2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)

Fixing Broken Graphs: LLM-Powered Automatic Code Optimization for DNN Programs

Haotian Wang[*], Yicheng Sui[*], Yudong Xie, Yicong Liu, Yufei Sun[†], Changqing Shi, Yuzhi Zhang
College of Software, Nankai University, Tianjin, China
{wanght, 2120240856, liuyicong, sccq}@mail.nankai.edu.cn, {suiyicheng, zyz}@nankai.edu.cn, yufei_sun@sina.com

***Abstract*—Deep learning compilers optimize DNN program execution by capturing them as operator-based computation graphs. However, developers' deep learning programs often contain complex Python language features that prevent compilers from recognizing the entire program as a complete computation graph, resulting in sub-optimal performance. Our analysis reveals that actual capture failures involve only a few lines of code, we believe this problem can be addressed through code repair rather than extensive compiler improvements. To address this challenge, we introduce GraphGlue, a multi-agent system that leverages LLMs to repair and optimize DNN programs for compiler requirements, thereby maximizing the performance benefits of deep learning compilers in inference scenarios. GraphGlue employs (1) graph-break cause mining (GCM) to identify hidden causes of computation graph breaks and facilitate LLM-based repair, and (2) self-correction with reject sampling (SRS) to alternate between code debugging and regeneration, effectively avoiding ineffective feedback attempts caused by incorrect initial optimization strategies. Experimental results demonstrate that programs optimized by GraphGlue achieve up to 2.19x (1.23x on average) speedup compared to using TorchDynamo directly, and deliver up to 15.77x (8.74x on average) memory savings compared to state-of-the-art AI compiler frontends. GraphGlue exhibits strong generalization capabilities across 1,411 real-world user programs, successfully optimizing 92.63% of them. Code is available at https://github.com/Jamesswang/GraphGlue.**

***Index Terms*—computation graph, LLM, code optimization**

I. Introduction

Deep learning frameworks serve as essential infrastructure for contemporary artificial intelligence research and applications [1]. In recent years, as model complexity increases, computational efficiency has become a critical bottleneck. To address this issue, mainstream frameworks like PyTorch and TensorFlow have introduced computation graph compilation techniques that capture computation graphs from user code and apply optimizations such as operator fusion and memory optimization, significantly enhancing performance [2].

Despite these significant performance advantages, computation graph compilation effectiveness critically depends on capturing accurate and complete computation graphs from user code [3]. This requirement demands high-quality deep learning code that avoids graph capture failures caused by Python's dynamic features and unsupported third-party library calls. However, most developers are not deep learning framework experts, making it extremely challenging for them to write such quality code. Currently, state-of-the-art graph capture techniques such as PyTorch's TorchScript, TorchDynamo [4], AutoGraph [5], MXNet's Hybridize [6] and MagPy [3] attempt to address this conflict by enhancing compiler functionality. However, these approaches require compiler designers to anticipate and support countless graph break scenarios, which is practically impossible.

These limitations prompted us to explore an alternative approach to the problem. Instead of focusing on compiler improvement, we examined the nature of code causing graph breaks. We find that computation graph capture failures are typically caused by just a few lines of inappropriate code that break the computational flow. As Fig. 1 shows, even a single operation can cause significant graph fragmentation. In this example, using `numpy.floor` instead of `torch.floor` within SubGraph 2 creates a "broken subgraph" that cannot be traced by PyTorch's graph capture mechanism. This single line of code fragments what should be a complete computation graph into three separate subgraphs (SubGraph 1, SubGraph 2, and SubGraph 3), with the numpy operation creating an untraceable gap in the computational flow. The impact becomes more severe due to module stacking and repetitive calls common in deep neural networks. When modules containing such problematic operations are repeatedly invoked or stacked in complex architectures, these small breaks accumulate, resulting in highly fragmented computation graphs. Importantly, we discovered that fixing these specific problematic code segments can effectively "glue" fragmented computation graphs back together. By simply replacing operations like `numpy.floor` with their PyTorch equivalents (e.g., `torch.floor`), we can restore graph continuity without requiring fundamental improvements to complex graph capture techniques.

Given these findings, we believe that targeting specific problematic code patterns is more effective than simply expanding compiler functionality. To this end, we propose using Large Language Models (LLMs) for addressing such failures. With their strong code understanding capabilities, LLMs can precisely identify and rewrite problematic code segments. Moreover, the limited dynamics of deep learning tasks, which often follow predictable programming patterns, make them particularly suitable for analysis by LLMs [3]. This approach allows us to directly address the root causes of graph breaks, rather than attempting to accommodate poorly written code

[*] These authors contributed equally to this work.
[†] Corresponding author: Yufei Sun (yufei_sun@sina.com).

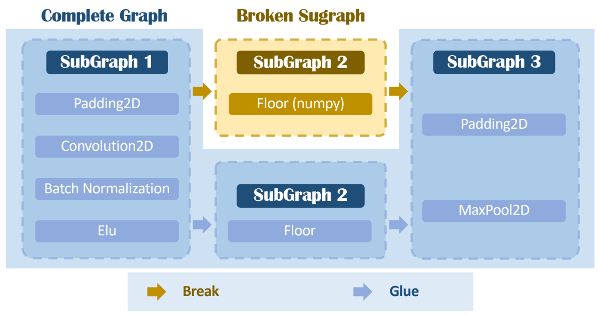


Fig. 1: Example of a single operation causing graph break.

through further enhancement of compiler features.

While this example highlights the promise of targeted code fixes, deploying LLMs for this purpose is far from trivial. Specifically, we encounter two main challenges: first, compilers hide the real causes for graph breaks, leaving LLMs without clear information needed to make targeted fixes; second, once an LLM starts with a wrong repair approach, it tends to keep following that same wrong path instead of trying different solutions, so the model gets stuck in errors rather than finding the correct answer.

To address these challenges, we propose GraphGlue, a multi-agent pre-compilation framework for inference scenarios that leverages LLMs to optimize deep learning programs *prior to compiler execution*. Pre-compilation means GraphGlue performs one-time source code repairing before runtime. It analyzes and repairs problematic patterns in user code, generating improved PyTorch code. The repaired code runs without GraphGlue involvement, ensuring no runtime overhead. Specifically, GraphGlue captures compiler-generated graph break information, transforms it into comprehensible explanations using LLM reasoning capabilities, and generates improved code that effectively "glues back together" computation graphs fragmented by code quality issues. This approach enables deep learning code to fully utilize the performance improvements offered by compilers during inference while reducing the learning curve for users.

The main contributions of this paper can be summarized as follows:

Code Optimization-Based Graph Capture: We propose, for the first time, a method to improve computation graph capture rates by optimizing user code rather than enhancing compiler functionality. Through experiments, we demonstrate that rewriting just a few key code segments can significantly improve compilation results, avoiding the need to design complex and ad-hoc compilation tools.

Graph-Break Cause Mining (GCM): We surface normally hidden graph-break traces of TorchDynamo and feed them to an LLM that rewrites the raw, compiler-centric logs into concise natural-language explanations. By converting opaque fallbacks into actionable clues, GCM equips subsequent repair stages with an explicit view of the true breakpoints, eliminating blind trial-and-error.

Self-Correction with Reject-Sampling (SRS): To prevent the LLM from pursuing increasingly incorrect repair attempts, we insert a reject-sampling loop into the feedback phase: the model may iterate on a repair several times, but if consecutive attempts continue to fail validation, the entire trajectory is discarded and the system restarts from the original code. This mechanism forces exploration of alternative solution paths and enables practical self-correction with only modest additional inference cost.

Comprehensive Experimental Validation: We evaluate GraphGlue's effectiveness across multiple real-world deep learning models. Experimental results demonstrate that GraphGlue significantly reduces the number of graph break points, resolves a wider range of graph breaks compared to baseline approaches, improves computation graph completeness, and ultimately enhances model computational efficiency.

II. Motivation and Challenges

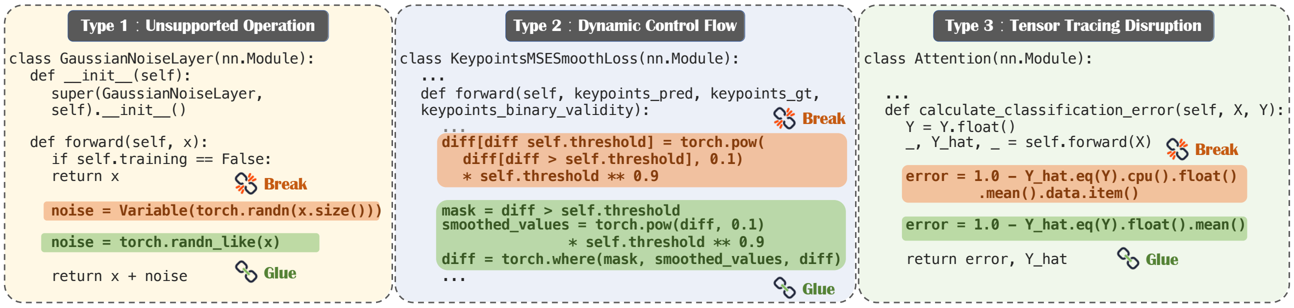
In this section, we outline the motivation for GraphGlue and identify key technical challenges through an empirical study and practical case analyses.

A. Motivation

In our empirical study, we utilize Torch Dynamo to analyze PyTorch-based repositories from GitHub, aiming to identify and summarize the primary causes of computation graph breaks. This analysis forms the foundation for our approach to computation graph optimization.

Our investigation reveals three predominant categories of causes leading to graph breaks, as illustrated in Fig. 2:

1) **Unsupported Operations:** When operations not directly supported by PyTorch's compilation mechanisms are used, graph breaks occur. These breaks typically require minimal code changes to fix, such as replacing deprecated APIs with modern equivalents or ensuring consistent device placement. As shown in our example, replacing `Variable(torch.randn(x.size()))` with the equivalent but compiler-friendly `torch.randn_like(x)` seamlessly resolves the break while preserving the original functionality.
2) **Dynamic Control Flow:** When code contains conditional branches or operations based on runtime tensor values, graph breaks occur. These breaks can often be resolved with simple restructuring that converts explicit control flow into tensor operations. For instance, replacing tensor indexing and conditional assignment (`diff[diff > self.threshold] = ...`) with the equivalent tensor-native operation using `torch.where(mask, smoothed_values, diff)` maintains the same computational logic while enabling successful graph capture.
3) **Tensor Tracing Disruptions:** When tensor values are converted to scalars or detached from the computation graph, breaks occur. These disruptions can typically be fixed by maintaining tensor semantics throughout the computation chain. As illustrated in our example, removing chain calls that extract scalar values


Type 1 : Unsupported Operation
Break
Glue
Type 2 : Dynamic Control Flow
Break
Glue
Type 3 : Tensor Tracing Disruption
Break
Glue

```
class GaussianNoiseLayer(nn.Module):
  def __init__(self):
    super(GaussianNoiseLayer,
    self).__init__()

  def forward(self, x):
    if self.training == False:
    return x
    noise = Variable(torch.randn(x.size()))
    noise = torch.randn_like(x)
    return x + noise
```

```
class KeypointsMSESmoothLoss(nn.Module):
  ...
  def forward(self, keypoints_pred, keypoints_gt,
  keypoints_binary_validity):
    ...
    diff[diff self.threshold] = torch.pow(
      diff[diff > self.threshold], 0.1)
      * self.threshold ** 0.9

    mask = diff > self.threshold
    smoothed_values = torch.pow(diff, 0.1)
                * self.threshold ** 0.9
    diff = torch.where(mask, smoothed_values, diff)
    ...
```

```
class Attention(nn.Module):

  ...
  def calculate_classification_error(self, X, Y):
    Y = Y.float()
    _, Y_hat, _ = self.forward(X)
    error = 1.0 - Y_hat.eq(Y).cpu().float()
          .mean().data.item()

    error = 1.0 - Y_hat.eq(Y).float().mean()

    return error, Y_hat
```

Fig. 2: Three common types of PyTorch computation graph breaks and how to glue them.

TABLE I: Examples of graph break logs and corresponding code.

Break Logs	Corresponding Code
dynamic shape operator: aten.bincount.default; to enable, set torch._dynamo.config.capture_dynamic_output_shape_ops = True	diff[diff $>$ self.threshold] = torch.pow(diff[diff $>$ self.threshold], 0.1) * self.threshold ** 0.9
torch.* op returned non-Tensor int call_function <Wrapped method <original item>>	res, _ = torch.topk(res.view((-1,)), int(num_voxels // self.k), sorted=False)

(`.cpu().float().mean().data.item()`) and preserving the tensor form (`.float().mean()`) effectively eliminates the graph break while producing functionally equivalent results.

Crucially, our analysis reveals that most graph breaks stem from specific code patterns rather than fundamental algorithmic requirements, and that targeted code modificationsoften involving just a few linescan successfully glue these broken graphs back together. These findings provide critical insights that inform our approach. Rather than pursuing the more complex path of enhancing the compiler's capabilities to handle problematic code patterns, we can identify specific break patterns and systematically apply appropriate code fixing to glue the graphs. This observation directly motivates our development of GraphGlue, a framework that focuses on detecting graph break points and automatically applying targeted code transformations to optimize neural network implementations for compilation.

B. Challenges

Our empirical study reveals that most graph breaks originate from a handful of subtle code snippets, yet deploying an automated repair pipeline is not as simple as calling an LLM. We must first overcome two compounding obstacles that have consistently undermined previous attempts.

Challenge 1: Unexposed Graph-Break Causes

When tracing fails, TorchDynamo does not surface an error; it simply partitions the program into multiple sub-graphs and continues execution, leaving the developer unaware of what went wrong. Developers can flip a debug flag to reveal the internal break reason, but the resulting dump is a dense mixture of operator names, guard failures, and tracing metadata that bears little resemblance to the original source code. Table I shows two such messages: both are rich in compiler jargon yet provide no obvious hint about which statement or tensor operation triggered the split. Consequently, neither humans nor LLMs have a clear, actionable starting point for repair.

Challenge 2: Lack of Effective Self-Correction

Once an LLM begins to patch the code, interaction typically follows a simple retry loop: the model proposes a fix, the compiler re-runs tracing, and success or failure is reported. If the initial hypothesis misses the real flaw, the model usually refines the same mistaken idea. When the repair strategy is fundamentally incorrect, subsequent attempts become futile variations that cannot resolve the underlying issue. Fig. 3 illustrates this challenge with a concrete example. Starting from an initial graph break, the system attempts multiple repairs but gets trapped in an incorrect solution. Despite four successive attempts with different repair strategy , the model fails to identify the root cause and cannot successfully repair the code, demonstrating how traditional feedback mechanisms lead to persistent failure cycles when the initial repair strategy is wrong.

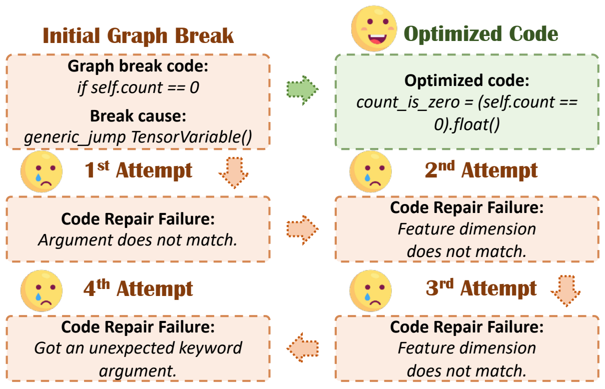


Fig. 3: Example of several failed attempts.

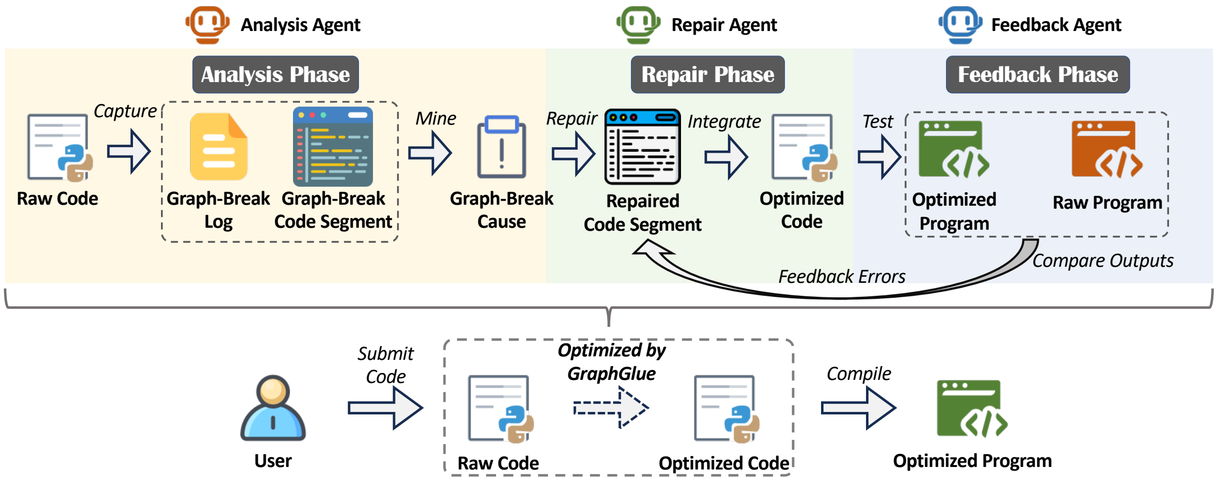


Fig. 4: The workflow of GraphGlue.

III. Methodology

A. Overview

GraphGlue addresses computation graph fragmentation through a systematic multi-agent approach that transforms the complex problem of graph capture into a manageable code repair task. Rather than improving compiler capabilities, GraphGlue focuses on the fundamental insight that graph breaks stem from identifiable code patterns that can be systematically detected and corrected.

As shown in Fig.4, GraphGlue employs a three-phase collaborative framework where specialized agents work in concert to achieve comprehensive computation graph repair. The methodology begins with automated graph-break cause extraction that precisely identifies fragmentation sources. This information guides the repair phase, which applies targeted modifications to restore graph continuity. Finally, GraphGlue systematically validates both functional correctness and successful graph capture, incorporating iterative feedback mechanisms for continuous improvement.

Each phase corresponds to a specialized agent within GraphGlue. By decomposing the complex graph repair problem into specialized, manageable subtasks, each agent can focus on its specific expertise while contributing to the overall system coherence. This design enables robust handling of diverse fragmentation patterns while maintaining reliability through systematic validation and self-correction capabilities [7].

B. Analysis Phase

The Analysis Phase forms the foundation of GraphGlue's repair pipeline, identifying and analyzing the root causes of graph breaks. Orchestrated by the **Analysis Agent**, this phase consists of two stages: computation graph capture and graph-break cause mining. They work together to transform raw compiler data into structured insights for downstream repairs.

1) Computation Graph Capture: In the stage of computation graph capture, the agent uses PyTorch's Dynamo compiler to perform a compilation-only analysis of the target model, without executing it. This allows graph break points to be captured in a controlled setting while avoiding the computational cost of full execution.

During compilation, Dynamo systematically identifies and logs each point where graph capture fails. When unsupported operations or patterns are encountered, detailed logs are generated, capturing the technical context of each failure.

However, these logs are inherently verbose and contain substantial internal compiler information that is difficult to interpret. More importantly, much of the content is only indirectly relevant to the actual code causing the graph breaks. This noise obscures the essential details needed for effective diagnosis, making further processing necessary to extract meaningful knowledge.

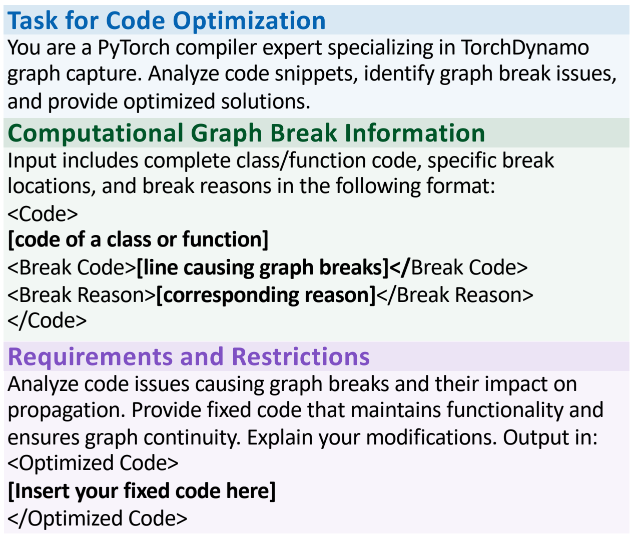
2) Graph-Break Cause Mining (GCM): To address the limitations of raw logs, the Analysis Agent performs GCM to systematically transform unstructured compiler logs into structured, interpretable representations. This stage also incorporates code structure analysis to ensure that repairs are targeted and contextually accurate.

The agent first analyzes each graph break using an LLM to generate natural language explanations of its underlying causes. These explanations capture the semantic meaning behind low-level compiler diagnostics, bridging the gap between technical logs and human-readable insights.

With these high-level interpretations in place, the agent then maps each identified cause to its corresponding class within the source code, or to functions when class structure is not available. This mapping is achieved through Abstract Syntax Tree (AST) parsing, which decomposes the code into class-based or function-based components and establishes precise associations between graph breaks and their contextual locations. By restricting the scope of analysis to individual classes or functions, this approach avoids unnecessary complexity.

Class or function-level structure inherently provides enough context to support informed repair decisions without requiring global code analysis.

C. Repair Phase



Task for Code Optimization

You are a PyTorch compiler expert specializing in TorchDynamo graph capture. Analyze code snippets, identify graph break issues, and provide optimized solutions.

Computational Graph Break Information

Input includes complete class/function code, specific break locations, and break reasons in the following format:

<Code>
[code of a class or function]
<Break Code>**[line causing graph breaks]</**Break Code>
<Break Reason>**[corresponding reason]**</Break Reason>
</Code>

Requirements and Restrictions

Analyze code issues causing graph breaks and their impact on propagation. Provide fixed code that maintains functionality and ensures graph continuity. Explain your modifications. Output in:
<Optimized Code>
[Insert your fixed code here]
</Optimized Code>

Fig. 5: Concise prompt template of Repair Agent.

The code repair phase functions as the main phase of GraphGlue, dedicated to optimizing user code while operating under the guidance of GCM and the Feedback Agent. At the center of this phase is the **Repair Agent**, which orchestrates the entire repair process by integrating static analysis from GCM and dynamic feedback from the Feedback Agent to generate effective code modifications. Fig. 5 shows the prompt design. We leveraged QWQ [8] as the cognitive foundation for the Repair Agent, with a detailed analysis of how models with varying capabilities affect system performance presented in Section IV-D.

The code repair phase fulfills two essential functions: (1) repairing graph-breaking code blocks in user programs based on GCM's static analysis results, and (2) iteratively refining code implementation according to dynamic execution feedback generated by the Feedback Agent. To ensure consistency and reliability, the code repair phase produces output in a standardized format where analytical reasoning and code modifications are encapsulated within dedicated tags. This structured approach not only enhances output quality but also facilitates accurate interpretation by other steps in the pipeline, significantly reducing errors caused by ambiguity or incomplete information transfer.

D. Feedback Phase

To ensure the correctness and reliability of optimized code transformations, comprehensive testing, validation, and feedback mechanisms are essential. In this phase, GraphGlue employs a **Feedback Agent** that provides systematic upward feedback through iterative testing to ensure code quality. This Feedback Agent drives the entire feedback process, serving as the core component that validates optimized code and guides the debugging workflow when issues arise.

Furthermore, to enhance feedback effectiveness, we propose a self-correction with reject-sampling (SRS) mechanism that strategically alternates between contextual feedback and complete regeneration, enabling more robust error correction and solution exploration.

1) Test and Validation: The Feedback Agent evaluates optimized code by comparing the execution results between the original user source code and the optimized version using test cases derived from actual user inputs, following the correctness validation approach of MagPy [3]. Specifically, we feed identical inputs to both the original and optimized code versions and verify that they produce exactly the same outputs.

The Feedback Agent focuses on two critical aspects during testing. First, it verifies code correctness by ensuring that the optimized code produces identical outputs to the original implementation. Second, it validates whether the optimized code can be captured as a complete graph.

The workflow operates as follows: if the optimized code executes correctly with the expected result matching the original code output, precompilation concludes and the vanilla deep learning compiler proceeds with execution. Otherwise, the Feedback Agent returns detailed error information to the Repair Agent for debugging.

2) Self-Correction with Reject-Sampling: Traditional feedback mechanisms forward previous output and error messages to the development module for debugging. However, we identified limitations in this approach for our specific task. While different implementations may produce identical results when developing a project, our scenario typically has only one optimal implementation. When initial feedback-guided modifications fail, it suggests the model is usually pursuing an incorrect approach, often due to LLM hallucinations [9], and further iterations along this path rarely yield meaningful improvements.

To address this limitation, the Feedback Agent introduces rejection sampling into the feedback process. The approach follows a cyclical pattern: first, the Feedback Agent provides feedback with full context of previous attempts; if this fails, it triggers a complete regeneration by discarding all previous context, prompting the model to restart from scratch. This alternating sequence continues until success or until reaching the maximum retry limit. This strategy preserves the advantages of traditional feedback while encouraging exploration of alternative solution paths, enabling the model to tackle more challenging problems. By strategically abandoning unproductive search trajectories, the Feedback Agent increases the probability of discovering optimal solutions within a constrained optimization space.

IV. Experiments

To evaluate the performance of GraphGlue, we designed a series of experiments to address the following five key **research questions (RQs)**:

RQ1: How does GraphGlue perform in terms of operational efficiency and memory consumption compared to various baseline methods?
RQ2: How does GraphGlue generalize compared to the baseline methods?
RQ3: What is the impact of model capabilities on task performance within GraphGlue?
RQ4: What is the impact of feedback iteration count and strategies on DNN program code optimization?
RQ5: What are the effects of the different components of GraphGlue?

A. Evaluation Setup

Baseline In this article, we propose an LLM-based precompilation technique to maximize the performance benefits of deep learning compilers. To evaluate the effectiveness of GraphGlue, we compare it with three types of baselines:
1. Eager mode: PyTorch's default execution paradigm with immediate operation execution, forgoing compilation and optimization for direct execution flexibility.
2. TorchDynamo [4]: PyTorch's official compiler front-end that dynamically modifies Python bytecode to extract computation graphs. It handles subgraph partitioning.
3. MagPy [3]: A state-of-the-art compiler front-end that monitors runtime execution states to capture variable reference relationships, generating more complete computation graphs.

Implementation Details Eager mode allows for direct execution without a compilation step. All other baselines use TorchInductor [4], PyTorch's official compiler backend, which translates captured computation graphs into optimized machine code, as the compiler backend. GraphGlue denotes repairing the source code and then applying TorchDynamo for graph capture. We use "qwen-max-latest" for the Analysis Agent and "qwq-32b" for the Repair Agent. Key parameters include: maximum tokens 8192, temperature 0.7, top-p 0.8. For end-to-end performance evaluation, the input scale is detailed in Table II, with a fixed batch size of 16. Time is measured by averaging 100 runs after 100 warm-ups. Peak memory usage is calculated using `torch.cuda.max_memory_allocated()`. Unless otherwise specified, the maximum number of feedback iterations is set to 3.

Platform The evaluation is performed on an NVIDIA V100S-PCIE-32GB GPU with two Intel Xeon Gold 5218 CPUs. The software versions are CUDA 12.2, PyTorch 2.5, Python 3.9.

Benchmark The evaluation process is grounded in two distinct model collections. The first is ParityBench [10], a test set of 1,418 PyTorch models, which is used to examine the application of various Python features in real code. Given that this benchmark is only capable of automatically generating input shapes, as opposed to acquiring real data, this results in discrepancies with real-world scenarios. Magpy demonstrates that a more complete captured computation graph leads to higher performance of the compiled model [3]. Therefore, in ParityBench, we equivalently verify whether the model is captured as a complete computation graph. To address this limitation, we selected another 7 representative deep learning models for performance and memory peak evaluation. These selected models encompass prevalent neural network architectures and application scenarios: the Transformer [11] architecture is represented by the traditional Bert [12], and ReFormer [13] which solving long sequence in Transformer models; and four CNNs with varying architectures were selected for the visual processing domain: YoloV5 [14], Monodepth-18 [15], MaNet [16],and TridentNet [17], a ResNet-based [18] architecture; and finally a quantized version of ResNet applying the Banner team's algorithm [19]. Table II shows the input configuration, citations, and code sources.

TABLE II: Model information. Citation represent the number of Google Scholar citations, and Source Code represents the GitHub repo.

Model	Input shape	Citation	Source Code
Quantized	image [16, 3, 224, 224]	439	eladhoffer/quantized
MaNet	image [16, 3, 224, 224]	43	JingyunLiang/MANet
YoloV5	image [16, 3, 224, 224]	60557	ultralytics/yolov5
TridentNet	image [16, 3, 224, 224]	1225	open-mmlab/mmdetection
MonoDepth	image [16, 3, 256, 256]	3837	OniroAI/MonoDepth-PyTorch
Bert	text length 256	128904	huggingface/transformers
ReFormer	text length 1024	3301	huggingface/transformers

B. RQ1: GraphGlue vs. Baseline

Fig. 6 compares the end-to-end inference performance of graph capture methods in our baseline.

GraphGlue achieves up to 1.49x (1.24x on average) speedup over Eager execution and up to 2.19x (1.23x on average) over vanilla TorchDynamo, with comparable performance to MagPy. Unlike MagPy, which fails on the reformer model, GraphGlue successfully compiles all tested models.

This acceleration stems from two key advantages: (1) GraphGlue eliminates graph breaks, enabling complete graph capture without Python interpreter intervention between subgraphs; (2) larger operator graphs enable more aggressive backend optimizations, particularly operator fusion, whose benefits will amplify as compiler technology advances.

Fig. 7 shows peak memory consumption across methods. As model sizes increase and device memory remains limited, memory efficiency becomes critical. GraphGlue achieves 1.45x (1.15x on average) and 2.54x (1.37x on average) memory savings over Eager and vanilla TorchDynamo, respectively. While Eager execution retains numerous intermediate tensors, TorchInductor's optimizations eliminate this overhead: operator fusion removes the need to materialize intermediate results, while GPU-specific memory layouts and efficient pool management reduce fragmentation. GraphGlue's larger computation graphs amplify these benefits compared to vanilla TorchDynamo. Notably, MagPy incurs up to 15.77x (8.74x on average) higher memory overhead due to its reliance on real tensor calculations during graph capture.

Table III lists the number of subgraphs generated by GraphGlue and vanilla TorchDynamo. The subgraph count

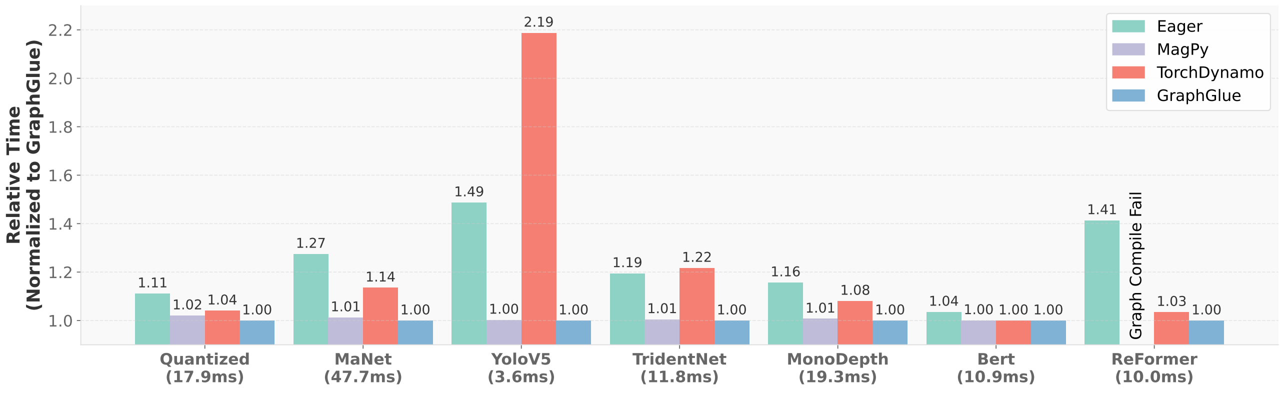


Fig. 6: End-to-end performance comparison on NVIDIA V100S, with results normalized to GraphGlue baseline, meaning all values are divided by GraphGlue performance. Horizontal axis values represent GraphGlue performance.

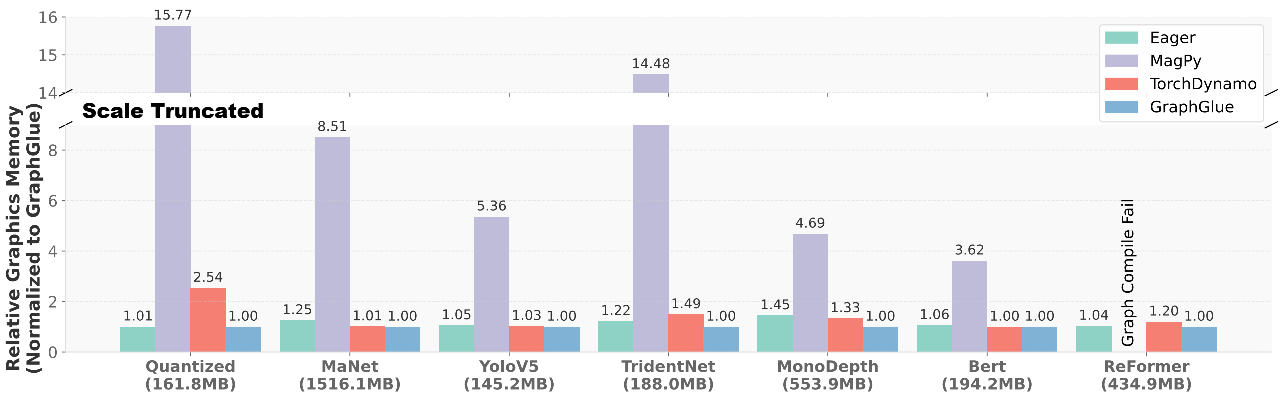


Fig. 7: Comparison of peak memory usage on NVIDIA V100S, normalized by GraphGlue, meaning all values are divided by GraphGlue results. Values on the horizontal axis represent GraphGlue results.

TABLE III: Number of subgraphs.

Model	TorchDynamo	MagPy	GraphGlue
Quantized	33	1	1
MaNet	9	1	1
YoloV5	12	1	1
TridentNet	25	1	1
MonoDepth	32	1	1
Bert	1	1	1
ReFormer	6	Error	4

TABLE IV: Result on ParityBench.

Valid cases	Method	Failed cases	Success rate
1411	TorchDynamo	285	79.80%
	MagPy	255	81.93%
	GraphGlue	104	92.63%

is calculated based on the number of times TorchInductor is called. These reported graph counts are lower than the actual values because certain tensor operators can be executed immediately without compilation. GraphGlue successfully transformed all test models into an average of 1.43 computational graphs, while vanilla TorchDynamo generated an average of 16.86 graphs. ReFormer could not be captured as a complete computational graph due to dynamic branch selection in its algorithm; nevertheless, our method still reduced graph breaks in the model, providing performance benefits where possible.

In the overhead measurement, GraphGlue evaluates across seven models, with an average precompilation time of 121.60 seconds. This process consists of log analysis (17.63 s), code optimization (97.24 s), and verification (6.73 s). Importantly, GraphGlue performs one-time precompilation before TorchDynamo, requiring no repeated execution as long as the model structure remains unchanged. This one-time cost is amortized over long-term use and becomes negligible in practice. Further discussion is provided in Section V-A.

C. RQ2: Generalization Assessment

We evaluate the generalization of our method using the ParityBench benchmark, which contains 2000 real deep learning programs crawled from GitHub. Following the settings of [3], we removed 589 samples from the benchmark where the benchmark could not generate input tensors or encountered errors. A sample is considered successful only if it is captured as a complete computation graph and passes the comparison-

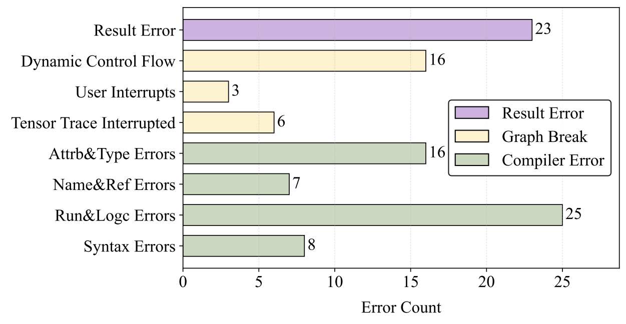


Fig. 8: Error analysis for GraphGlue.

based test in the Feedback Agent. The experimental results are shown in Table IV. TorchDynamo and MagPy can convert 79.80% and 81.93% of the models to complete computation graphs, respectively. In contrast, our method can successfully export 92.63% of the models to complete graphs. This demonstrates that our method has stronger generalization and can handle more complex Python programs.

To better understand the error modes of GraphGlue, we conduct a qualitative error analysis. By manually examining 104 failed cases, we compile statistics on various error causes as shown in Fig. 8. Our analysis reveals three primary categories of failures: compiler errors (3.99%, 56 cases), incorrect results, which means that the optimized model fails the comparison-based test (1.63%, 23 cases), and graph breaks during transformation (1.77%, 25 cases). Compiler errors are further classified into four subcategories: (1) Syntax Errors (e.g., SyntaxError, IndentationError); (2) Attribute & Type Errors (e.g., AttributeError, TypeError); (3) Name & Reference Errors (e.g., NameError, IndexError, ImportError); and (4) Runtime & Logic Errors (e.g., ValueError, ZeroDivisionError, RuntimeError, CompilationError). Graph breaks primarily stem from three sources: models with complex dynamic control flows (e.g., RNNs), explicit interruptions of the Dynamo capture process through constructs such as @torch.dynamo.disable, and tensor tracing disruptions. The majority of failures occur in scenarios that are theoretically unrepresentable by a complete computation graph, particularly those involving complex dynamic control flow or explicit disabling of graph capture.

Further analysis shows that only a small subset of compiler errors (0.78%, 11 cases) stems from language model hallucinations, such as indentation mistakes or calls to undefined methods. The majority (3.19%, 45 cases) result from aggressive optimization attempts on complex code patterns or dynamic structure scenarios where MagPy similarly fails. This suggests that hallucinations have minimal impact on overall failure rates. Instead, failures are primarily due to inherent limitations of graph-based representations rather than model-specific deficiencies.

Additionally, we evaluated the accuracy of our Graph-Break Cause Mining (GCM) component by manually auditing 603 GCM-produced analyses across ParityBench and end-to-end models. The results show 98.67% accuracy (595 correct, 8 incorrect), with the few inaccuracies occurring in edge cases involving complex compound statements. These rare errors do not affect the overall conclusions of RQ2.

TABLE V: GraphGlue with different LLMs.

Model	Size	Failed cases	Success rate
Domain-Specific Models			
Qwen2.5-Coder-7B-Instruct	7B	169	88.02%
Qwen2.5-Coder-32B-Instruct	32B	138	90.22%
General-Purpose Models			
Qwen2.5-32B-Instruct	32B	140	90.08%
Qwen-2.5-max	-	142	89.94%
Reasoning Models			
DeepSeek-r1	671B	119	91.57%
QWQ(Ours)	32B	104	92.63%

D. RQ3: Revise on different LLMs

Repair phase is the main phase of GraphGlue. To explore the impact of model capabilities on code repairing, we implemented different agents using QWQ [8], Qwen2.5-Coder-7B-Instruct [20], Qwen2.5-Coder-32B-Instruct, Qwen2.5-32B-Instruct [21], Qwen2.5-Max and DeepSeek-R1 [22].

As shown in Table V, Qwen2.5-Coder-32B-Instruct outperforms Qwen2.5-Coder-7B-Instruct, highlighting the importance of model scale and capacity. Furthermore, Qwen2.5-Coder-32B-Instruct successfully processes more models than Qwen2.5-32B-Instruct, demonstrating the advantage of domain-specific training for code-related tasks. The reasoning models QWQ and DeepSeek-R1 significantly outperform Qwen2.5-Coder-32B-Instruct, underscoring the importance of chain-of-thought reasoning abilities in understanding prompt instructions and applying them to code generation tasks. QWQ performs slightly better than DeepSeek-R1, possibly because QWQ is released later and includes more relevant knowledge in its pretraining data. These results suggest that GraphGlue benefits from LLMs with strong chain-of-thought reasoning capabilities and domain-specific expertise in code understanding and generation.

E. RQ4: The impact of Feedback

To evaluate the impact of feedback mechanisms on code optimization, we conduct controlled experiments varying the number of feedback iterations up to 5. Fig. 9 presents the number of error cases (y-axis) across different feedback iteration counts (x-axis) for both traditional and our proposed feedback strategies. The results demonstrate that feedback mechanisms significantly reduce optimization failures, with the most substantial improvement occurring between 0 and 1 iterations. While both approaches show diminishing returns with additional iterations, GraphGlue exhibits consistent improvements throughout all rounds, whereas traditional feedback reaches a plateau after 2 iterations.

This performance divergence stems from fundamental differences in feedback strategies. Traditional methods, which debug previous results on the context of each iteration, often become trapped in local optima. In contrast, GraphGlue

TABLE VI: The effectiveness of each agent in GraphGlue. Green highlighted portions indicate improvements compared to Vanilla LLM. Bold values represent the best performance.

Roles	Failed cases	Success rate
Vanilla LLM	192	86.39%
w/o Analysis Agent	150(↑ 42)	89.37%
w/o Feedback Agent	145(↑ 47)	89.72%
Feedback$_3$ (Ours)	**104**(↑ 88)	**92.63%**

employs rejection sampling during specific feedback phases (iterations 2 and 4), enabling the LLM to abandon unsuccessful attempts and explore alternative solution paths. This backtracking mechanism prevents the model from persisting with incorrect approaches while effectively resolving syntax errors. As shown in Fig. 9, GraphGlue reduces the error count by 21 additional cases in the fifth iteration compared to traditional methods, demonstrating that effective feedback design is crucial for complex code optimization tasks with multiple potential solution strategies.

F. RQ5: Ablation Study

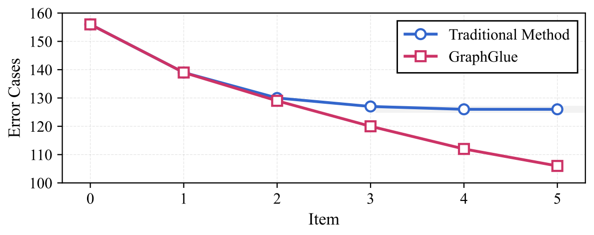


Fig. 9: The effect of feedback for GraphGlue.

To validate the contribution of each component in our proposed GraphGlue framework, we conducte a comprehensive ablation study. Table VI presents the failure rates of different variants of our model, where each variant removes a specific component from the complete framework.

The results clearly demonstrate that GraphGlue and all its variants significantly outperform the vanilla LLM baseline in code optimization tasks, with our complete GraphGlue framework achieving the lowest failure rate among all variants compared to the vanilla LLM, successfully optimizing 88 more test cases. When any part of the system is removed, GraphGlue's performance decreases. These findings indicate that each component contributes positively to the overall performance.

G. Case Study

We conduct two case analyses to qualitatively assess the effectiveness of our method from different perspectives.

Case 1: Prompt Engineering and Feedback Loop. We first examine the complete workflow using `test_pytorch_ignite` in ParityBench as an example, as shown in Fig. 10.

GraphGlue first locates graph-breaking code segments from the compiler frontend TorchDynamo and extracts compiler logs (block (2)). To enhance interpretability, GraphGlue translates compiler-level information into natural language modality (block (3)). The code optimization LLM (Repair Agent) then combines the translated graph break reasons with context information to optimize the user code (block (4)). Our prompt engineering employs three structured components to ensure effective optimization: (1) task definition that specifies the agent's role in code optimization, (2) computational graph break information including code location and break analysis from the Analysis Agent, and (3) explicit output format requirements with specified response tags. This structured approach enables the LLM to understand both the technical context and expected response format, where different agents receive tailored information: the Analysis Agent processes raw TorchDynamo logs while the Repair Agent works with interpreted break reasons.

The testing module comprehensively evaluates the rewritten code, including accuracy testing and graph-breaking testing. In this test, LLM attempts to directly use alpha as a parameter for *torch.clamp*, but overlooks the requirement that the *min* and *max* parameters must have consistent types, resulting in an error. The feedback module promptly reports the error message upstream (block (5)). The code optimization LLM subsequently repairs the code based on feedback (block (6)), establishing consistent data types for the parameters. The testing module evaluates the modified code and confirms all tests pass, concluding the code generation process (block (7)).

In contrast, direct LLM optimization attempts fail because the model lacks awareness of graph-breaking code locations. This automated pre-compilation workflow operates without requiring user attention or external input, effectively modifying user code to achieve higher performance while maintaining interpretability.

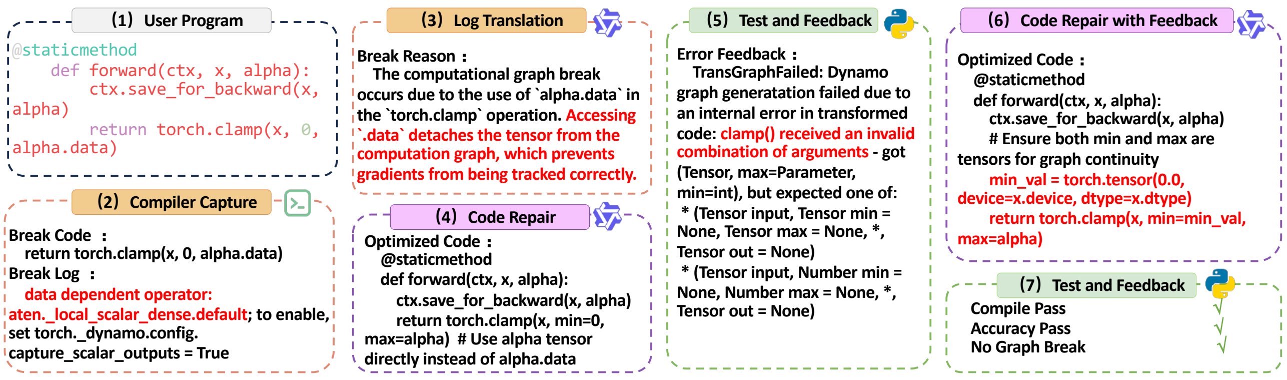
Case 2: Class-level Context Analysis. To demonstrate how class-level structure inherently provides sufficient context for effective code repair, we present a representative case involving a `DenseLayer` class, as illustrated in Fig. 11. Although this `DenseLayer` is instantiated and called by another class, GraphGlue successfully handles the repair using only class-level information.

The repair process follows three steps, as shown in Fig. 11. After GCM identifies the graph break at `if self.weight_coef != 1`, GraphGlue analyzes the class-level context to understand the nature of this interruption. Through contextual analysis, it determines that this graph break can be resolved by eliminating the functionally redundant conditional check within the `forward` method's computational flow, demonstrating an informed repair strategy. This case demonstrates that class-level structure provides sufficient context for effective code repair decisions without requiring global code analysis.

V. DISCUSSION

A. Limitations and Possible Solutions

We identify two primary limitations of our approach and propose potential solutions:



```
@staticmethod
    def forward(ctx, x, alpha):
        ctx.save_for_backward(x,
alpha)
        return torch.clamp(x, 0,
alpha.data)
```

Break Code :
    return torch.clamp(x, 0, alpha.data)
Break Log :
    data dependent operator:
aten._local_scalar_dense.default; to enable, set torch._dynamo.config.
capture_scalar_outputs = True

Break Reason :
    The computational graph break occurs due to the use of `alpha.data` in the `torch.clamp` operation. Accessing `.data` detaches the tensor from the computation graph, which prevents gradients from being tracked correctly.

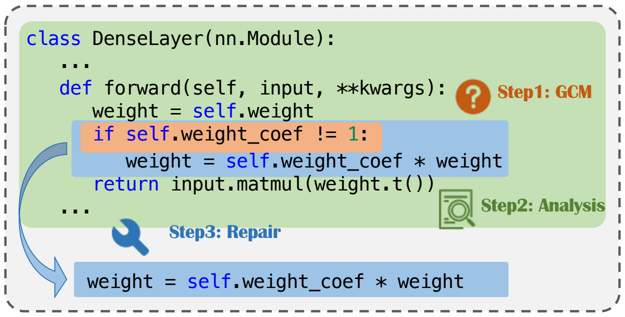
Optimized Code :
```
    @staticmethod
    def forward(ctx, x, alpha):
        ctx.save_for_backward(x, alpha)
        return torch.clamp(x, min=0,
max=alpha)  # Use alpha tensor
directly instead of alpha.data
```

Error Feedback :
    TransGraphFailed: Dynamo graph generatation failed due to an internal error in transformed code: clamp() received an invalid combination of arguments - got (Tensor, max=Parameter, min=int), but expected one of:
    * (Tensor input, Tensor min = None, Tensor max = None, *, Tensor out = None)
    * (Tensor input, Number min = None, Number max = None, *, Tensor out = None)

Optimized Code :
```
    @staticmethod
    def forward(ctx, x, alpha):
        ctx.save_for_backward(x, alpha)
        # Ensure both min and max are
tensors for graph continuity
        min_val = torch.tensor(0.0,
device=x.device, dtype=x.dtype)
        return torch.clamp(x, min=min_val,
max=alpha)
```

Compile Pass √
Accuracy Pass √
No Graph Break √

Fig. 10: A case study of test_pytorch_ignite in ParityBench.



```
class DenseLayer(nn.Module):
    ...
    def forward(self, input, **kwargs):
        weight = self.weight
        if self.weight_coef != 1:
            weight = self.weight_coef * weight
        return input.matmul(weight.t())
    ...
```

```
weight = self.weight_coef * weight
```

Fig. 11: A case of class-level context analysis.

(1) Compilation Time Overhead. The rapid advancement in LLM inference capabilities has dramatically reduced the time overhead associated with LLM-driven optimization approaches like GraphGlue. Modern specialized hardware accelerators such as Cerebras [23] can compress LLM inference processes from tens of seconds down to mere seconds. This represents a one-time cost that becomes increasingly negligible as inference speeds continue to improve, making GraphGlue more practical for routine deployment workflows.

(2) Dynamic Control Flow. Dynamic branches often depend on intermediate tensor values determined at runtime to decide subsequent computation graph structures, which is independent of code quality. This creates significant challenges for capturing the complete computation graph at the compiler front-end. Furthermore, we validated Demucs [24], an RNN model in the audio domain with over 370 citations, and a GCN model from TorchBenchmark [25]. The dynamism in RNNs stems from their recurrent structure and internal state evolution, where the hidden state acts as a memory updated at each time step through parameter sharing across sequencesintroducing data-dependent control flow. For GNN models, while basic implementations avoid dynamic behavior, practical deployments require self-loop additions to preserve node features. The dynamic nature arises from operations that filter existing self-loops before adding missing ones, causing variable output dimensions dependent on input graph structurean unavoidable variability since self-loop existence is input-dependent. Meanwhile, TorchInductor also lacks support for RNN models [3], which further complicates the adaptation of such models. Nevertheless, GraphGlue can still optimize certain dynamic scenarios by converting dynamic branches into torch.where operations or equivalent computations with mask tensors. COCKTAILER [26] attempts to address this challenge by designing a dedicated compiler backend for dynamic control flows. We believe that enabling dynamic control flow capture in the front-end represents a challenging future research direction. Potential solutions might involve developing hybrid approaches that combine static analysis with runtime information collection, or designing specialized intermediate representations that can accommodate conditional execution paths based on symbolic execution techniques.

Despite these limitations, our approach demonstrates promising results, and the modular framework design facilitates continuous improvement as technology advances.

B. Threats to Validity

External Threats. Although GraphGlue was extensively evaluated on ParityBench, comprising 1,411 real-world programs, the benchmark may not fully capture the entire spectrum of deep learning code patterns encountered in practice. Variations in coding styles and program structures could impact the generalizability of our approach.

Internal Threats. The performance of GraphGlue is influenced by the underlying large language models (LLMs). Variability across LLMs in code comprehension, optimization generation, and error correction may affect consistency and effectiveness. Additionally, the non-deterministic behavior of LLMs may lead to divergent outputs even for identical inputs.

C. Future Work

Based on our findings, we present three future directions for improving GraphGlue's broken graph repair capabilities.

Retrieval Database Construction GraphGlue currently relies on pre-trained LLM knowledge without learning from previously resolved issues. Integrating Retrieval-Augmented Generation (RAG) or domain-specific fine-tuning could enhance accuracy and reduce computational overhead. The primary challenge lies in constructing a high-quality knowledge base of broken-fixed code pairs with sufficient diversity and generalizability to avoid introducing noise or irrelevance.

More Complete Testing Following the experimental approach of MagPy [3], GraphGlue uses user-provided sample inputs for correctness validation. We plan to augment the Feedback Agent with an LLM-powered test case generation module to automatically produce comprehensive test cases, improving coverage and robustness. Additionally, we will explore incorporating non-executable code review and self-reflection to reduce reliance on executable validation.

Support for Training Current evaluation of GraphGlue focuses primarily on inference. Although the methodology is theoretically generalizable to training, several practical limitations emerge when applied to the full training lifecycle. Training involves not only forward propagation but also backward propagation and gradient updates, requiring any modification to maintain correctness throughout the entire computational process. This imposes higher demands on the LLMs and substantially increases validation overhead, as execution-based testing must now cover full training cycles to ensure correctness. Future work will explore integrating GraphGlue into IDEs to assist users with real-time optimization suggestions.

VI. Related Work

A. Deep Learning Compiler

Research in deep learning compilation primarily focuses on computation graph capture and graph-level optimization. For computation graph capture, trace-based and analysis-based approaches prevail. Trace-based methods such as torch.jit.trace [27] and torch.fx [28] generate static graphs with limited adaptability, while AutoGraph [5] and JAX.jit [29] offer recompilation capabilities but fail to detect external value modifications. LazyTensor [30], Terra [31], and Torchy [32] ensure correctness through re-tracing but introduce runtime overhead. Analysis-based approaches such as Janus [33], torch.jit.script [27], TorchDynamo [4], and MagPy [3] extract graphs through code analysis, but either support limited Python features or struggle with complex programs.

Graph-level optimization generally employs graph replacement or operator fusion. TASO [34], PET [35] and EinNet [36] optimize the computational graph structure using replacement techniques, while Rammer [37], DNNFusion [2], AStitch [38], GraphTurbo [39], and Welder [40] combine neighboring operations to reduce memory access and improve computational density. Additionally, systems like Rammer [37], DNNFusion [2], AStitch [38], GraphTurbo [39], and Welder [40] target individual operator performance optimization. Furthermore, HARP [41] enhances computation graphs through holistic analysis of host code and graph interactions. However, it still relies on existing graph capture techniques.

Despite these advances, existing methods suffer from limitations in computation graph capture that prevent handling incomplete capture scenarios. Since graph-level optimizations depend on complete computation graph capture, pursuing more comprehensive graph capture approaches is essential.

B. Automatic Programming

As LLM technology has evolved iteratively, NLP-based automatic programming systems have gained significant traction across various domains [42]–[47]. Recently, LLM-based agents implementing the ReAct paradigm [48] have further accelerated progress in automatic programming [49]–[52].

Several notable frameworks have emerged in this space. Multi-agent systems have emerged to tackle limitations in single-LLM code generation. Works like Self-Collaboration [53] and MetaGPT [54]address complex task decomposition through role-based collaboration, while TRANSAGENT [55] targets cross-language translation accuracy through specialized error handling. RRG [56] focuses on improving retrieval-augmented approaches by bridging preference gaps between retrieved and generated code.

However, an emerging challenge that has not received sufficient attention is the generation of deep learning compiler-friendly DNN programs. Despite the significant performance benefits that compilers offer through hardware-specific optimizations, current LLMs lack exposure to compiler-friendly DNN code in their pre-training data, limiting their ability to generate programs that leverage these advantages.

VII. Conclusion

In this paper, we addressed the fundamental gap between user-written DNN programs and compiler requirements: deep learning compilers often fail to capture complete computation graphs due to complex Python language features, resulting in sub-optimal performance. To bridge this gap, we presented GraphGlue, an LLM-based deep learning pre-compilation framework that effectively repairs and optimizes DNN programs to maximize the performance benefits offered by deep learning compilers. GraphGlue employs two key innovations: (1) Graph-Break Cause Mining (GCM) to identify hidden causes of computation graph breaks and facilitate LLM-based repair, and (2) Self-Correction with Reject Sampling (SRS) to alternate between code debugging and regeneration, effectively avoiding ineffective feedback attempts caused by incorrect initial optimization strategies. Our comprehensive evaluation demonstrates that GraphGlue achieves up to 2.19x speedup compared to TorchDynamo and up to 15.77x memory savings compared to state-of-the-art compiler frontends, with a 92.63% successful optimization rate on 1,411 real-world user programs. These results establish GraphGlue as an effective solution for bridging the gap between user-written DNN code and optimized compiler representations through targeted code repair. Future development includes extending GraphGlue to frameworks like TensorFlow and MXNet, and enhancing support for decoder-only architectures to improve generality and practical utility.

Acknowledgements

We sincerely thank all the anonymous reviewers for providing valuable feedback. This work is supported by the Natural Science Foundation of Tianjin (No.24JCQNJC02170).

References

[1] Q. Zhang, K. Ding, T. Lv, X. Wang, Q. Yin, Y. Zhang, J. Yu, Y. Wang, X. Li, Z. Xiang *et al.*, "Scientific large language models: A survey on biological & chemical domains," *ACM Computing Surveys*, vol. 57, no. 6, pp. 1–38, 2025.

[2] W. Niu, J. Guan, Y. Wang, G. Agrawal, and B. Ren, "Dnnfusion: accelerating deep neural networks execution with advanced operator fusion," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 883–898.

[3] C. Zhang, R. Dong, H. Wang, R. Zhong, J. Chen, and J. Zhai, "{MAGPY}: Compiling eager mode {DNN} programs by monitoring execution states," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024, pp. 683–698.

[4] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski *et al.*, "Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 929–947.

[5] D. Moldovan, J. M. Decker, F. Wang, A. A. Johnson, B. K. Lee, Z. Nado, D. Sculley, T. Rompf, and A. B. Wiltschko, "Autograph: imperative-style coding with graph-based performance.(2019)," *arXiv preprint arXiv:1810.08061*, 2019.

[6] Apache Software Foundation, "Apache mxnet: A flexible and efficient library for deep learning," 2021, apache Software Foundation. [Online]. Available: https://mxnet.apache.org/

[7] R. Kamoi, Y. Zhang, N. Zhang, J. Han, and R. Zhang, "When can llms actually correct their own mistakes? a critical survey of self-correction of llms," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 1417–1440, 2024.

[8] Q. Team, "Qwq-32b: Embracing the power of reinforcement learning," *URL: https://qwenlm. github. io/blog/qwq-32b*, 2025.

[9] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang *et al.*, "A survey on evaluation of large language models," *ACM transactions on intelligent systems and technology*, vol. 15, no. 3, pp. 1–45, 2024.

[10] Jansel and contributors, "pytorch-jit-paritybench," https://github.com/jansel/pytorch-jit-paritybench, 2025, gitHub repository.

[11] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[12] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, 2019, pp. 4171–4186.

[13] N. Kitaev, Ł. Kaiser, and A. Levskaya, "Reformer: The efficient transformer," *arXiv preprint arXiv:2001.04451*, 2020.

[14] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[15] C. Godard, O. Mac Aodha, and G. J. Brostow, "Unsupervised monocular depth estimation with left-right consistency," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 270–279.

[16] P. He, L. Jiao, R. Shang, S. Wang, X. Liu, D. Quan, K. Yang, and D. Zhao, "Manet: Multi-scale aware-relation network for semantic segmentation in aerial scenes," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 60, pp. 1–15, 2022.

[17] Y. Li, Y. Chen, N. Wang, and Z. Zhang, "Scale-aware trident networks for object detection," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019, pp. 6054–6063.

[18] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[19] R. Banner, I. Hubara, E. Hoffer, and D. Soudry, "Scalable methods for 8-bit training of neural networks," *Advances in neural information processing systems*, vol. 31, 2018.

[20] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu *et al.*, "Qwen2. 5-coder technical report," *arXiv preprint arXiv:2409.12186*, 2024.

[21] A. Yang, B. Yu, C. Li, D. Liu, F. Huang, H. Huang, J. Jiang, J. Tu, J. Zhang, J. Zhou *et al.*, "Qwen2. 5-1m technical report," *arXiv preprint arXiv:2501.15383*, 2025.

[22] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi *et al.*, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," *arXiv preprint arXiv:2501.12948*, 2025.

[23] S. Lie, "Cerebras architecture deep dive: First look inside the hw/sw co-design for deep learning : Cerebras systems," in *2022 IEEE Hot Chips 34 Symposium (HCS)*, 2022, pp. 1–34.

[24] A. Défossez, N. Usunier, L. Bottou, and F. R. Bach, "Music source separation in the waveform domain," *CoRR*, vol. abs/1911.13254, 2019. [Online]. Available: http://arxiv.org/abs/1911.13254

[25] PyTorch Team, "Torchbench: Pytorch benchmarks," https://github.com/pytorch/benchmark, 2025, a collection of open source benchmarks used to evaluate PyTorch performance.

[26] C. Zhang, L. Ma, J. Xue, Y. Shi, Z. Miao, F. Yang, J. Zhai, Z. Yang, and M. Yang, "Cocktailer: Analyzing and optimizing dynamic control flow in deep learning," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 681–699.

[27] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[28] J. Reed, Z. DeVito, H. He, A. Ussery, and J. Ansel, "torch. fx: Practical program capture and transformation for deep learning in python," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 638–651, 2022.

[29] R. Frostig, M. J. Johnson, and C. Leary, "Compiling machine learning programs via high-level tracing," *Systems for Machine Learning*, vol. 4, no. 9, 2018.

[30] A. Suhan, D. Libenzi, A. Zhang, P. Schuh, B. Saeta, J. Y. Sohn, and D. Shabalin, "Lazytensor: combining eager execution with domain-specific compilers," *arXiv preprint arXiv:2102.13267*, 2021.

[31] T. Kim, E. Jeong, G.-W. Kim, Y. Koo, S. Kim, G. Yu, and B.-G. Chun, "Terra: Imperative-symbolic co-execution of imperative deep learning programs," *Advances in Neural Information Processing Systems*, vol. 34, pp. 1468–1480, 2021.

[32] N. P. Lopes, "Torchy: A tracing jit compiler for pytorch," in *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, 2023, pp. 98–109.

[33] E. Jeong, S. Cho, G.-I. Yu, J. S. Jeong, D.-J. Shin, and B.-G. Chun, "{JANUS}: fast and flexible deep learning via symbolic graph execution of imperative programs," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 453–468.

[34] 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*, 2019, pp. 47–62.

[35] 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)*, 2021, pp. 37–54.

[36] L. Zheng, H. Wang, J. Zhai, M. Hu, Z. Ma, T. Wang, S. Huang, X. Miao, S. Tang, K. Huang *et al.*, "{EINNET}: Optimizing tensor programs with {Derivation-Based} transformations," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 739–755.

[37] L. Ma, Z. Xie, Z. Yang, J. Xue, Y. Miao, W. Cui, W. Hu, F. Yang, L. Zhang, and L. Zhou, "Rammer: Enabling holistic deep learning compiler optimizations with {rTasks}," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 881–897.

[38] Z. Zheng, X. Yang, P. Zhao, G. Long, K. Zhu, F. Zhu, W. Zhao, X. Liu, J. Yang, J. Zhai *et al.*, "Astitch: enabling a new multi-dimensional optimization space for memory-intensive ml training and inference on modern simt architectures," in *Proceedings of the 27th ACM Interna-*

tional Conference on Architectural Support for Programming Languages and Operating Systems, 2022, pp. 359–373.

[39] J. Zhao, S. Feng, X. Dan, F. Liu, C. Wang, S. Yuan, W. Lv, and Q. Xie, “Effectively scheduling computational graphs of deep neural networks toward their {Domain-Specific} accelerators,” in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 719–737.

[40] Y. Shi, Z. Yang, J. Xue, L. Ma, Y. Xia, Z. Miao, Y. Guo, F. Yang, and L. Zhou, “Welder: Scheduling deep learning memory access via tile-graph,” in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 701–718.

[41] W. Zhou, Y. Zhao, G. Zhang, and X. Shen, “Harp: Holistic analysis for refactoring python-based analytics programs,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 506–517.

[42] M. Liu, T. Yang, Y. Lou, X. Du, Y. Wang, and X. Peng, “Codegen4libs: A two-stage approach for library-oriented code generation,” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 434–445.

[43] D. Haughton and F. Balado, “Biocode: Two biologically compatible algorithms for embedding data in non-coding and coding regions of dna,” *BMC bioinformatics*, vol. 14, pp. 1–16, 2013.

[44] H. Zhang, W. Du, J. Shan, Q. Zhou, Y. Du, J. B. Tenenbaum, T. Shu, and C. Gan, “Building cooperative embodied agents modularly with large language models,” *arXiv preprint arXiv:2307.02485*, 2023.

[45] J. Li, G. Li, Y. Li, and Z. Jin, “Structured chain-of-thought prompting for code generation,” *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 2, pp. 1–23, 2025.

[46] Z. Li, C. Zhang, M. Pan, T. Zhang, and X. Li, “Aacegen: Attention guided adversarial code example generation for deep code models,” in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1245–1257.

[47] Z. Tang, J. Ge, S. Liu, T. Zhu, T. Xu, L. Huang, and B. Luo, “Domain adaptive code completion via language models and decoupled domain databases,” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 421–433.

[48] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, “React: Synergizing reasoning and acting in language models,” in *International Conference on Learning Representations (ICLR)*, 2023.

[49] H. Sami, S. Charas, A. Gandhi, P.-E. Gaillardon, V. Tenace *et al.*, “Nexus: A lightweight and scalable multi-agent framework for complex tasks automation,” *arXiv preprint arXiv:2502.19091*, 2025.

[50] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, “Reflexion: Language agents with verbal reinforcement learning,” *Advances in Neural Information Processing Systems*, vol. 36, pp. 8634–8652, 2023.

[51] B. Y. Lin, Y. Fu, K. Yang, F. Brahman, S. Huang, C. Bhagavatula, P. Ammanabrolu, Y. Choi, and X. Ren, “Swiftsage: A generative agent with fast and slow thinking for complex interactive tasks,” *Advances in Neural Information Processing Systems*, vol. 36, pp. 23 813–23 825, 2023.

[52] Q. Luo, Y. Ye, S. Liang, Z. Zhang, Y. Qin, Y. Lu, Y. Wu, X. Cong, Y. Lin, Y. Zhang *et al.*, “Repoagent: An llm-powered open-source framework for repository-level code documentation generation,” in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2024, pp. 436–464.

[53] Y. Dong, X. Jiang, Z. Jin, and G. Li, “Self-collaboration code generation via chatgpt,” *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, pp. 1–38, 2024.

[54] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin *et al.*, “Metagpt: Meta programming for a multi-agent collaborative framework,” in *ICLR*, 2024.

[55] Z. Yuan, W. Chen, H. Wang, K. Yu, X. Peng, and Y. Lou, “Transagent: An llm-based multi-agent system for code translation,” *arXiv preprint arXiv:2409.19894*, 2024.

[56] X. Gao, Y. Xiong, D. Wang, Z. Guan, Z. Shi, H. Wang, and S. Li, “Preference-guided refactored tuning for retrieval augmented code generation,” in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 65–77.