

# GraphNet: A Large-Scale Computational Graph Dataset for Tensor Compiler Research

Xinqi Li\* Yiqun Liu Shan Jiang Enrong Zheng Huaijin Zheng

Wenhao Dai Haodong Deng Dianhai Yu Yanjun Ma

PaddlePaddle Team, Baidu Inc.

## Abstract

We introduce GraphNet, a dataset of 2.7K real-world deep learning computational graphs with rich meta-data, spanning six major task categories across multiple deep learning frameworks. To evaluate tensor compiler performance on these samples, we propose the benchmark metric Speedup Score  $S_t$ , which jointly considers runtime speedup and execution correctness under tunable tolerance levels, offering a reliable measure of general optimization capability. Furthermore, we extend  $S_t$  to the Error-aware Speedup Score  $ES_t$ , which incorporates error information and helps compiler developers identify key performance bottlenecks. In this report, we benchmark the default tensor compilers, CINN for PaddlePaddle and TorchInductor for PyTorch, on computer vision (CV) and natural language processing (NLP) samples to demonstrate the practicality of GraphNet. The full construction pipeline with graph extraction and compiler evaluation tools is available at <https://github.com/PaddlePaddle/GraphNet>.

## 1 Introduction

The development of high-performance GPU kernels has become critical for computational efficiency in modern deep learning workloads. A widely adopted approach integrates deep learning frameworks, such as PaddlePaddle[8] and PyTorch[14], with vendor-specific operator libraries (e.g., cuDNN [2], oneDNN [12]). However, emerging hardware featuring low-precision formats (e.g., BF16[19], FP8[10]) and advanced memory access patterns is creating a growing demand for custom operators—a demand that vendor-specific libraries often struggle to accommodate. To meet these demands, modern deep learning systems increasingly rely on tensor compilers (e.g., TVM [1], XLA [6], BladeDISC [21]) to lower high-level computational graphs into efficient backend implementations for heterogeneous hardware platforms.

Despite these advances, deep learning engineers still rely heavily on manual tuning, as existing tensor compilation frameworks often lack fine-grained hardware control and offer limited support for cross-platform complex algorithms. Meanwhile, researchers are increasingly exploring the use of large language models (LLMs) [13] and AI coding agents [18] to automatically generate efficient operators and kernels. In this challenging context, systematic evaluation of existing tensor compilers is essential to identify performance bottlenecks and guide the evolution of next-generation compilers.

Existing benchmarks are often ad hoc, relying on a small set of hand-picked samples, with limited coverage of up-to-date, real-world models from the community and lacking sufficient support for cross-framework evaluation. To address these limitations, we introduce **GraphNet**, a large-scale dataset of deep learning computational graphs designed to enable systematic evaluation of tensor compilers across tasks and frameworks. We further propose a unified metric to benchmark compiler performance by jointly accounting for runtime speedup, numerical correctness, and compilation failures. Our main contributions are as follows:

---

\*Corresponding author: [lixinqi04@baidu.com](mailto:lixinqi04@baidu.com)

- We collected over 2.7k computational graphs from real-world models, covering diverse task categories and mainstream frameworks (e.g., PaddlePaddle, PyTorch). All samples are stored in a unified format and are compatible with multiple tensor compiler backends.
- We evaluate the performance of CINN[17] and TorchInductor on CV and NLP samples from GraphNet, and propose the **Speedup Score** ( $S_t$ ) to measure the general optimization capability of tensor compilers, as well as the **Error-aware Speedup Score** ( $ES_t$ ), which accounts for all types of execution failures.
- We present a detailed study of the dataset construction pipeline and sample constraints, and release the GraphNet dataset along with open-source extraction and evaluation tools.

The rest of this paper is organized as follows. Section 2 introduces dataset properties and its distribution. Section 3 presents the design of our evaluation metric and discusses experimental results. Section 4 describes the dataset construction methodology. Sections 5 and 6 review related works and outline future directions.

## 2 Dataset Properties

**Authenticity:** All samples are extracted from models in mainstream deep learning frameworks, focusing on PaddlePaddle libraries (e.g., PaddleNLP, PaddleX, PaddleScience) and PyTorch libraries (e.g., TorchVision, timm, mmseg). These libraries are actively maintained and widely adopted by the community, ensuring that GraphNet reflects real-world workloads rather than synthetic benchmarks.

**Compatibility:** GraphNet samples adopt a standardized format that integrates computational graphs with inputs, weights, and custom operators. This format preserves complete computation semantics without information loss, ensuring compatibility with diverse compiler backends, including CINN (PaddlePaddle), TorchInductor (PyTorch), XLA (TensorFlow), TVM, BladeDISC, and TensorRT. This enables fair and reproducible evaluation, as well as seamless extensions to new compilers.

**Diversity:** The dataset spans six major task categories: computer vision, natural language processing, audio, multimodal learning, scientific computing, and others. Models vary in scale from a few thousand to 10B parameters, with the most complex sample containing up to 3,686 operators. GraphNet also supports multiple data types, such as BF16, FP16, and FP32, which are essential for evaluating compiler optimizations across different numerical precisions and hardware backends.

## 3 Benchmark of Tensor Compilers

### 3.1 Compiler Evaluation

We introduce an automated compiler evaluation workflow that enables unified and repeatable testing of multiple tensor compilers. The evaluation takes GraphNet samples as input and consists of the following key steps:

1. **Baseline Execution:** Execute the original model in Eager mode on a given framework and record its output and execution time  $T_{eager}$  as the baseline for subsequent comparisons.
2. **Compiler Configuration:** Compile the original model into an optimized executable by specifying the target compiler. Currently supported compilers include CINN, TorchInductor, XLA, TVM, BladeDISC, and TensorRT.
3. **Compiled Execution:** Execute the compiled model after a warmup phase to eliminate cold-start overhead and obtain the pure execution time  $T_{compiled}$ . Record its output to verify accuracy.

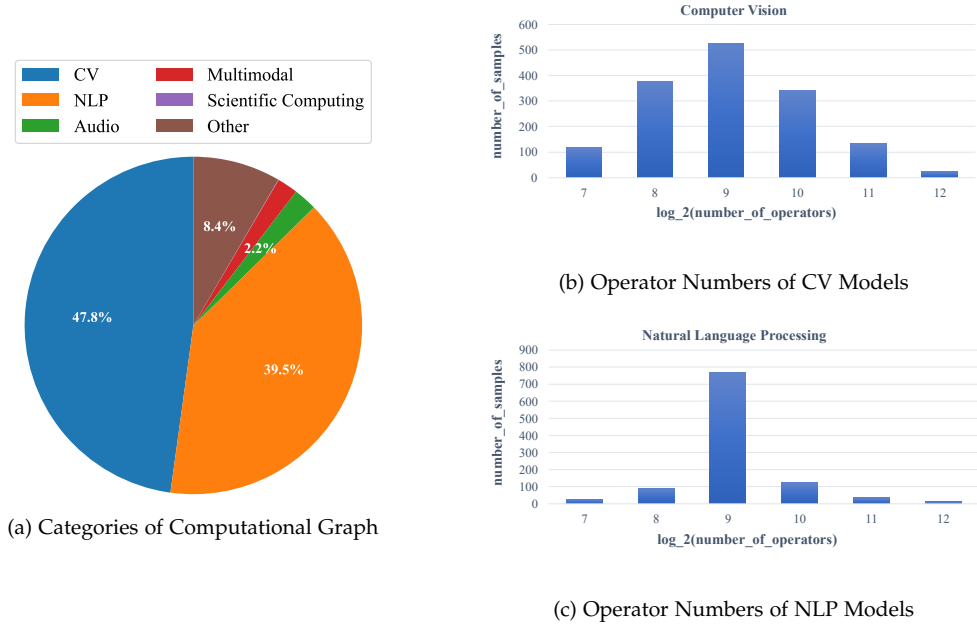


Figure 1: **Statistical properties of the GraphNet dataset.** (a) The distribution of computational graphs across six major task categories, showing that Computer Vision (47.8%) and Natural Language Processing (39.5%) are the dominant domains. (b) and (c) Histograms showing the distribution of operator counts (on a  $\log_2$  scale) for CV and NLP models, respectively. Both categories show a high concentration of graphs with operator counts around  $2^9$  (512).

4. **Performance Analysis:** Compare the outputs collected from baseline and compiled execution to validate correctness. Quantify compiler performance by considering speedup ratio  $T_{eager}/T_{compiled}$ , execution correctness, and performance degradation. The details of the evaluation metrics are discussed in Section 3.2.

## 3.2 Evaluation Metrics

We design two distinct metrics: one for benchmarking tensor compilers and one for compiler development. The benchmarking metric, detailed in Section 3.2.1, focuses primarily on acceleration and treats the error penalty as a fixed constant. The development metric (Section 3.2.2), exposes detailed error penalty information, allowing developers to configure it on demand.

### 3.2.1 Metrics for Compiler Benchmark

The performance evaluation metric is parameterized by a tunable **tolerance**  $t$ , which enables the metric to capture compiler performance under varying correctness criteria, from strict numerical precision to more relaxed settings. Tolerance  $t$  determines whether a sample is considered correctly executed, based on the check `assert_close(x, y, rtol(t), atol(t))`. The values of relative tolerance (rtol) and absolute tolerance (atol) corresponding to different  $t$  settings and data types are summarized in Appendix A.

Based on tolerance  $t$  and its associated correctness criteria, we define the **Speedup Score**  $S_t$  as a unified metric that combines speedup, accuracy, and penalty factors, providing a comprehensive measure of compiler performance:

$$S_t = \alpha^\lambda \cdot \beta^{\lambda\eta p} \cdot b^{1-\lambda} \quad (1)$$

For readability, we omit the explicit ( $t$ ) notation in this expression, while noting that  $\alpha, \beta, \lambda$ , and  $\eta$  are all dependent on the tolerance  $t$ . Eq. (1) can be viewed as the product of three components:

- **Correct executions:**  $\alpha^\lambda$  where  $\alpha$  is the geometric mean speedup of all correctly executed samples, and  $\lambda$  is the fraction of correctly executed samples, weighting the contribution of this component.
- **Performance degradation:**  $\beta^{\lambda\eta p}$  where  $\eta$  is the fraction of correctly executed samples that have speedup  $< 1$  (i.e., slowdowns among correct executions),  $\beta$  is the geometric mean speedup of these slowdown cases, and  $p \in (0, 1)$  is the penalty (default 0.1) applied to emphasize their impact.
- **Failures:**  $b^{1-\lambda}$  where  $1 - \lambda$  denotes the fraction of samples that execute incorrectly (tolerance violations, compilation failures, or runtime crashes), and  $b \in (0, 1)$  is the penalty factor (default 0.1).

The equivalence between the macro formulation and its sample-level interpretation is formally proven in Proposition B.1 (Appendix B).

**Experiment Result:** We conducted experiments on the NVIDIA H20 GPU using CV and NLP samples from GraphNet, evaluating **CINN** (the default compiler backend of PaddlePaddle) and **TorchInductor** (the default compiler backend of PyTorch) on their respective frameworks. As the workloads originate from their native ecosystems, we refer to the two settings as **PaddlePaddle** and **PyTorch** in the following discussion. Detailed experimental settings, including framework versions, are summarized in Appendix D.

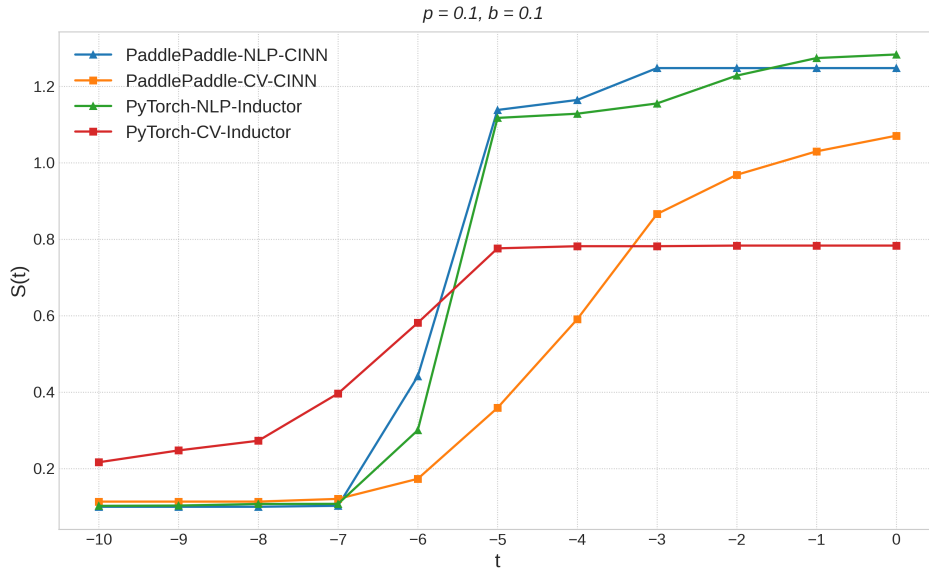


Figure 2: **Speedup Score  $S_t$  on NVIDIA H20 for CV and NLP workloads.** The vertical axis shows  $S_t$ , which integrates speedup, pass rate, and failure penalties into a unified score. Higher  $S_t$  indicates better compiler performance under the correctness-aware speedup metric defined in Equation 1. The horizontal axis  $t$  represents different numerical tolerance levels used for correctness checks, where larger  $t$  implies more relaxed thresholds.

As shown in Figure 2, the  $S_t$  trajectories exhibit distinct trends across workloads and compiler configurations. For both PaddlePaddle-NLP (blue) and PyTorch-NLP (green),  $S_t$  remains low under strict tolerances ( $t \leq -7$ ) but rises sharply within  $t \in [-6, -5]$ , indicating that most samples pass the correctness check once the tolerance is slightly relaxed. Both compilers converge to similar peak values around  $S_t \approx 1.2$  at  $t = 0$ , with PaddlePaddle achieving a marginally higher score.

In contrast, PaddlePaddle-CV (orange) exhibits a smoother performance increase from  $t = -7$  to  $t = 0$ , whereas PyTorch-CV (red) reaches a peak  $S_t \approx 0.8$  at  $t = -5$  and then plateaus. This flat trajectory suggests that increasing the tolerance further (beyond  $t = -5$ ) no longer improves the score by resolving accuracy issues, which implies that its accuracy-related failures are effectively resolved at this point. However, its performance score remains constrained by an upper bound well below 1.0, indicating heavy penalties from either widespread execution failures or performance degradation (slowdowns). While the  $S_t$  score reflects this combined impact, it does not separate these root causes. This breakdown is provided by the development metrics in the following section.

### 3.2.2 Metrics for Compiler Development

For compiler developers, error information from incorrectly executed samples is crucial. To encode this information into the evaluation metric, we reinterpret the positive tolerance domain  $t \in (0, +\infty)$  to represent discrete levels of error tolerance. We assign error codes  $c \in \{1, 2, 3\}$  to accuracy errors, runtime crashes, and compilation failures, respectively. The metric then tolerates errors based on the level  $t$ :  $t \geq 1$  tolerates accuracy errors ( $c = 1$ ),  $t \geq 2$  additionally tolerates runtime crashes ( $c = 2$ ), and  $t \geq 3$  tolerates all failures ( $c = 3$ ).

Based on these discrete tolerance levels and error codes, we extend the fixed penalty factor  $b$  in  $S_t$  (Eq. 1) to a tolerance-dependent form  $\gamma_t$ :

$$\gamma_t = \prod_c (b^{\mathbb{1}(t < c)})^{\pi_c} = b^{\sum_c \pi_c \mathbb{1}(t < c)} \quad (2)$$

where  $c \in \{1, 2, 3\}$  is the error code,  $\pi_c$  denotes the proportion of error code  $c$  among all erroneous samples, and  $\mathbb{1}(\cdot)$  is an indicator function. As  $t$  increases, more error types are tolerated and  $\gamma_t$  monotonically increases from  $b$  to 1. With  $\gamma_t$  defined, we obtain the **Error-aware Speedup Score**:

$$ES_t = \alpha^\lambda \cdot \beta^{\lambda \eta p} \cdot \gamma_t^{1-\lambda} \quad (3)$$

For  $t \leq 0$ ,  $ES_t$  reduces to the original  $S_t$ . As  $t$  grows over the positive domain, more categories of errors are tolerated and  $ES_t$  increases monotonically. When all errors are tolerated ( $t \geq 3$ ),  $ES_t$  reaches its maximum, representing the theoretical upper bound of achievable compiler performance.

Appendix C analyzes the sample-level meaning of  $ES_t$ .

**Experiment Result:** We obtain  $ES_t$  under the same conditions as in Section 3.2.1, as shown in Figure 3.

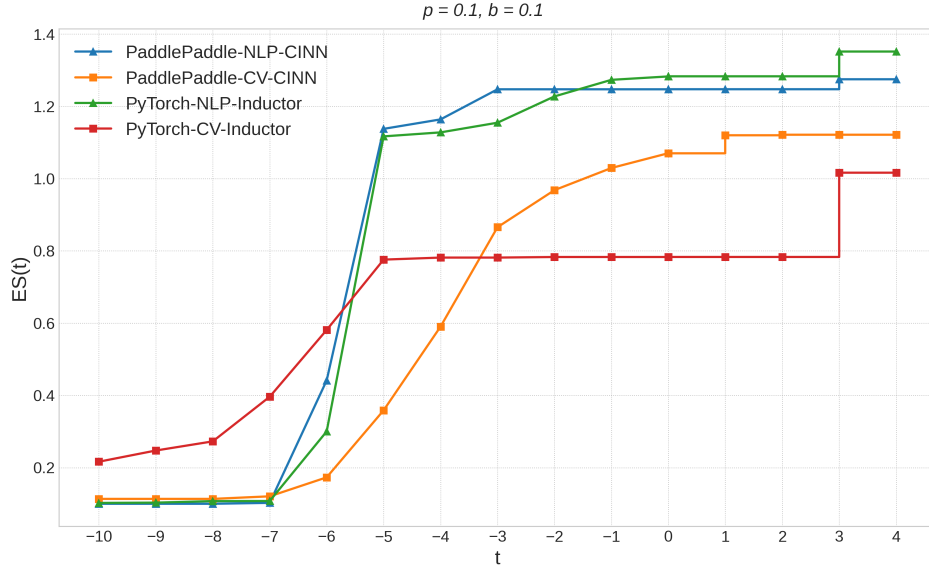


Figure 3: **Error-aware Speedup Score  $ES_t$  for CV and NLP workloads.** The vertical axis shows  $ES_t$  values, capturing compiler performance with increasing fault tolerance. Higher  $ES_t$  indicates better compiler performance under the error-tolerant speedup metric defined in Equation 3. The horizontal axis  $t$  represents different tolerance levels: for  $t \leq 0$ , it reflects numerical correctness thresholds (as in  $S_t$ ); for  $t > 0$ , it encodes the categories of tolerated errors:  $t \geq 1$  tolerates accuracy mismatches,  $t \geq 2$  tolerates runtime crashes, and  $t \geq 3$  tolerates compilation failures.

In Figure 3, PaddlePaddle-CV (orange) exhibits a discrete jump from  $t = 0$  to  $t = 1$ , indicating that a portion of its failures are due to substantial accuracy violations that are tolerated once  $t \geq 1$ . Jumps also occur at  $t = 3$  for both PyTorch-NLP (green) and PaddlePaddle-NLP (blue), revealing that a fraction of samples fail due to compilation errors. Notably, PyTorch-CV (red) shows a sharp rise at  $t = 3$ , suggesting that a large number of its samples do not pass the compilation stage.

Finally, when  $t \geq 3$ ,  $ES_t$  no longer incorporates any penalties for accuracy violations, runtime crashes, or compilation errors, and thus reflects the raw speedup score the theoretical upper bound of compiler performance. PyTorch-CV ultimately plateaus around  $ES_t \approx 1.0$ , indicating that even with all failures ignored, its average performance is neutral. This suggests many of its samples are negatively optimized (i.e., compilation leads to slower execution). As shown in Figure 6 (Appendix D), the distribution of per-sample speedups confirms that PyTorch-CV contains a substantial number of negatively optimized samples.

While  $S_t$  and  $ES_t$  provide an aggregated view of compiler performance, Tables 3–6 in the appendix provide detailed values of each component  $(\alpha, \beta, \lambda, \eta, \gamma)$ , offering a fine-grained view of how  $S_t$  and  $ES_t$  are constructed.

## 4 Construction of GraphNet

GraphNet provides a unified workflow for automated graph extraction, validation, and cross-backend performance evaluation. This section details the dataset requirements and construction methodology.

### 4.1 Dataset Constraints

We define five constraints applied to every computational graph in GraphNet to ensure overall dataset quality and cross-platform compatibility. These constraints act as validation requirements during dataset construction, ensuring consistency and usability at the source. Specifically, all computational graphs must satisfy the following constraints:

- **Runnable:** Each computational graph must successfully execute forward propagation under the designated framework without syntax errors, type mismatches, or runtime crashes.
- **Serializable:** Each sample and its associated metadata (e.g., input shapes, weight parameters) must be serializable into standard formats (e.g., JSON) and correctly de-serializable upon reloading.
- **Decomposable:** The entire computational graph must be decomposable into multiple non-overlapping subgraphs, where each subgraph represents an independent optimization unit. This supports compiler backends in performing fusion, scheduling, and other optimization tasks.
- **Statically Analyzable:** The name, type, attributes, and dependency relations of all operators must be statically extractable (e.g., via `torch.fx`) without model execution. This allows automated analysis tools to fully interpret operator semantics for structural traversal and pattern matching.
- **Custom Operator Accessible:** If a sample includes user-defined custom operators, the corresponding source code for these operators must be traceable and accessible in a modular form, ensuring reusability and integration across compiler environments.

**Remark:** At the time of writing, the first three constraints (Runnable, Serializable, and Statically Analyzable) are strictly enforced by our current validation pipeline. The remaining two (Decomposable and Custom Operator Accessible) are actively under development and represent our roadmap for expanding the datasets capabilities.

### 4.2 Construction Methodology

GraphNet provides an automated workflow for collecting computational graphs across multiple platforms while enforcing the constraints introduced in Section 4.1. The dataset construction pipeline, illustrated in Figure 4, consists of two primary components: graph extraction and graph validation.



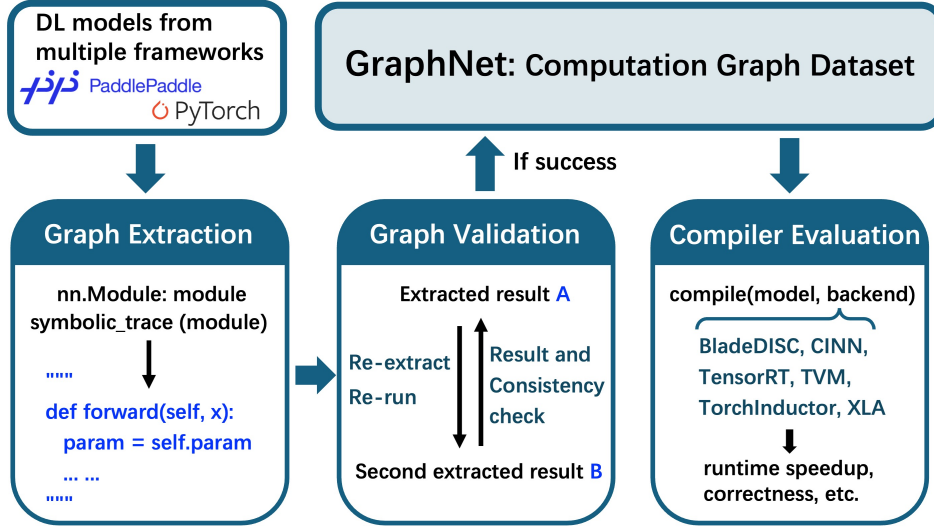


Figure 4: **GraphNet Workflow Overview.** The workflow consists of three stages: (1) **Graph Extraction:** Traces and captures computational graphs from DL models (e.g., PaddlePaddle, PyTorch); (2) **Graph Validation:** Performs a consistency check via re-extraction and re-execution to ensure usability; and (3) **Compiler Evaluation:** Uses the validated graphs from the dataset to benchmark the runtime speedup and correctness of various compiler backends.

#### 4.2.1 Graph Extraction

We design a lightweight extraction mechanism that captures dynamic computational graphs from real models and saves them as standardized GraphNet samples. As illustrated in the "Graph Extraction" component of Figure 4, we first create Python decorator-based interfaces, including `graph_net.paddle.extract` and `graph_net.torch.extract`. Users can wrap the target model with the extractor, which automatically triggers the graph extraction process at runtime. During execution, the extractor employs symbolic tracing and dynamic graph tracking mechanisms built into the framework to capture all operator invocations and tensor dependencies, thereby generating a complete dynamic computational graph. The captured graph is stored as a standardized set of files (as shown in Figure 5), which encompasses the high-level IR of the computational graph in `model.py`, weights and input metadata, and optional custom operator implementations. Together, these components form a complete GraphNet sample.

#### 4.2.2 Graph Validation

To ensure that the extracted data meets the dataset constraints, we adopt a re-extraction and re-execution mechanism during the validation stage. This workflow, illustrated in the center of Figure 4, consists of four steps: First, the validator takes an extracted sample A (the original computational graph) and deserializes it into an executable Python function, reconstructing the model structure with its input and weight metadata. Second, the reconstructed model is executed to verify that the computational graph is runnable. Third, the validator performs the extraction process again on the reconstructed model, producing a second computational graph B. Finally, the validator repeats the reconstruction and execution on graph B, comparing its outputs with those from the second stage and checking for structural and node-level consistency between graphs A and B.

This validation procedure inherently enforces all dataset constraints. Failures in serialization or deserialization terminate the reconstruction process; non-runnable models are detected during execution; and inaccessible custom operator code interrupts re-extraction. In addition, the consistency check guarantees that static analysis of all operators is performed completely and correctly.

GraphNet also incorporates a graph deduplication mechanism during data collection to eliminate redundancy. For each extracted computational graph, a unique *graph hash* value is generated from the models

source code and graph topology. The validator then identifies and removes duplicate samples by comparing their hash values, ensuring that only distinct computational graphs are retained in the final dataset.

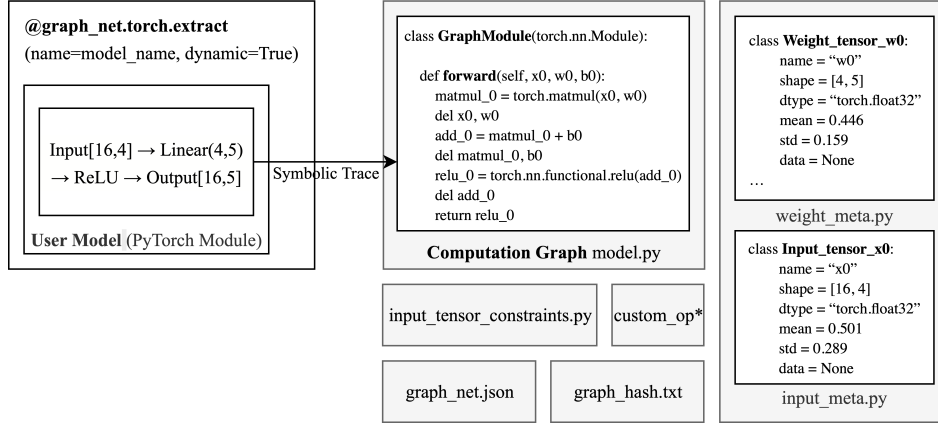


Figure 5: **GraphNet Sample Composition.** A user’s model (left), wrapped by the `@graph_net` extractor, is symbolically traced to generate a standardized set of files. This set forms a complete sample, including the high-level IR of the computation graph (`model.py`), metadata for inputs and weights (`input_meta.py`, `weight_meta.py`), and other components such as optional custom operator code.

## 5 Background and Related Works

**Tensor Compiler:** Tensor compilers[7] transform high-level computation graphs into optimized device-specific kernels through IR lowering and scheduling. Systems such as TVM [1] and Ansor [20] represent search-based compilers that rely on auto-tuning with cost models, while XLA [6] and Glow [15] follow a heuristic-based approach with graph-level optimizations. Recent works include MetaSchedule [16], Hidet [4], and BladeDISC [21], which improve scheduling abstraction, tuning efficiency, and dynamic-shape support. Industry systems include framework-specific compilers such as CINN (PaddlePaddle) and TorchInductor (PyTorch), which are tightly integrated into their respective DL frameworks, as well as vendor-specific tools like TensorRT, which are specialized for the NVIDIA ecosystem.

**Performance Benchmarks:** DL model performance evaluation can be traced back to DeepBench [11], which measured the performance of fundamental operations such as matrix multiplication across different hardware platforms. This direction was later extended by the industry-wide MLPerf [9] suite, standardizing end-to-end evaluation of training and inference workloads. More recently, CompilerGym [3] introduced benchmark datasets and reinforcement learning environments aimed at leveraging ML to improve compiler optimization, while KernelBench [13] proposed a benchmark for evaluating correctness and speedup of LLM-generated GPU kernels. In parallel, Furutanpey et al.[5] evaluated graph compilers across heterogeneous settings and introduced NGraphBench to address the gap between theoretical performance and actual deployment, emphasizing the importance of incorporating compiler effects into ML research.

## 6 Conclusion and Future Work

In this paper, we introduced GraphNet along with  $S_t$  and  $ES_t$  metrics to enable reproducible evaluation of tensor compilers across tasks and frameworks. GraphNet provides an open and well-structured resource for kernel research at the computational graph level, ensuring broad coverage of real-world workloads. Through experiments, we showed that these tools allow researchers to gain an objective and comprehensive perspective on compiler optimization capability and to better identify potential performance bottlenecks.



Our future roadmap focuses on expanding GraphNet to better serve compiler researchers and developers:

## 6.1 Completing GraphNet

**Framework Expansion** GraphNet currently supports only two deep learning frameworks, and its evaluation is limited to NVIDIA GPUs (e.g., H20, A100). We plan to expand the dataset by adding more samples from additional frameworks such as TensorFlow, JAX, and MindSpore, and extend evaluation to a wider range of hardware platforms, including TPUs and NPUs.

**Task Category Refinement** We will introduce more fine-grained categories within the existing six major domains (CV, NLP, Audio, Multimodal, Scientific Computing, and Other). This will enable more targeted evaluation of compilers on specific application scenarios.

**Sample Feature Enhancement** We aim to enrich sample features by enhancing our graph decomposition tools, allowing full graphs to be more easily split into disjoint subgraphs. We also seek to broaden support for more complex custom operators to preserve model-specific functionality.

**Distributed Scenario Support** Finally, we will extend GraphNet to incorporate distributed computing scenarios, so that GraphNet can capture computation graphs with communication operators and support the evaluation of compiler optimizations in large-scale distributed systems.

## 6.2 Applying GraphNet

**Systematic Compiler Evaluation** Beyond the limited experiment on CINN and TorchInductor, GraphNet can be extended to benchmark a broader range of tensor compilers under a unified metric. From the users perspective, this enables users to select compilers based on task categories and framework requirements. From the compiler developers perspective, it helps developers quickly identify worst-case scenarios and uncover optimization bottlenecks.

**High-level IR Translation** GraphNet includes computation graphs from multiple deep learning frameworks, providing a foundation for high-level IR translation. Such translation unifies graphs across ecosystems, ensuring that compiler evaluations are based on fully aligned datasets.

**AI for Compiler Research** GraphNet can serve as training and evaluation data for AI-generated compiler passes and kernels. These AI-generated optimizations can be directly applied to GraphNet samples and systematically compared with existing compiler backends under the  $ES_t$  metric. This enables a fair measurement of their speedup and correctness, and positions GraphNet as a benchmark platform for advancing AI-driven compiler research.

## 7 Acknowledgment

We thank the developers from the PaddlePaddle community for their invaluable support in constructing GraphNet. In particular, we especially acknowledge Zichao Xia, Ruqi Yang for their key contributions, and we also gratefully recognize the efforts of Guoyong Fang, Mengyuan Liu, Min Li, Shun Liu, Xin Wang, Xujun Chen, Yimeng Xu, Yiqiao Zhang, and Zeping Wu.

## References

- [1] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: an automated end-to-end optimizing compiler for deep learning. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, page 579594, 2018.

- [2] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning, 2014. URL <https://arxiv.org/abs/1410.0759>.
- [3] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, et al. Compilergym: Robust, performant compiler optimization environments for ai research. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 92–105. IEEE, 2022.
- [4] Yaoyao Ding, Cody Hao Yu, Bojian Zheng, Yizhi Liu, Yida Wang, and Gennady Pekhimenko. Hidet: Task-mapping programming paradigm for deep learning tensor programs. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 370–384, 2023.
- [5] Alireza Furutanpey, Carmen Walser, Philipp Raith, Pantelis A. Frangoudis, and Schahram Dustdar. Leveraging neural graph compilers in machine learning research for edge-cloud systems, 2025. URL <https://arxiv.org/abs/2504.20198>.
- [6] Samuel J. Kaufman, Phitchaya Mangpo Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. A learned performance model for tensor processing units. In *Conference on Machine Learning and Systems (MLSys)*, 2021.
- [7] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):708727, March 2021. ISSN 2161-9883. doi: 10.1109/tpds.2020.3030548. URL <http://dx.doi.org/10.1109/TPDS.2020.3030548>.
- [8] Yanjun Ma, Dianhai Yu, Tian Wu, and Haifeng Wang. Paddlepaddle: An open-source deep learning platform from industrial practice. *Frontiers of Data and Computing*, pages 105–115, 2019. URL [http://www.jfdc.cn/cn/abstract/article\\_2.shtml](http://www.jfdc.cn/cn/abstract/article_2.shtml).
- [9] Peter Mattson, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, David Patterson, Guenther Schmuelling, Hanlin Tang, et al. Mlperf: An industry standard benchmark suite for machine learning performance. *IEEE Micro*, 40(2):8–16, 2020.
- [10] Paulius Micikevicius, Dusan Stolic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart Oberman, Mohammad Shoeybi, Michael Siu, and Hao Wu. Fp8 formats for deep learning, 2022. URL <https://arxiv.org/abs/2209.05433>.
- [11] Sharan Narang and Baidu Research. Deepbench: Benchmarking deep learning operations on different hardware, 2016. URL <https://github.com/baidu-research/DeepBench>.
- [12] oneDNN Contributors. oneAPI Deep Neural Network Library (oneDNN). URL <https://github.com/uxlfoundation/oneDNN>.
- [13] Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Re, and Azalia Mirhoseini. Kernelbench: Can LLMs write efficient GPU kernels? In *Forty-second International Conference on Machine Learning*, 2025. URL <https://openreview.net/forum?id=yeoN1iQT1x>.
- [14] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Conference on Neural Information Processing Systems (NeurIPS)*, pages 8024–8035. Curran Associates, Inc., 2019.
- [15] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, Jack Montgomery, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, Misha Smelyanskiy, and Man Wang. Glow: Graph lowering compiler techniques for neural networks, 2019. URL <https://arxiv.org/abs/1805.00907>.

- [16] Junru Shao, Xiyu Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. Tensor program optimization with probabilistic programs. *Advances in Neural Information Processing Systems*, 35:35783–35796, 2022.
- [17] Baidu PaddlePaddle Team. Cinn: Compiling intermediate representation to neural networks, 2021. URL <https://github.com/PaddlePaddle/CINN>. PaddlePaddle Compiler Infrastructure.
- [18] Jianghui Wang, Vinay Joshi, Saptarshi Majumder, Xu Chao, Bin Ding, Ziqiong Liu, Pratik Prabhanjan Brahma, Dong Li, Zicheng Liu, and Emad Barsoum. Geak: Introducing triton kernel ai agent & evaluation benchmarks, 2025. URL <https://arxiv.org/abs/2507.23194>.
- [19] Norman P. Jouppi Wang, Cliff Young, Nishant Patil, and David Patterson. Bfloat16: The secret to high performance on cloud tpus. <https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus>, 2019. Accessed: 2025-10-23.
- [20] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating High-Performance tensor programs for deep learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.
- [21] Zhen Zheng, Zaifeng Pan, Dalin Wang, Kai Zhu, Wenyi Zhao, Tianyou Guo, Xiafei Qiu, Minmin Sun, Junjie Bai, Feng Zhang, Xiaoyong Du, Jidong Zhai, and Wei Lin. Bladedisc: Optimizing dynamic shape machine learning workloads via compiler approach. *Proc. ACM Manag. Data*, pages 206:1–206:29, 2023. URL <https://doi.org/10.1145/3617327>.

# Appendix

## A Configuration of atol(t) and rtol(t)

Data Type	atol(t)	atol(-5)	atol(0)
float16	$10^t$	1e-5	1
bfloat16	$10^t$	1e-5	1
float32	$10^t$	1e-5	1
float64	$10^{t \cdot 7/5}$	1e-7	1
complex32	$10^t$	1e-5	1
complex64	$10^t$	1e-5	1
complex128	$10^{t \cdot 7/5}$	1e-7	1
quint8	$10^t$	1e-5	1
quint2x4	$10^t$	1e-5	1
quint4x2	$10^t$	1e-5	1
qint8	$10^t$	1e-5	1
qint32	$10^t$	1e-5	1
others	0.0	0.0	0.0

Table 1: atol configuration

Data Type	rtol(t)	rtol(-5)	rtol(0)
float16	$10^{t \cdot 3/5}$	1e-3	1
bfloat16	$10^{t \cdot 1.796/5}$	1.6e-2	1
float32	$10^{t \cdot 5.886/5}$	1.3e-6	1
float64	$10^{t \cdot 7/5}$	1e-7	1
complex32	$10^{t \cdot 3/5}$	1e-3	1
complex64	$10^{t \cdot 5.886/5}$	1.3e-6	1
complex128	$10^{t \cdot 7/5}$	1e-7	1
quint8	$10^{t \cdot 5.886/5}$	1.3e-6	1
quint2x4	$10^{t \cdot 5.886/5}$	1.3e-6	1
quint4x2	$10^{t \cdot 5.886/5}$	1.3e-6	1
qint8	$10^{t \cdot 5.886/5}$	1.3e-6	1
qint32	$10^{t \cdot 5.886/5}$	1.3e-6	1
others	0.0	0.0	0.0

Table 2: rtol configuration

Building on PyTorch’s default testing settings, we develop a log-linear interpolation scheme to construct a (atol, rtol) configuration table. Specifically, we perform log-linear interpolation between two reference points, for instance,  $\text{atol}_{\text{fp32}}(-5) = 10^{-5}$  and  $\text{atol}_{\text{fp32}}(0) = 1$ . In essence, this means that  $\lg(\text{atol}(t))$  and  $\lg(\text{rtol}(t))$  vary linearly with  $t$ , leading to the unified representation  $\text{atol}(t), \text{rtol}(t) = 10^{kt}$ . This formulation provides a coherent framework that encompasses floating-point (float16, bfloat16, float32, float64), complex, and quantized integer types. As a result, the tolerance  $t \in (-\infty, 0]$  is mapped smoothly onto tolerance bounds across diverse precisions.

## B Sample-level Interpretation of $S_t$

In this section, we show that the macro-level Speedup Score  $S_t$  (Eq. 1) can be equivalently expressed as the geometric mean of per-sample rectified speedups.

**Definition B.1** (Rectified Speedup). For each test sample  $i$ , the *rectified speedup* under tolerance  $t$  is defined as:

$$\tilde{s}_{t,i} = \begin{cases} s_i, & \text{if } \text{correct}_{t,i} \wedge s_i \geq 1, \\ s_i^{p+1}, & \text{if } \text{correct}_{t,i} \wedge s_i < 1, \\ b, & \text{if } \neg \text{correct}_{t,i}, \end{cases} \quad (4)$$

where  $s_i$  denotes the raw speedup ratio,  $\text{correct}_{t,i}$  indicates whether the execution satisfies the tolerance criterion  $t$ ,  $p \in (0, 1)$  is the degradation penalty coefficient, and  $b \in (0, 1)$  is the failure penalty. This formulation ensures that: (i) correct executions with speedup retain their measured gain; (ii) correct executions with slowdown are exponentially penalized; and (iii) failed executions incur a fixed penalty.

We further define the *Geometric Mean Rectified Speedup* (GMRS) as:

$$\text{GMRS}_t = \left( \prod_{i=1}^N \tilde{s}_{t,i} \right)^{1/N} \quad (5)$$

where  $N$  is the total number of evaluated test samples.

**Proposition B.1.** The macro-level Speedup Score  $S_t$  defined in Eq. 1 is equivalent to the geometric mean of per-sample rectified speedups:

$$S_t = \alpha^\lambda \cdot \beta^{\lambda\eta p} \cdot b^{1-\lambda} = \text{GMRS}_t$$

*Proof.* Consider a benchmark containing  $N$  test samples in total. Among them, let  $M$  be the number of correctly executed samples, and  $K$  be the subset of those whose raw speedup is less than one (i.e., slowdowns). Formally, we define:

$$\begin{aligned} M &= \#\{i \mid \text{correct}_{t,i}\}, & \lambda &= \frac{M}{N}, \\ K &= \#\{i \mid \text{correct}_{t,i}, s_i < 1\}, & \eta &= \frac{K}{M}. \end{aligned}$$

Thus, the sample space can be partitioned as:

- $(M - K)$  correctly executed and accelerated samples ( $s_i \geq 1$ );
- $K$  correct executions with slowdown ( $s_i < 1$ );
- $(N - M)$  failed or incorrect samples.

Expanding Eq. (5) under this partition gives:

$$\text{GMRS}_t^N = \prod_{i:\text{correct}, s_i \geq 1} s_i \cdot \prod_{i:\text{correct}, s_i < 1} s_i^{p+1} \cdot \prod_{i:\neg \text{correct}} b \quad (6)$$

Define the geometric means:

$$\alpha = \left( \prod_{i:\text{correct}} s_i \right)^{1/M}, \quad \beta = \left( \prod_{i:\text{correct}, s_i < 1} s_i \right)^{1/K}$$

Hence,

$$\prod_{i:\text{correct}} s_i = \alpha^M, \quad \prod_{i:\text{correct}, s_i < 1} s_i = \beta^K,$$

and consequently:

$$\prod_{i:\text{correct}, s_i \geq 1} s_i = \frac{\alpha^M}{\beta^K}$$

Substituting into Eq. (6):

$$\begin{aligned} GMRSt^N &= \frac{\alpha^M}{\beta^K} \cdot (\beta^K)^{p+1} \cdot b^{N-M} \\ &= \alpha^M \cdot \beta^{Kp} \cdot b^{N-M} \end{aligned}$$

Taking the  $N$ -th root gives:

$$GMRSt = \alpha^{M/N} \cdot \beta^{Kp/N} \cdot b^{(N-M)/N}$$

Substituting  $\lambda = M/N$  and  $\eta = K/M$  yields:

$$GMRSt = \alpha^\lambda \cdot \beta^{\lambda\eta p} \cdot b^{1-\lambda} \tag{7}$$

This matches exactly the macro-level formulation in Eq. 1, completing the proof. □



## C Sample-level Interpretation of $ES_t$

This section provides the sample-level interpretation of the Error-aware Speedup Score  $ES_t$ . We show that the aggregated error penalty  $\gamma_t$  in Eq. (3) is equivalent to the geometric mean of per-sample penalty factors, and that  $ES_t$  can be viewed as the geometric mean of per-sample *error-aware rectified speedup*.

**Definition C.1.** For each erroneous sample  $i$ , let  $c_i \in \{1, 2, 3\}$  denote its error code (1 for accuracy errors, 2 for execution crashes, 3 for compilation failures). Given tolerance level  $t$ , we define the sample-level *penalty factor* as

$$r_{t,i} = \begin{cases} b, & t < c_i, \\ 1, & \text{otherwise,} \end{cases} \quad (8)$$

where  $b \in (0, 1)$  is the base penalty introduced in Section 3.2.1. This factor equals  $b$  if the current tolerance level  $t$  does not forgive the error type  $c_i$ , and 1 otherwise.

**Proposition C.1.** The aggregated penalty  $\gamma_t$  can be written as the geometric mean of  $\{r_{t,i}\}$  over all erroneous samples:

$$\gamma_t = \left( \prod_{i=1}^E r_{t,i} \right)^{1/E} \quad (9)$$

where  $E$  is the total number of erroneous samples.

*Proof.* From the definition of  $r_{t,i}$ , each term in the product  $\prod_i r_{t,i}$  contributes  $b$  iff  $t < c_i$ :

$$\prod_{i=1}^E r_{t,i} = \prod_{i=1}^E b^{\mathbb{1}(t < c_i)} = b^{\sum_{i=1}^E \mathbb{1}(t < c_i)}$$

Grouping by error code  $c$  and letting  $E_c$  denote the number of samples with code  $c$ , we have

$$\sum_{i=1}^E \mathbb{1}(t < c_i) = \sum_{c=1}^3 E_c \mathbb{1}(t < c)$$

thus

$$\prod_{i=1}^E r_{t,i} = b^{\sum_{c=1}^3 E_c \mathbb{1}(t < c)}$$

Taking the  $E$ -th root gives

$$\left( \prod_{i=1}^E r_{t,i} \right)^{1/E} = b^{\frac{1}{E} \sum_{c=1}^3 E_c \mathbb{1}(t < c)} = b^{\sum_{c=1}^3 \pi_c \mathbb{1}(t < c)}$$

where  $\pi_c = E_c/E$  is the fraction of error code  $c$  among all erroneous samples. The right-hand side is exactly the definition of  $\gamma_t$  in Eq. (2).  $\square$

**Definition C.2.** (Error-aware rectified speedup). Extending the rectified speedup defined in Appendix B, we introduce the per-sample *error-aware rectified speedup*:

$$\text{error\_aware\_rectified\_speedup}_{t,i} = \begin{cases} \text{speedup}_i, & \text{correct}_{t,i} \wedge \text{speedup}_i \geq 1, \\ \text{speedup}_i^{p+1}, & \text{correct}_{t,i} \wedge \text{speedup}_i < 1, \\ r_{t,i}, & \text{otherwise.} \end{cases} \quad (10)$$

**Proposition C.2.** Since  $\gamma_t$  acts as the geometric mean of  $\{r_{t,i}\}$  for a given  $t$ , the macro-level metric  $ES_t$  in Eq. (3) can be trivially proved as the geometric mean of these error-aware rectified speedup values:

$$ES_t = \left( \prod_{i=1}^N \text{error\_aware\_rectified\_speedup}_{t,i} \right)^{1/N} \quad (11)$$

## D Details of Benchmark Experiments

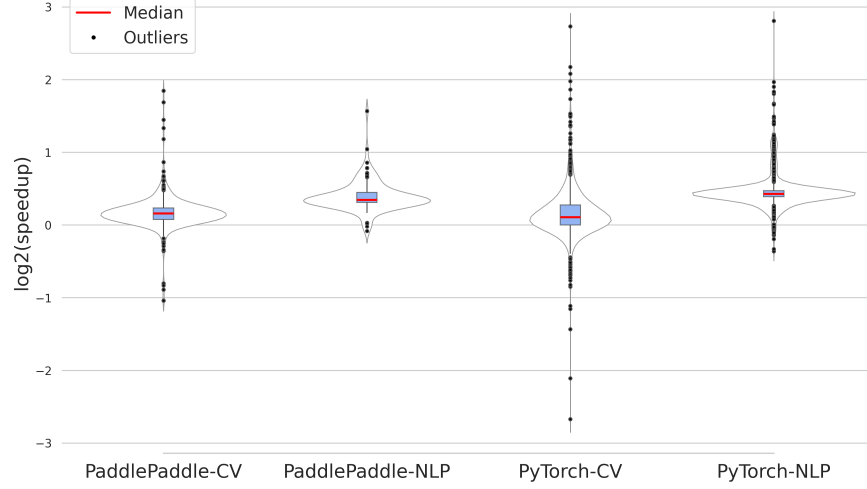


Figure 6: **Violin Plots of Per-sample Speedups ( $\log_2(\text{speedup})$ )**. It shows distribution of per-sample speedups under correctness checks  $t = 0$ . Each violin shows the speedup spread for a compiler-task pair. Wider regions denote more frequent speedup values. Samples with  $\log_2(\text{speedup}) < 0$  indicate performance degradation.

Experiments were conducted on a server with dual Intel Xeon Platinum 8563C CPUs, 2 TB DRAM, and eight NVIDIA H20-3e GPUs (144 GB each). Only one GPU was used to ensure a single-device setting. The software environment included Python 3.11, PyTorch 2.8, and PaddlePaddle 3.2.

PaddlePaddle and PyTorch show broadly consistent dataset distributions. For graphs with fewer than 256 operators, the proportions are 6% vs. 8.5% in NLP (PaddlePaddle vs. PyTorch) and 13% vs. 5.6% in CV. For graphs with more than 256 operators, the proportions are 94% vs. 91.5% in NLP and 87% vs. 94.4% in CV. These residual differences will be resolved through a high-level IR transformation tool.

The four tables below (Tables 3-6) report detailed scores for the NLP / CV models tested on PaddlePaddle and PyTorch, corresponding to the statistical results behind each point of the  $S_t$  plots in Section 3.2.1 and the  $ES_t$  plots in Section 3.2.2. Since  $S_t$  does not consider the region  $t > 0$ , the corresponding entries are marked “-” to indicate “empty”.  $\gamma$  is used only when computing  $ES_t$ . For points that contain no correct samples,  $\eta$  is initialized to 0.  $\alpha$  and  $\beta$  are initialized to 1 if no correct samples exist.

$t$	$\alpha$	$\beta$	$\lambda$	$\eta$	$S(t)$	$\gamma$	$ES(t)$
-10	1.000	1.000	0.000	0.000	0.100	0.100	0.100
-9	1.000	1.000	0.000	0.000	0.100	0.100	0.100
-8	1.000	1.000	0.000	0.000	0.100	0.100	0.100
-7	1.441	1.000	0.009	0.000	0.103	0.100	0.103
-6	1.321	0.858	0.575	0.049	0.442	0.100	0.442
-5	1.284	0.878	0.953	0.050	1.139	0.100	1.139
-4	1.282	0.878	0.962	0.049	1.165	0.100	1.165
-3	1.278	0.878	0.991	0.048	1.248	0.100	1.248
-2	1.278	0.878	0.991	0.048	1.248	0.100	1.248
-1	1.278	0.878	0.991	0.048	1.248	0.100	1.248
0	1.278	0.878	0.991	0.048	1.248	0.100	1.248
1	1.278	0.878	0.991	0.048	-	0.100	1.248
2	1.278	0.878	0.991	0.048	-	0.100	1.248
3	1.278	0.878	0.991	0.048	-	1.000	1.276
4	1.278	0.878	0.991	0.048	-	1.000	1.276

Table 3: Values for PaddlePaddle-NLP

$t$	$\alpha$	$\beta$	$\lambda$	$\eta$	$S(t)$	$\gamma$	$ES(t)$
-10	0.981	0.927	0.010	0.800	0.102	0.100	0.102
-9	1.038	0.927	0.013	0.615	0.103	0.100	0.103
-8	1.170	0.840	0.029	0.321	0.107	0.100	0.107
-7	1.175	0.840	0.030	0.310	0.108	0.100	0.108
-6	1.344	0.822	0.423	0.064	0.300	0.100	0.300
-5	1.342	0.817	0.930	0.046	1.118	0.100	1.118
-4	1.345	0.817	0.933	0.046	1.129	0.100	1.129
-3	1.348	0.817	0.941	0.045	1.156	0.100	1.156
-2	1.354	0.818	0.963	0.045	1.229	0.100	1.229
-1	1.360	0.818	0.975	0.045	1.275	0.100	1.275
0	1.363	0.818	0.977	0.044	1.284	0.100	1.284
1	1.363	0.818	0.977	0.044	-	0.100	1.284
2	1.363	0.818	0.977	0.044	-	0.100	1.284
3	1.363	0.818	0.977	0.044	-	1.000	1.353
4	1.363	0.818	0.977	0.044	-	1.000	1.353

Table 4: Values for PyTorch-NLP

$t$	$\alpha$	$\beta$	$\lambda$	$\eta$	$S(t)$	$\gamma$	$ES(t)$
-10	1.056	0.917	0.054	0.739	0.114	0.100	0.114
-9	1.056	0.917	0.054	0.739	0.114	0.100	0.114
-8	1.056	0.917	0.054	0.739	0.114	0.100	0.114
-7	1.056	0.920	0.080	0.529	0.121	0.100	0.121
-6	1.117	0.906	0.228	0.216	0.173	0.100	0.173
-5	1.137	0.906	0.526	0.129	0.359	0.100	0.359
-4	1.131	0.915	0.732	0.109	0.591	0.100	0.591
-3	1.125	0.912	0.892	0.105	0.866	0.100	0.866
-2	1.123	0.911	0.939	0.105	0.969	0.100	0.969
-1	1.122	0.911	0.965	0.105	1.031	0.100	1.031
0	1.121	0.910	0.981	0.105	1.072	0.100	1.072
1	1.122	0.910	0.993	0.104	-	1.000	1.121
2	1.122	0.910	0.993	0.104	-	1.000	1.123
3	1.122	0.910	0.993	0.104	-	1.000	1.123
4	1.122	0.910	0.993	0.104	-	1.000	1.123

Table 5: Values for PaddlePaddle-CV

$t$	$\alpha$	$\beta$	$\lambda$	$\eta$	$S(t)$	$\gamma$	$ES(t)$
-10	0.949	0.825	0.345	0.618	0.217	0.100	0.217
-9	0.954	0.829	0.404	0.623	0.249	0.100	0.249
-8	0.957	0.821	0.447	0.609	0.274	0.100	0.274
-7	0.984	0.819	0.605	0.552	0.399	0.100	0.399
-6	1.006	0.814	0.766	0.505	0.586	0.100	0.586
-5	1.030	0.815	0.882	0.472	0.783	0.100	0.783
-4	1.028	0.815	0.886	0.474	0.788	0.100	0.788
-3	1.028	0.815	0.886	0.474	0.788	0.100	0.788
-2	1.028	0.815	0.887	0.473	0.790	0.100	0.790
-1	1.028	0.815	0.887	0.473	0.790	0.100	0.790
0	1.028	0.815	0.887	0.473	0.790	0.100	0.790
1	1.028	0.815	0.887	0.473	-	0.100	0.790
2	1.028	0.815	0.887	0.473	-	0.100	0.790
3	1.028	0.815	0.887	0.473	-	1.000	1.025
4	1.028	0.815	0.887	0.473	-	1.000	1.025

Table 6: Values for PyTorch-CV