Verifying Semantic Equivalence of Large Models with Equality Saturation

Kahfi Soobhan Zulkifli*
University of Virginia
USA

Wenbo Qian*
Northeastern University
USA

Shaowei Zhu
Amazon Web Services
USA

Yuan Zhou
Amazon Web Services
USA

Zhen Zhang
Amazon Web Services
USA

Chang Lou
University of Virginia
USA

Abstract

Modern machine learning frameworks support very large models by incorporating parallelism and optimization techniques. Yet, these very techniques add new layers of complexity in ensuring the correctness of the computation. An incorrect implementation of these techniques might lead to compile-time or runtime errors that can easily be observed and fixed, but it might also lead to *silent errors* that will result in incorrect computations in training or inference, which do not exhibit any obvious symptom until the model is used later. These subtle errors not only waste computation resources, but involve significant developer effort to detect and diagnose.

In this work, we propose AERIFY, a framework to automatically expose silent errors by verifying semantic equivalence of models with equality saturation. AERIFY constructs equivalence graphs (e-graphs) from intermediate representations of tensor programs, and incrementally applies rewriting rules—derived from generic templates and refined via domain-specific analysis—to prove or disprove equivalence at scale. When discrepancies remain unproven, AERIFY pinpoints the corresponding graph segments and maps them back to source code, simplifying debugging and reducing developer overhead. Our preliminary results show strong potentials of AERIFY in detecting real-world silent errors.

CCS Concepts: • **Computer systems organization** → **Reliability**; • **Software and its engineering** → **Formal software verification**; • **Computing methodologies** → **Machine learning**.

Keywords: Machine Learning, Silent Errors, Model Verification, Semantic Equivalence, Equality Saturation

*Both authors contributed equally to this research.

EuroMLSys '25, March 30-April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1538-9/2025/03
https://doi.org/10.1145/3721146.3721943

ACM Reference Format:
Kahfi Soobhan Zulkifli, Wenbo Qian, Shaowei Zhu, Yuan Zhou, Zhen Zhang, and Chang Lou. 2025. Verifying Semantic Equivalence of Large Models with Equality Saturation . In *The 5th Workshop on Machine Learning and Systems (EuroMLSys '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3721146.3721943

1 Introduction

Machine learning research today has increasingly become a race to scale, where larger models consistently outperform smaller counterparts across diverse tasks. From language models like GPT [13] to vision transformers [27], performance gains are often linked to model size, with billions of parameters enabling more expressive representations and generalization. The latest model of Llama [38] boasts 405 billion parameters, and DeepSeekV3 [42] exceeds 671 billion.

For years, scaling techniques, such as distributed training or inference, memory optimizations, and hardware accelerations, have been the primary focus to support ever-larger models. However, as models reach unprecedented scales, a new reliability crisis is emerging. Developers increasingly observe silent errors [19, 28, 39, 40]—severe quality degradation in trained models without triggering explicit error signals. For example, they are often introduced by bugs [7–9] in distributed training when synchronizing between devices. Thus, scaling alone is not enough; ensuring robustness becomes the next critical challenge.

Yet, detecting and diagnosing silent errors in ML models remains a crucial but elusive challenge. Deep learning models emerge from a complex pipeline involving ML frameworks [3, 4], graph optimizers [6, 14], schedulers [10, 15], and backends [1]. A recent study suggests such issues are also observed from faulty hardware [32]. Bugs can lurk at any of these layers, making troubleshooting a daunting task.

Despite efforts in testing of ML frameworks [35, 52] and compilers [28, 30, 55], they would not be able to ensure absence of errors. Silent errors in the ML training and inference context are currently often discovered after observing loss divergence in training or severe output quality degradation in inference, making it highly desirable to reason about correctness of the computation prior to runtime. Yet, developers

today are still relying on an *ad hoc* approach—manually extracting and comparing intermediate tensor values at different phases. This process is not only tedious but also unreliable, as floating-point computations naturally introduce small discrepancies, which makes comparisons imprecise and inconclusive. Even when discrepancies are detected, pinpointing the root cause remains a painstaking effort, leaving developers struggling with uncertainty and inefficiency. A systematic and robust detection solution is urgently needed.

In this work, we argue that *verifying* ML computations rather than relying on testing—is not only feasible but essential. ML computations are usually expressed in computational graphs, which capture the structured flow of data and transformations throughout training and inference. By analyzing the semantic equivalence between the original and transformed graphs, we can move beyond numerical comparisons to systematically detect and diagnose silent errors.

How can we ensure the correctness of a computational graph? One intuitive way to ensure that transformations done to a computational graph are correct is to use a verified tensor compiler or provably correct graph rewriting [11, 21, 36, 48, 51]. But there are currently few ML frameworks or compilers in practice offering correctness guarantees, not to mention that ML developers often need to do hand-optimizations due to the fast-moving nature of the field. Given a computational graph and its transformed version—without guarantees on transformation properties as in verified graph rewrites—it is unclear how to efficiently establish their equivalence. This challenge arises from the undecidability and complexity of program equivalence, even when restricted to DAGs without loops [24]. To tackle this, we consider a technique based on static analysis combined with equality saturation [41], which constructs an e-graph that facilitates reasoning about graph equivalence. Equality saturation has traditionally been used to optimize the compilation process while preserving correctness. However, its compact representation of equivalence classes and efficient reasoning of equivalence makes it a compelling candidate for verifying the correctness of ML computational graphs by jointly reasoning about the semantics of many potential transformations.

How do we tackle the complexity of equivalence checking? Even though the space of semantic-preserving graph transformations is vast, we observe that typical transformations on the computational graphs that enable distributed ML training or inference follow a few distinct patterns. For example, to distribute an associative computation onto multiple hardware accelerators, one typically needs to specify how data and computations are distributed, and also how the distributed computations should be aggregated to obtain the same result as if it were executed on a single device. Therefore, we focus on designing effective heuristics that can help us reason about the correctness of such patterns.

In this work, we introduce AERIFY, a framework that automatically verifies the semantic equivalence of large models

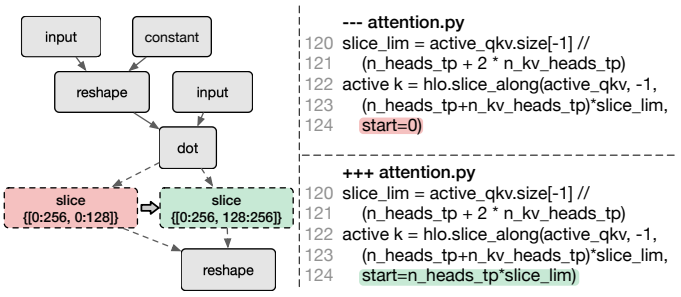

Figure 1. Incorrect Tensor Slicing When Fusing QKV Projection.

between their original and transformed versions. By leveraging the intermediate representations (IR) generated by ML frameworks, it analyzes data flow and operators, repurposing these traditionally optimization-driven graphs to expose silent errors. As a safeguard before vast computational resources are wasted on incorrect computations, AERIFY ensures that transformations preserve correctness, preventing undetected errors from affecting production workloads.

We evaluate AERIFY with real-world bug cases from AWS `transformers-neuronx` [5]. We have successfully identified and reproduced two real-world production issues. Currently, we are collaborating with AWS developers to test, improve, and deploy the prototype in production workflows.

This paper is structured as follows. In Section 2 we provide the background knowledge for ML computational graph and equality saturation. We use a motivational example to introduce our approach in Section 3. We then discuss challenges brought by large models in Section 4, and describe the system design of AERIFY in Section 5, emphasizing on how to scale our solution to very large models with derived rules and incremental checking. We conclude challenges and future works in Section 7.

2 Background

2.1 Computational Graph

DNN programs including training and inference can be represented as a computational graph for execution. A computation graph is a directed acyclic graph (DAG) that records the execution sequence of an operation set for high-level operations [2], which represents semantics of the programs.

Silent errors are often introduced by semantic changes, which can be reflected in the computational graphs. We use a real-world bug found in `transformers-neuronx` framework [5], to demonstrate how to expose silent errors using computational graphs. Shown in Figure 1, the bug manifest when concatenating a model's query, key and value (QKV) weight matrices due to an incorrect slicing offset for `active_k` in the attention computation (the root cause is marked in red). The buggy version mistakenly starts the slice at 0, while it should have accounted for the number of attention heads by offsetting the start position using `n_heads_tp * slice_lim`. The wrong start position can lead to incorrect model outputs

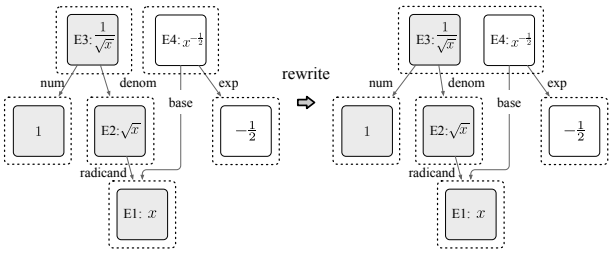

Figure 2. E-Graph Example. The two e-classes at the top (E3 and E4) are unified with rewrite rule $\frac{1}{\sqrt{x}} \rightarrow x^{-\frac{1}{2}}$, which is an actual rule needed in reasoning about self-attention computations.

without explicit errors. The fix was straightforward, requiring only an adjustment of the start parameter in the slicing operation. However, the debugging process took significant time due to the subtle nature of the indexing issue. Such issues could have been prevented if developers had access to a solution for checking the semantic equivalence with respect to an oracle implementation at the computational graph level.

2.2 Equality Saturation

Equality saturation [41] is an emerging technique within compilers and programming languages. It unifies all possible rewritten forms of an expression or program within a single, growing data structure called an e-graph. Rather than applying rewrite rules in a linear, step-by-step fashion, equality saturation systematically applies all valid rewrites in parallel, making it an ideal option to explore multiple optimized (but equivalent) forms efficiently.

Figure 2 illustrates an example of an e-graph. An e-graph is a data structure used for representing and reasoning about program equivalences efficiently. It consists of e-classes (dashed boxes), which group structurally different but semantically equivalent expressions. Within each e-class are e-nodes (solid boxes), which represent individual expression terms. Edges in the e-graph connect e-nodes to their respective child e-classes, capturing how expressions are composed.

To use equality saturation, users usually provide a list of rewrite rules. Every rewrite rule has two components: an initial pattern and a transformed pattern. Both of these patterns are semantically equivalent, but they are constructed using different components. When rewrite rules are applied, new e-nodes and edges are introduced (but existing structures are never removed). Instead, newly added expressions are merged into the corresponding e-class, preserving equivalence relationships while expanding the set of recognized expressions.

In this work, we also rely on the capability of `egglog` that combines equality saturation and analysis based on Datalog [53]. Static analyses on the computational graph is done in the form of Datalog rules, which we introduce in Section 5.

3 Verifying Equivalence with Equality Saturation

We advocate an approach using equality saturation to verify graph equivalence to expose potential silent errors. Figure 3(a) shows a distributed matrix multiplication on two devices. It partitions the input matrices, computes local matrix multiplications independently, and aggregates the results using `all-reduce`. Despite these structural differences, the final output tensors should remain semantically equivalent. As shown in Figure 3(b), we can verify the correctness of this distributed matrix multiplication by comparing its computation graph with a single-device, non-optimized baseline. We first run the ML pipeline to generate two IR graphs: the original graph (in this case, the single device setup) and the transformed graph (with the distributed setup). The system records configurations for the transformed graph, including parallelism degree and the model architecture. Our goal is to construct a unified e-graph that integrates representations of both computational graphs, systematically identifying equivalent structures.

We then iteratively expand the e-graph by incorporating nodes from IR graphs, applying rewrite rules, and merging nodes that represent the same structure into equivalence classes (e-classes). We provide a list of rules that support common features such as tensor parallelism (encoding semantics of communicative primitives such as `all-reduce`) and developers can optionally add rules for emerging parallelization and optimization techniques. As shown in the graph representation (Figure 3(b)), both the single-device and distributed computations converge to the same e-class in the final stage. This means that even though the distributed version introduces additional steps and different shape attributes, these transformations do not alter the final computation's semantics. The e-graph will eventually capture all valid transformations when the process stops (saturation is reached).

The two versions of models are semantically equivalent (verified) if the output nodes of both graphs belong to the same e-class. If developers observe severe model quality degradation but the models are verified, this result eliminates the possibility of software bugs at the level of computational graph transformations, which significantly reduces the search space in debugging. Otherwise, developers can focus on the dangling nodes that cannot be merged into a single equivalence class, which requires much less effort than blindly and repeatedly examining intermediate tensor values.

4 Challenges in Large-Scale Models

The example in Figure 3 shows how we verify the correctness of distributed matrix multiplication when parallelizing onto two devices, where the computation graph is much simpler and smaller than training or serving large DL models. In this section, we identify key challenges in scaling our approach to realistic use cases.

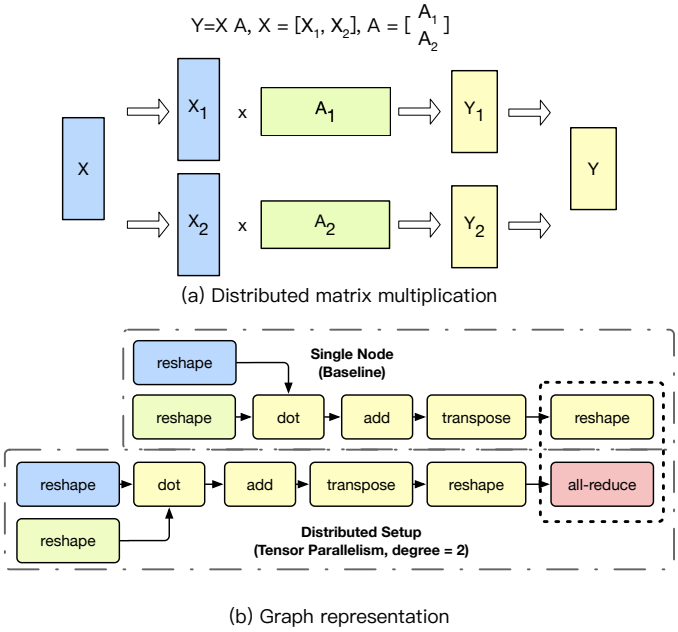

Figure 3. Distributed Matrix Multiplication Example.

Balancing rule generality and practicality. Ideally the rewriting process should only leverage general rules to capture more bug cases, but this creates a computation bottleneck—matching more nodes in the e-graph increases time and memory costs. This has more visible impacts on popular operators such as "dot" (commonly used for vector/matrix multiplications). On the other hand, highly specific rules cover fewer cases and bloat the rule set with limited reusability.

Graph scaling in large models. The size of the e-graph might be much larger compared to the size of the original computational graph, and might grow exponentially with respect to the number of nodes. In one experiment, naïvely including three layers in Llama3.1 instead of one (tripling the graph nodes) caused rule application time to jump from 1.57 seconds to 3 minutes. Expanding to 12 layers pushed computation beyond 3 hours and consumed 100GB of memory. Managing graph size to address this exponential growth is crucial for deep learning models with hundreds of layers.

Lack of debugging support. Unverified results do not automatically translate to source code fixes. Large models often involve numerous code components and transformation steps—manually searching through the whole code space is tedious. Beyond confirming bugs, developers need precise guidance to resolve discrepancies.

5 System Design and Challenges

AERIFY introduces the following key techniques:

- **Graph partitioning before checking** AERIFY partitions the computational graphs into smaller subgraphs, enabling efficient, parallel processing without overwhelming system resources (Section 5.1).
- **Layout and distribution analysis using `egglog`** AERIFY employs a scalable Datalog-style static analysis to reason about the layout and distribution of tensor slices onto different devices (Section 5.2).
- **Rewrite rule generation**: Before rewriting, AERIFY generates rewrite rules dynamically from a set of general templates based on graph analysis (Section 5.3).
- **Bug localization**: After the analysis, AERIFY maps discrepancies back to their corresponding source code lines, pinpointing the root cause of errors to streamline debugging (Section 5.4).

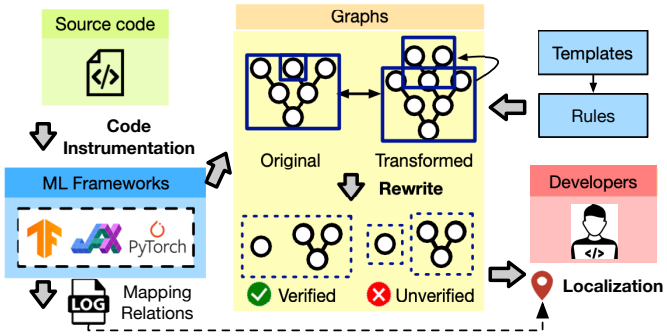

Figure 4. AERIFY Workflow.

System Workflow. We introduce AERIFY, a framework that automatically verifies the semantic equivalence of large models by incorporating these techniques. On the high level, the complete workflow of AERIFY consists of several stages. The tool starts with a preparation stage by instrumenting source codes to encode mapping relations between the IR and source codes, which is useful for the later localization step. Next, the tool generates two IR graphs using built-in functionalities of ML frameworks. AERIFY analyzes the target graphs and generates a set of efficient rules tailored for them. It then divides the graphs into several subgraphs based on different layers and constructs a unified e-graph incrementally. It outputs results (verified/unverified, reasoning steps, localization information) to developers at the end.

5.1 Graph Partitioning

As the computational graphs become larger, the computation resources in graph rewriting may increase exponentially in the worst case. To address this issue, AERIFY partitions the graphs to be verified into smaller subgraphs using heuristics. We first divide the graph at neural network layer boundaries, observing that a large class of deep learning framework and compiler optimizations do not involve graph transformations across layer boundaries, with possible reasons being 1) whole-graph optimizations can be more costly compared to local optimizations [36]; and 2) hardware vendors may provide highly optimized implementations of layers to be used [14],

making it hard to perform inter-layer optimizations. Some layers (*e.g.,* attention layers) generate significantly larger graphs, where the heuristic of dividing at layer boundaries may not be enough. For such layers we further subdivide these graphs by splitting at a predefined list of operators (*e.g.,* `softmax()` in self-attention layers). We are still evaluating the effectiveness of this subdivision heuristic in our ongoing experiments.

5.2 Tensor Layout and Partition Analysis

A key complication arises from the analysis of layout dynamics and how tensor layout relates to communication primitives, which is essential to reasoning about the correctness of distributed computations, *i.e.,* we need a sound way to reason about the frequent reshaping, slicing, transposes, and aggregation of tensors across multiple devices. A `reshape` operation might be used to adjust tensor layout for efficiency reasons or to shard tensors onto multiple devices, *e.g.,* transforming a tensor `X` with shape `(2, 6)` using `reshape(X, (2, 3, 2))` and then put its two slices `X[:, :, 0]` and `X[:, :, 1]` onto two difference devices to be processed in parallel.

AERIFY computes a relation between single-device tensors and distributed tensors. This analysis is done through the following inference rules implemented in `egglog`. First, if a single-device tensor `X` is in the same e-class as a distributed tensor `X'`, and a tensor `Y'` is obtained as a slice of `X'` along an axis d while the number of such slices equals the TP degree, then we add to our set of known facts that the concatenation of Y' on all ranks (devices) along axis d will be the same as tensors `X'` and `X`. This relation is propagated through the operators in the computational graph. For example, for element-wise operators like `add()` it is clear that if we `concat` together the resulting partial results on each rank along the sharding axis d, we will get the same result as if we `concat` the operands along axis d first and then perform the addition on the concatenated tensors. For operators with a reduction dimension such as `matmul(A, B)`, a little more care is needed since we need to use the `AllReduce()` collective on the shards if the operands `A` and `B` are sharded long the reduction dimension, and `AllGather()` otherwise.

5.3 Rewrite Rule Generation

As we discussed before, the set of *valid* rewrite rules is huge, and not all rewrite rules are *meaningful* to verify a particular computational graph. For example, one could rewrite a single `reshape()` to an arbitrary sequence of valid `reshape`'s where the shape specified in the last call is the same as the original one, but most such rewrites are useless to us. AERIFY generates rewrite rules automatically from a predefined set of templates. For example, if we observe a distributed tensor `X': [2, 4096]` and that `cat(X', dim=0)` is in the same e-class as `X` for some single-device tensor `X` along with an operator `Y' = transpose(X')`, then we generate a rewrite rule that allows us to conclude that `transpose(cat(Y', dim=1))` can be unioned with `X`, suppose we have basic properties like `transpose(transpose(A))` $\rightarrow$ `A` always available. Theoretically, one could add numerous rules involving `transpose` into the picture, yet those rules will be less useful than the one we add above since our rule connects a new distributed tensor `Y'` to an existing e-class containing at least one single-device tensor. We find such heuristics work well in practice, since the particular layout transformations we see in these computational graphs often serve particular purposes. In particular, we understand that `reshape()` and `transpose()` generated by ML frameworks and compilers are often related to tiling, sharding of weights and inputs to multiple devices, or transforming the physical layout of tensors to work together with the parallel or reduction dimension on hardware accelerators. Thus, a simple approach can effectively handle most common computational graphs without requiring comprehensive reasoning about all potential layout transformations.

5.4 Bug Localization in Source Code

We believe the equivalence checking tool should not only give the binary result (verified/unverified) but also aid developers in locating the bugs. The generated e-graph can highlight the difference (*e.g.,* a missing synchronization step between device output) at the level of computational graph nodes. Still, developers could benefit from deeper insights, such as the exact source lines where issues originate. To facilitate this, we instrument the ML compiler using our tool logging API to capture source-level debugging information during graph generation. The required manual effort is moderate as we only need to instrument a few common interfaces.

When constructing the e-graph from the source code, AERIFY stores metadata and associates source AST nodes with IR graph nodes. Each IR graph node is extended with a unique identifier, which points to the source file, function, and line number, *e.g.,* `{source line: "flash_decoding.py:42", expr: "hlo.exp(...)"}`. When converting IR graphs to e-graphs, this metadata information is preserved when incrementally adding IR graph nodes into the e-graph.

Sometimes the unverified node may not be the culprit—the real issue may stem from upstream computation. For example, a prior sharding operation introduced an incompatibility manifesting later. Thus, once the tool identifies dangling nodes, it should trace back to not only the unverified node but also dependent nodes in the chain, and retrieve the corresponding source code snippets for all tainted nodes. At the end, AERIFY generates a debugging report that lists unmatched nodes and their original expressions. It also lists all reasoning steps during rewriting using egg built-in functions. Such reasoning is useful to cross-check correctness of verification process. It also aids performance optimization; excessive use of certain rewrite rules in reasoning logs may indicate opportunities to refine them to match with fewer constructs in the graph.

6 Preliminary Results

We have implemented a prototype, AERIFY, which applies the methods outlined above. The tool is built on HLO IR, though our proposed technique is also applicable to other deep learning frameworks that leverage IR graphs. For evaluation, we focus on issues from the AWS `transformers-neuronx` [5] repository, and we have collected over 10 real-world silent errors. For each bug in the dataset, we reproduce the issue and generate two graphs: one for the baseline and one for the transformed version. Emerging models such as GPT, BERT, and Llama serve as the workloads for testing. We apply the tool to assess whether it successfully identifies the correct root cause. Currently, the tool supports 12 rules for semantics in distributed training and optimization. We have successfully used the current version to verify 2 real-world silent error cases. We are actively adding more rules and extending support for more complex semantics, such as advanced parallelism patterns (*e.g.,* context and pipeline parallelisms).

7 Discussion

We discuss limitations and the future plan of this work.

One key area for improvements is to enhance the generation and refinement of rewriting rules. Our current approach focuses on layout dynamics, however, another common type not supported yet is operator dynamics. For example, when the chain grow longer with a number of nested `add(add(add(x, y), z), w)` operands, a meta framework will recognize that they all represent a single cumulative addition and can automatically adapt to any number. This requires us to perform more fine-grained contextual analysis that can adapt to various dynamic patterns in computational graphs.

Another important direction is to extend support for more fine-grained parallelism patterns, such as context and pipeline parallelism. These parallelism techniques inherently split the model's computations across multiple contexts or stages. A rewriting rule designed to fuse operations in one stage might not be applicable in another if the operations are separated by inter-device communication boundaries. Additionally, the behavior of the model isn't solely determined by the static computational graph. We plan to add more instrumentation to collect runtime information such as timing and order of operations to support such semantics.

We plan to expand the applicability and scalability of our framework beyond the current Amazon ecosystem. We will test AERIFY on a wider range of models and ML frameworks such as DeepSpeed, and incorporating feedback from production use from AWS developers.

Finally, we see great potentials in incorporating large language models (LLMs) into our debugging process. AERIFY currently pinpoints the line introducing discrepancies, but it may not necessarily be the root cause of bugs. The discrepancies could come from missing operands due to bugs in other logic. Also, it still relies developers to manually diagnose and resolve discrepancies. By integrating other semantic knowledge, LLMs could potentially suggest more accurate diagnosis results as well as possible code fixes, thereby reducing developer effort and speeding up the debugging cycle.

8 Related Work

Failures in ML Workflows. Failures are a common reliability challenge in ML workflows. Many works [12, 33, 43] propose runtime recovery strategies to detect and mitigate crash failures or hangs. GEMINI [46] checkpoints to CPU memory of the host machines with large bandwidth to enable fast recovery from crashes. Oobleck [25] introduces a crash-fault tolerant training design with equivalent replicas. They rely on explicit error signals and are vulnerable to subtle silent errors. Inspired by silent failure studies in distributed systems [20, 23, 31, 50], we focus on exposing silent errors in ML workflows to mitigate their impacts in time.

Graph Rewriting. Optimizing ML performance through computational graph rewriting [21, 48, 51] has gained significant attention in recent years. TASO [26] automates graph substitutions by leveraging formally verified operator specifications. TENSAT [36] employs equality saturation to accelerate graph superoptimization. Alpa [54] optimizes computation graphs by automatically generating parallel execution plans at both the inter- and intra-operator levels. Mirage [51] explores an alternative approach by generating equivalent graph expressions and verifying their correctness using Z3. While their primary focus is performance optimization, these works ensure correctness by enforcing equivalence between transformations.

Equivalence Checking. Though the general problem of checking equivalence of programs is a well-known undecidable problem, various approaches have been proposed for verifying semantic equivalence in practice [22, 37, 45]. Typically, these techniques need to guess a correspondence between the two programs to be checked for equivalence by constructing product programs [16] or simulation relations [18], and discharge the proof obligations for verifying the equivalence of program fragments to SMT solvers. Work by Dahiya and Bansal [18] is similar to ours in terms of setting, since they also consider arbitrary transformations as opposed to translation validation, yet the problem of verifying the semantics of distributed and parallel ML models is drastically different from comparing compiled C programs.

ML Testing. It is known that machine learning frameworks and compilers are error-prone to bugs due to their complexity. Researchers have conducted bug studies [19, 39, 40] and propose many testing approaches for deep learning libraries [34, 44, 49] and compilers [30, 47, 55]. NNSmith [28] generates diverse DNN test models by extending existing graphs and

uses differential testing to identify bugs. PolyJuice [55] generates equivalent but syntactically different graphs with equality saturation to find mis-compilation errors. Testing can expose some but cannot eliminate all bugs, causing many issues still escaping from checking. AERIFY aims to capture those missing issues before they manifest at runtime.

Verified Compilers. Compiler verification has been a promising direction to ensure the correctness of compiled tensor programs. There have been efforts verifying the soundness of rewrite rules in compilers such as ATL [29], XLA [11] and Halide [17]. Our approach is complementary to these efforts since we start from the already transformed computational graphs. The technique for proving XLA rewrite rules [11] can also be used to formally reason about correctness of our rewrite rule templates.

9 Conclusion

Large ML models demand new techniques to detect and diagnose silent errors—issues that undermine performance without obvious error signals. We propose AERIFY to address these challenges by verifying semantic equivalence between the original and transformed IR graphs generated by modern ML frameworks. By combining equality saturation with rule generation, incremental checking, and localization, AERIFY scales to large graphs and provides guidance for efficient debugging. The framework thereby offers a systematic approach to reinforce the reliability of large-scale machine learning.

Acknowledgement

We thank the anonymous reviewers for their insightful reviews. Chang Lou is supported by the National Science Foundation grant CNS-2441284.

References

[1] Cuda toolkit. https://developer.nvidia.com/cuda-toolkit.

[2] Mlir-hlo: A standalone "hlo" mlir-based compiler. https://github.com/tensorflow/mlir-hlo.

[3] Pytorch. https://pytorch.org/.

[4] Tensorflow: An end-to-end platform for machine learning. https://www.tensorflow.org/.

[5] Transformers neuron. https://github.com/aws-neuron/transformers-neuronx.

[6] Xla (accelerated linear algebra). https://openxla.org/xla.

[7] Manual optimization does not synchronize gradients in ddp. https://github.com/Lightning-AI/pytorch-lightning/issues/9237, 2021.

[8] [bug] gradients not synchronized. https://github.com/kohya-ss/sd-scripts/issues/924, 2023.

[9] Ddp: moving model to cpu and back to gpu breaks gradient synchronization. https://github.com/pytorch/pytorch/issues/104336, 2023.

[10] S. Agarwal, C. Yan, Z. Zhang, and S. Venkataraman. Bagpipe: Accelerating deep recommendation model training. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 348–363, Koblenz, Germany, 2023.

[11] J. Arora, S. Lu, D. Jain, T. Xu, F. Houshmand, P. M. Phothilimthana, M. Lesani, P. Narayanan, K. S. Murthy, R. Bodik, A. Sabne, and C. Mendis. Tensorright: Automated verification of tensor graph rewrites. *Proc. ACM Program. Lang.*, 9(POPL), Jan. 2025.

[12] S. Athlur, N. Saran, M. Sivathanu, R. Ramjee, and N. Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 472–487, Rennes, France, 2022.

[13] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Vancouver, BC, Canada, 2020.

[14] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594. USENIX Association, Oct. 2018.

[15] A. Choudhury, Y. Wang, T. Pelkonen, K. Srinivasan, A. Jain, S. Lin, D. David, S. Soleimanifard, M. Chen, A. Yadav, R. Tijoriwala, D. Samoylov, and C. Tang. MAST: Global scheduling of ML training across Geo-Distributed datacenters at hyperscale. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 563–580. USENIX Association, July 2024.

[16] B. Churchill, O. Padon, R. Sharma, and A. Aiken. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1027–1040, Phoenix, AZ, USA, 2019.

[17] B. Clément and A. Cohen. End-to-end translation validation for the halide language. *Proc. ACM Program. Lang.*, 6(OOPSLA1), Apr. 2022.

[18] M. Dahiya and S. Bansal. Black-box equivalence checking across compiler optimizations. In *Programming Languages and Systems: 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings 15*, pages 127–147. Springer, 2017.

[19] H. Guan, Y. Xiao, J. Li, Y. Liu, and G. Bai. A comprehensive study of real-world bugs in machine learning model optimization. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 147–158, Melbourne, Victoria, Australia, 2023.

[20] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*, pages 1–16, Santa Clara, CA, USA, Oct. 2016.

[21] G. He, Z. Singh, and E. Yoneki. Mcts-geb: Monte carlo tree search is a good e-graph builder. In *Proceedings of the 3rd Workshop on Machine Learning and Systems*, EuroMLSys '23, page 26–33, Rome, Italy, 2023.

[22] Y. Herklotz, J. D. Pollard, N. Ramanathan, and J. Wickerson. Formal verification of high-level synthesis. *Proc. ACM Program. Lang.*, 5(OOPSLA), Oct. 2021.

[23] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS XVI. ACM, May 2017.

[24] O. H. Ibarra and B. S. Leininger. On the simplification and equivalence problems for straight-line programs. *J. ACM*, 30(3):641–656, July 1983.

[25] Jang, Z. Yang, Z. Zhang, X. Jin, and M. Chowdhury. Resilient distributed training of large models using pipeline templates. SOSP '23.

[26] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 47–62, Huntsville, Ontario, Canada, 2019.

[27] S. Khan, M. Naseer, M. Hayat, S. W. Zamir, F. S. Khan, and M. Shah. Transformers in vision: A survey. *ACM computing surveys (CSUR)*, 54(10s):1–41, 2022.

[28] Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. Zhang. Generating diverse and valid test cases for deep learning compilers. ASPLOS '23.

[29] A. Liu, G. Bernstein, A. Chlipala, and J. Ragan-Kelley. A verified compiler for a functional tensor language. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024.

[30] J. Liu, Y. Wei, S. Yang, Y. Deng, and L. Zhang. Coverage-guided tensor compiler fuzzing with joint ir-pass mutation. *Proc. ACM Program. Lang.*, 6(OOPSLA1), Apr. 2022.

[31] C. Lou, Y. Jing, and P. Huang. Demystifying and checking silent semantic violations in large distributed systems. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '22, pages 91–107. USENIX Association, July 2022.

[32] J. Ma, H. Pei, L. Lausen, and G. Karypis. Understanding silent data corruption in llm training, 2025.

[33] J. Mohan, A. Phanishayee, and V. Chidambaram. CheckFreq: Frequent, Fine-Grained DNN checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 203–216. USENIX Association, Feb. 2021.

[34] K. Pei, Y. Cao, J. Yang, and S. Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 1–18, Shanghai, China, 2017.

[35] H. V. Pham, T. Lutellier, W. Qi, and L. Tan. Cradle: Cross-backend validation to detect and localize bugs in deep learning libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1027–1038, 2019.

[36] J. A. Pienaar, M. Phothilimthana, M. Willsey, R. Wang, S. Roy, and Y. Yang. Equality saturation for tensor graph superoptimization. In *MLSys*, 2021.

[37] L.-N. Pouchet, E. Tucker, N. Zhang, H. Chen, D. Pal, G. Rodríguez, and Z. Zhang. Formal verification of source-to-source transformations for hls. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '24, page 97–107, Monterey, CA, USA, 2024.

[38] M. A. Research. Llama 3.1: Enhanced capabilities in large language modeling. https://ai.meta.com/llama, 2023. Version 3.1.

[39] Q. Shen, H. Ma, J. Chen, Y. Tian, S.-C. Cheung, and X. Chen. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 968–980, Athens, Greece, 2021.

[40] F. Tambon, A. Nikanjam, L. An, F. Khomh, and G. Antoniol. Silent bugs in deep learning frameworks: An empirical study of keras and tensorflow. *CoRR*, abs/2112.13314, 2021.

[41] Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. POPL '09.

[42] D. Team. Deepseek v3: A next-generation deep learning search engine. https://www.deepseek.ai, 2023. Version 3.

[43] J. Thorpe, P. Zhao, J. Eyolfson, Y. Qiao, Z. Jia, M. Zhang, R. Netravali, and G. H. Xu. Bamboo: Making preemptible instances resilient for affordable training of large DNNs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 497–513. USENIX Association, Apr. 2023.

[44] Y. Tian, K. Pei, S. Jana, and B. Ray. Deeptest: automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 303–314, Gothenburg, Sweden, 2018.

[45] Verdoolaege, G. Janssens, and M. Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. TOPLAS '12.

[46] Wang, Z. Jia, S. Zheng, Z. Zhang, X. Fu, T. S. E. Ng, and Y. Wang. Fast failure recovery in distributed training with in-memory checkpoints. SOSP '23.

[47] H. Wang, J. Chen, C. Xie, S. Liu, Z. Wang, Q. Shen, and Y. Zhao. Mlirsmith: Random program generation for fuzzing mlir compiler infrastructure. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1555–1566, 2023.

[48] H. Wang, J. Zhai, M. Gao, Z. Ma, S. Tang, L. Zheng, Y. Li, K. Rong, Y. Chen, and Z. Jia. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 37–54. USENIX Association, July 2021.

[49] J. Wang, T. Lutellier, S. Qian, H. V. Pham, and L. Tan. Eagle: creating equivalent graphs to test deep learning libraries. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 798–810, Pittsburgh, Pennsylvania, 2022.

[50] S. Wang, G. Zhang, J. Wei, Y. Wang, J. Wu, and Q. Luo. Understanding silent data corruptions in a large production cpu population. SOSP '23.

[51] M. Wu, X. Cheng, S. Liu, C. Shi, J. Ji, K. Ao, P. Velliengiri, X. Miao, O. Padon, and Z. Jia. Mirage: A multi-level superoptimizer for tensor programs, 2024.

[52] X. Xie, L. Ma, H. Wang, Y. Li, Y. Liu, and X. Li. Diffchaser: Detecting disagreements for deep neural networks. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 5772–5778. International Joint Conferences on Artificial Intelligence Organization, 7 2019.

[53] Y. Zhang, Y. R. Wang, O. Flatt, D. Cao, P. Zucker, E. Rosenthal, Z. Tatlock, and M. Willsey. Better together: Unifying datalog and equality saturation. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023.

[54] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E. P. Xing, J. E. Gonzalez, and I. Stoica. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578. USENIX Association, July 2022.

[55] C. Zhou, B. Qian, G. Go, Q. Zhang, S. Li, and Y. Jiang. Polyjuice: Detecting mis-compilation bugs in tensor compilers with equality saturation based rewriting. *Proc. ACM Program. Lang.*, 8(OOPSLA2), Oct. 2024.