sketching and Solving. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization* (Las Vegas, NV, USA) *(CGO '25)*. Association for Computing Machinery, New York, NY, USA, 15–30. https://doi.org/10.1145/3696443.3708956

[4] Lorenzo Chelini, Andi Drebes, Oleksandr Zinenko, Albert Cohen, Nicolas Vasilache, Tobias Grosser, and Henk Corporaal. 2021. Progressive raising in multi-level ir. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, IEEE, Virtual Event, 15–26.

[5] Zhaodong Chen, Andrew Kerr, Richard Cai, Jack Kosaian, Haicheng Wu, Yufei Ding, and Yuan Xie. 2024. Evt: Accelerating deep learning training with epilogue visitor tree. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ACM, New York, NY, USA, 301–316.

[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 *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 35. Curran Associates, Inc., N/A, 16344–16359.

[7] José Wesley de Souza Magalhães, Jackson Woodruff, Jordi Armengol-Estapé, Alexander Brauckmann, Luc Jaulmes, Elizabeth Polgreen, and Michael FP O'Boyle. 2025. Guess, Measure & Edit: Using Lowering to Lift Tensor Code. In *2025 34th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, IEEE, Virtual Event, 216–228.

[8] Google Developers. 2017. XLA: Accelerated Linear Algebra Compiler. https://www.tensorflow.org/xla. Google OpenXLA Compiler Infrastructure.

[9] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. 2020. Array programming with NumPy. *nature* 585, 7825 (2020), 357–362.

[10] Norman P Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvin Subramanian, Andy Swing, Brian Towles, et al. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning Training. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, New York, NY, USA, 1–14.

[11] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.

[12] 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), 341–353.

[13] Andreas Kuehlmann and Florian Krohm. 1997. Equivalence checking using cuts and heaps. In *Proceedings of the 34th Annual Design Automation Conference* (Anaheim, California, USA) *(DAC '97)*. Association for Computing Machinery, New York, NY, USA, 263–268. https://doi.org/10.1145/266021.266090

[14] Woosuk Kwon. 2025. *vLLM: an efficient inference engine for large language models*. University of California, Berkeley, Berkeley, CA.

[15] Chris Lattner, Mehdi Amini, Uzi Bickelson, Nicolas Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Tian, and Nicolas Zheng. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. *IEEE Micro* 41, 3 (2021), 29–39.

[16] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Tompkins, James White, Vincent Zhao, Aja Huang, et al. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097.

[17] Yixuan Li, José Wesley de Souza Magalhães, Alexander Brauckmann, Michael FP O'Boyle, and Elizabeth Polgreen. 2025. Guided tensor lifting. *Proceedings of the ACM on Programming Languages* 9, PLDI (2025), 1984–2006.

[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*. ACM, N/A, 237–253.

[19] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to Polyhedral MLIR. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event) *(PACT '21)*. Association for Computing Machinery, New York, NY, USA, 12 pages.

[20] 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. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–28.

[21] Greg Nelson and Derek C Oppen. 1980. Fast decision procedures based on congruence closure. *Journal of the ACM (JACM)* 27, 2 (1980), 356–364.

[22] Daniel Nichols, Abhinav Bhatele, Gokcen Kestor, and Mary Hall. 2023. Evaluating Large Language Models for High-Performance Computing Code Generation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23)*. ACM/IEEE, Denver, CO, USA, 1–14.

[23] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 32. Curran Associates, Inc., Red Hook, NY, USA, 8024–8035.

[24] Jie Qiu, Colin Cai, Sahil Bhatia, Niranjan Hasabnis, Sanjit A. Seshia, and Alvin Cheung. 2024. Tenspiler: A Verified-Lifting-Based Compiler for Tensor Operations. In *38th European Conference on Object-Oriented Programming (ECOOP 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 313)*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 32:1–32:28. https://doi.org/10.4230/LIPIcs.ECOOP.2024.32

[25] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. *OpenAI Blog* 1, 8 (2019), 9.

[26] 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), N/A. arXiv:1909.08053 http://arxiv.org/abs/1909.08053

[27] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, New York, NY, USA, 264–276.

[28] 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 (MAPL)*. ACM, New York, NY, USA, 10–19.

[29] 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*. ACM, New York, NY, USA, 1022–1039.

[30] Boris Weisfeiler and Andrei Leman. 1968. The reduction of a graph to canonical form and the algebra which appears therein. *nti, Series* 2, 9 (1968), 12–16.

[31] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and Extensible Equality Saturation. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–29.

[32] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Timothée Rault, Rémi Louf, Morgan Funtowicz, et al. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations (EMNLP)*. Association for Computational Linguistics, Online, 38–45.

[33] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How Powerful are Graph Neural Networks? *CoRR* abs/1810.00826 (2018), 1–10. arXiv:1810.00826 http://arxiv.org/abs/1810.00826

[34] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Lee, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Berkeley, CA, USA, 521–538.

[35] Yang Yu, Peiyu Zang, Chi Hsu Tsai, Haiming Wu, Yixin Shen, Jialing Zhang, Haoyu Wang, Zhiyou Xiao, Jingze Shi, Yuyu Luo, et al. 2026. Towards automated kernel generation in the era of llms. *arXiv preprint arXiv:2601.15727* (2026).

[36] Qi Zhan, Xing Hu, Xin Xia, and Shanping Li. 2026. Verify Implementation Equivalence of Large Models. *arXiv preprint arXiv:2603.21851* N/A, N/A (2026), 1–10.

[37] 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. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 468–492.

[38] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proc. VLDB Endow.* 16, 12 (Aug. 2023), 3848–3860. https://doi.org/10.14778/3611540.3611569

[39] Chijin Zhou, Bingzhou Qian, Gwihwan Go, Quan Zhang, Shanshan Li, and Yu Jiang. 2024. Polyjuice: Detecting mis-compilation bugs in tensor compilers with equality saturation based rewriting. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 1309–1335.

[40] 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 preprint arXiv:2509.10694* 1 (2025), 1–10.

[41] 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*. ACM, New York, NY, USA, 82–89.