Step-TP: A Grounded, Step-Level Dataset with Chain-of-Thought Reasoning for LLM-Guided Tensor Program Optimization

Mengfan Liu
The University of Hong Kong
Hong Kong, China
ml621@connect.hku.hk

Da Zheng
Ant Group
Hangzhou, China
zhengda.zheng@antgroup.com

Junwei Su*
University of Science and Technology of China
Hefei, China
junweisu.cs@gmail.com

Chuan Wu*
The University of Hong Kong
Hong Kong, China
cwu@cs.hku.hk

Abstract

Despite the strong reasoning capabilities of large language models (LLMs), optimizing the execution efficiency of tensor programs remains challenging due to the need for precise, composable transformation decisions. Recent LLM-guided approaches frame tensor program optimization as an iterative decision process, but existing datasets provide only end-to-end optimized program pairs using token-inefficient representations, lacking verifiable step-level supervision and interpretability. As a result, LLMs struggle to make reliable single-step decisions in large combinatorial optimization spaces. We introduce Step-TP, a post-training dataset for tensor program optimization that provides grounded, atomic, step-level supervision with structured chain-of-thought (CoT) reasoning. Step-TP forms a closed reasoning loop over intermediate program states, enabling reliable multi-step optimization rather than outcome imitation. Its design is guided by four principles: (i) a token-efficient, verifiable intermediate representation (IR) that deterministically lowers to TVM TIR; (ii) atomic and composable optimization strategies that decompose complex trajectories into interpretable single-step decisions; (iii) structured CoT supervision coupled with explicit IR-to-IR state transitions; and (iv) strategy filtering to balance coverage while preventing shortcut exploitation. The dataset and implementation are available at a GitHub link https://github.com/LIUMENGFAN-gif/StepTP.

CCS Concepts

• **Computing methodologies** → **Artificial intelligence**.

Keywords

LLM-based Tensor Program Optimization, Tensor Program, Dataset, Chain-of-Thought Reasoning

*Corresponding authors.

KDD '26, Jeju Island, Republic of Korea

ACM ISBN 979-8-4007-2259-2/2026/08
https://doi.org/10.1145/3770855.3817484

ACM Reference Format:
Mengfan Liu, Da Zheng, Junwei Su, and Chuan Wu. 2026. Step-TP: A Grounded, Step-Level Dataset with Chain-of-Thought Reasoning for LLM-Guided Tensor Program Optimization. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2 (KDD '26), August 09–13, 2026, Jeju Island, Republic of Korea.* ACM, Jeju, Korea, 12 pages. https://doi.org/10.1145/3770855.3817484

1 Introduction

Background and Motivation. Efficient execution of deep neural networks on GPUs is fundamentally a problem of *tensor program optimization*, in which high-level mathematical operators must be lowered into kernels that effectively exploit the GPU's massive parallelism and hierarchical memory system [1, 37, 38, 44, 51, 55]. Although modern GPUs provide extraordinary peak throughput, realizing this performance in practice requires carefully coordinating computation, data movement, and parallel scheduling through techniques such as loop tiling, memory hierarchy selection, operator fusion, and thread–block binding. A s model architectures become deeper, wider, and increasingly irregular, vanilla kernel implementations often lead to poor hardware utilization and excessive memory traffic, making performance highly sensitive to low-level code generation decisions [6, 17, 20, 22, 33, 36]. This challenge is further amplified in contemporary workloads involving large language models (LLMs) and foundation models [2, 25], where even minor tensor-program-level inefficiencies accumulate into substantial increases in end-to-end latency, energy consumption, and deployment cost at scale. Consequently, efficient GPU execution has emerged as a central enabling factor for scalable, cost-effective, and sustainable deep learning, motivating the development of principled and automated tensor optimization techniques that can systematically reason about GPU execution behavior.

Motivated by these challenges, recent work has explored the use of LLMs as decision-makers in tensor program optimization, leveraging their capacity to reason over structured program representations and long-range dependencies in optimization sequences [4, 12, 16, 24, 26, 32, 35, 43, 47]. Tensor optimization entails navigating a vast, discrete, and highly nonconvex search space composed of different-level decisions (e.g., graph, operator, memory, math levels), where effective strategies often depend jointly on local program structure and global execution context [3, 7, 14, 18, 18, 39, 44, 53, 54, 56]. LLMs are particularly well suited to this setting because

they naturally model optimization as a sequential decision process, integrating symbolic program information with high-level optimization intent to produce interpretable, step-wise transformations [4, 8, 12, 16, 35]. In contrast to black-box search methods or purely statistical cost models, LLM-based approaches offer the potential to incorporate domain knowledge, generalize across workloads and hardware backends, and reuse learned optimization patterns across programs. As tensor programs and accelerator architectures continue to grow in complexity, LLM-guided optimization provides a promising direction toward more flexible and data-efficient tensor optimization frameworks.

Pressing Need for Post-Training Datasets. Despite recent progress, the practical effectiveness of LLM-based tensor optimization remains constrained without targeted post-training on domain-specific data [8, 9, 43]. Although general-purpose LLMs exhibit strong reasoning and pattern recognition capabilities, they are not trained to manipulate low-level tensor programs under the strict correctness, hardware, and performance constraints required by real-world GPU execution. In the absence of post-training data that explicitly encodes these requirements, LLMs tend to fall back on superficial pattern matching or outcome imitation, resulting in brittle behavior, poor generalization, and limited interpretability [5, 43, 45, 47]. Meanwhile, reinforcement learning or search-based fine-tuning approaches are prohibitively expensive in this domain due to the high cost of compiling and benchmarking candidate programs [4, 24, 34]. Consequently, a carefully constructed dataset for post-training—one that provides grounded intermediate program states, atomic optimization actions, and verifiable state transitions—becomes essential for enabling data-efficient learning, stable reasoning, and systematic evaluation. Such datasets are critical for transforming LLMs from coarse heuristic generators into reliable optimization agents capable of reasoning about tensor programs at scale.

Desired Properties of Post-Training Datasets. To be effective, such post-training datasets must do more than simply collect optimized programs—they must be explicitly designed to support step-wise reasoning and iterative decision making in tensor optimization. Tensor optimization is inherently an iterative decision process rather than a single-shot prediction task: it requires precise, step-wise reasoning over program representations, where each transformation must be valid, composable, and compatible with downstream steps [7, 15, 19, 40–42, 49–51]. Accordingly, effective post-training for LLM-based tensor optimization demands datasets that (i) expose *step-level supervision* instead of only final outcomes, (ii) provide *faithfully grounded reasoning traces* aligned with executable program transformations, and (iii) cover a *diverse set of optimization strategies* representative of real-world workloads. In addition, as the inference and reasoning capacity of LLMs is constrained by context length, post-training datasets should (iv) encode optimization processes in a *context-efficient representation* that supports effective reasoning within practical prompt-length limits.

Limitations of Existing Datasets. Despite growing interest in this direction, existing datasets fail to jointly satisfy the desired requirements in several important respects. *First*, most existing datasets rely primarily on outcome-only supervision [28, 46–48, 52], providing only final high-performance tensor programs produced through complex compositions of multiple optimization strategies. Such supervision encourages LLMs to memorize surface patterns in optimized code rather than to internalize the underlying decision-making process, leading to weak reasoning ability, poor generalization to unseen programs, and limited capacity to explore novel strategy compositions. *Second*, existing datasets such as ConCuR [21] adopt low-level CUDA or Python code as the primary representation space for optimization. While expressive, these representations are verbose and poorly suited for compactly encoding optimization intent and intermediate program states, resulting in excessively long descriptions that hinder effective reasoning within the limited context length of LLMs [5, 23, 27]. *Third*, existing datasets (e.g., LOOPerSet [28], IR-OptSet [46], and ConCuR [21]) exhibit limited strategy diversity, focusing mainly on easily modularized transformations such as loop tiling while largely omitting more sophisticated mathematical optimizations (e.g., online softmax [29]). Because strategy diversity directly determines the effective optimization search space, this narrow coverage constrains an LLM's ability to reason about, compose, and generalize high-performance solutions for real-world tensor programs [10, 11, 13, 18, 44, 54]. Together, these limitations highlight a critical need for constructing datasets for LLM-based tensor program optimization that are *representation-efficient*, support *grounded, step-level supervision*, and enable *reliable reasoning over a diverse set of optimization strategies*.

Contributions. To address these limitations, we introduce *Step-TP*, a post-training dataset for LLM-based tensor program optimization that provides grounded, atomic, step-level supervision with structured chain-of-thought (CoT) reasoning across diverse tensor-program–level optimization strategies. A comparison between *Step-TP* and existing datasets is shown in Table 1. The main contributions of this paper are twofold:

(1) **Design of an Effective Intermediate Representation (IR).** We propose LEIR, a *verifiable and token-efficient intermediate representation* tailored for LLM-based tensor optimization. The LEIR supports compact and precise expression of intermediate program states, enables seamless application of atomic optimization strategies, and can be deterministically converted to TVM TIR, providing semantic grounding and correctness verification.
(2) **Construction of a Step-Level Post-Training Dataset.** Building on the proposed IR, we construct *Step-TP*, a post-training dataset for LLM-based tensor optimization that incorporates: (i) a systematic decomposition of complex optimization trajectories into *atomic, composable strategies*, transforming a large and opaque search space into interpretable single-step decisions; (ii) *structured CoT supervision* that couples strategy-level rationale with explicit IR-to-IR state-transition mappings; and (iii) a *strategy filtering mechanism* based on preconditions, parameters, and synthesis depth to balance strategy distribution, ensure broad coverage, and prevent shortcut exploitation.

We further conduct an extensive empirical study of *multi-step optimization via step-level guidance*. Our results show that *Step-TP* enables effective step-level guidance, empowering diverse search paradigms to achieve strong performance with remarkable efficiency. Our results demonstrate that this guidance allows models to generate executable, grounded transformations across a diverse set of strategies and can navigate long-horizon optimization trajectories across various GPU architectures.

Dataset	Task	Target Platform	Executable IR/Program	CoT	Strategy-Driven	Step-level Supervision
LOOPerSet [28]	Polyhedral compiler optimization	CPU/GPU	×	×	×	×
TenSet [52]	Cost model	CPU/GPU	○	×	×	×
Tlp [48]	Cost model	CPU/GPU	○	×	×	×
TpuGraphs [31]	Cost model	TPU	○	×	×	×
IR-OptSet [46]	Tensor program optimization	CPU	✓	×	×	×
ConCur [21]	Tensor program optimization	GPU	✓	✓	×	×
Step-TP (Ours)	Tensor program optimization	GPU	✓	✓	✓	✓

Table 1: Representative Datasets for Tensor Programs. Our dataset Step-TP is the only post-training dataset for LLM-based tensor program optimization that provides grounded, atomic, step-level supervision with structured CoT across diverse optimization strategies.

: loop part : equation part
green: memory type; pink: loop notation; **red**: loop range;
purple: data type; **sky blue**: loop index;
orange: element-wise computation; **black**: variable

```
extern "C" __global__ void __launch_bounds__(719) main_kernel(double* __restrict__ A,
double* __restrict__ C, double* __restrict__ D) {
  for (int a = 0; a < 549; ++a) {
    for (int c = 0; c < 2252; ++c) {
      for (int d = 0; d < 2193; ++d) {
        if (d == 0) {
          D[((((int)threadIdx.x) * 1236348) + (a * 2252)) + c)] = 0.0;
        }
        D[((((int)threadIdx.x) * 1236348) + (a * 2252)) + c)] +=
          (A[((((int)threadIdx.x) * 1203957) + (a * 2193)) + d)] * C[(((d * 2252) + c)]);}}}}
```

(a) Core part of CUDA

```
def main(A: T.Buffer((719, 549, 2193), "float64"), C: T.Buffer((2193, 2252), "float64"),
D: T.Buffer((719, 549, 2252), "float64")):
    T.func_attr({"tir.noalias": True})
    # with T.block("root"):
    for tx in T.thread_binding(719, thread="threadIdx.x"):
      for a,c,d in T.grid(549,2252,2193):
        with T.block("D"):
          v_tx,v_c,v_a,v_d=T.axis.remap("SSSR",[tx,c,a,d])
          T.reads(A[v_tx,v_a,v_d],C[v_d,v_c])
          T.writes(D[v_tx,v_a,v_c])
          with T.init():
            D[v_tx,v_a,v_c]=T.float64(0)
          D[v_tx,v_a,v_c]=D[v_tx,v_a,v_c]+A[v_tx,v_a,v_d]*C[v_d,v_c]
```

(b) TIR

```
B^{719}_{tx=0}L^{549}_{a=0}L^{2252}_{c=0}L^{2193}_{d=0}
[D^{f64,g}_{tx,a,c}=D^{f64,g}_{tx,a,c}+A^{f64,g}_{tx,a,d}*C^{f64,g}_{d,c};];
```

(c) Our LEIR

Figure 1: Comparison of the same tensor program representation among CUDA, TVM TIR, and LEIR. As illustrated, our LEIR provides a more efficient representation for matrix multiplication ($D = D + A \times C$) and its associated loop structure.

2 Design of Intermediate Representation (IR)

This section presents the design of our IR for tensor-program-level transformations. We begin by examining why existing program representations are ill-suited for learning and reasoning about transformation logic, and distill a key structural insight from this analysis. This insight motivates a *high-density loop–equation* representation that covers full tensor program optimization space, which we formalize as LEIR and illustrate through a concrete matrix multiplication case study.

2.1 Limitations of Existing IR.

The reasoning capability of LLMs is constrained by finite context length, making representation efficiency a first-order concern. To enable effective modeling of program transformation logic, a tensor program representation should therefore be semantically dense and minimize entanglement with transformation-irrelevant details.

Mainstream representations such as CUDA and TVM TIR [15] are executable and compiler-oriented, but they introduce substantial syntactic and structural noise—such as type annotations, control-flow scaffolding, and compiler-mandated boilerplate—that obscure the core transformation logic (e.g., loop restructuring and algebraic fusion). As a result, LLMs are overwhelmed with implementation artifacts that are orthogonal to tensor-program-level reasoning. We illustrate these limitations using a matrix multiplication example.

CUDA. As an explicit, hardware-oriented imperative model, CUDA prioritizes fine-grained control over GPU execution, thereby causing high-level transformation logic to be scattered across fragmented, implementation-specific constructs. As illustrated in Figure 1(a), explicit type annotations in loop indices (e.g., `int a`, `((int)threadIdx.x)`) embed formatting details within the loops and index arithmetic. Meanwhile, manual initialization via control flow (e.g., `if (d == 0)`) structurally separates the initialization of a reduction from its accumulation update. Although executed within the same loop nest, this separation breaks the structural coherence of the reduction, complicating the identification of the canonical matrix multiplication pattern (*e.g.*, $D = D + A \times C$) as a unified transformation unit for learning-based models. Furthermore, CUDA typically entangles the computation with micro-architectural designs such as memory bank-conflict avoidance. For instance, padding shared memory introduces intricate index offsets that obscure the logical iteration space, thereby introducing optimization concerns that are orthogonal to tensor-program-level transformations.

TVM TIR. Compared to CUDA, TIR offers a more structured representation aligned with tensor-program-level optimizations. However, as a compiler-oriented IR, TIR imposes a heavy burden of declarative boilerplate. As shown in Figure 1(b), even a standard matrix multiplication is encased within extensive metadata (e.g., `T.reads`) and explicit axis-remapping mechanisms (e.g., `T.axis.remap`), which introduce substantial repetition without adding new semantic value to the algebraic computation. Moreover, the core $D = D + A \times C$ logic is buried under multiple layers of syntactic scaffolding, such as the nested `T.block` and `T.init` scopes. These constructs, while essential for compiler correctness, create a high degree of structural depth that weakens the visibility of the fundamental transformation intent for learning-based models.

Therefore, neither CUDA nor TIR provides an efficient format for constructing transformation datasets.

2.2 Loop-Equation IR (LEIR).

To address these limitations, we examine the essential tensor-program-level structure common to both CUDA and TIR. After abstracting

away low-level execution details and compiler boilerplate, both representations reduce to two irreducible semantic components:

(1) a *loop structure* defining the iteration space, and
(2) an *equation structure* specifying the algebraic computation at each iteration,

where these two components still cover the full tensor-program-level optimization space detailed in Appendix A, in contrast to prior abstractions (e.g., EINNET [53]) that employ non-unified loop representations (hindering operator-level transformations like loop binding) and are restricted to summation-based computations.

This observation motivates an IR that preserves only these two components, yielding a representation that is both semantically dense and suitable for step-level learning.

LEIR design principles. Building on this insight, we propose LEIR, a high-density representation that balances structural parsimony with the fidelity required to capture tensor-program-level transformations. Our design is guided by three core principles:

(1) **Irreducible semantic minimality.** Unlike existing IRs that mandate extensive metadata for compiler analysis, LEIR distills the program representation into its minimal semantic components. It consolidates the fragmented constructs of CUDA and the multi-layered scaffolding of TIR into just two irreducible structures (i.e., nested loops and algebraic equations), thereby significantly enhancing the semantic density. This design ensures that the majority of tokens in the representation correspond directly to meaningful elements of the optimization space, rather than to syntactic overhead.
(2) **Explicit organization logic.** Explicit organization logic. To maintain expressiveness, LEIR avoids the pitfall of excessive abstraction, such as representing programs solely as optimization parameters without the entire organizational logic of the computation. Instead, it explicitly preserves the structural hierarchy of execution. Specifically, the sequential order of loop descriptors and equations captures the execution flow. Meanwhile, the mapping of iteration spaces to logical execution levels (e.g., thread-block binding) is embedded within loop descriptors, and the assignment of data to memory hierarchies is encoded into tensor variables. This design ensures that the underlying computational pattern remains intact and reconstructible, allowing the learning-based models to reason about the spatial and temporal organization of the computation.
(3) **Parseable syntax.** LEIR adopts a LaTeX-based syntax to represent the tensor programs, which leverages the prior knowledge of LLMs to enable direct parsing and reasoning. Together, these principles enable LEIR to provide a concise yet expressive enough representation to capture tensor-program-level optimizations, while remaining interpretable by LLMs.

Case Study. As illustrated in Figure 1(c), we exemplify the design of LEIR through a matrix multiplication case. A typical tensor program in LEIR consists of one or more expressions separated by semicolons, with each expression comprising a loop part (in yellow) and an equation part (in green). To ensure brevity, LEIR employs implicit initialization to maintain a concise algebraic flow.

Loop structure. The loop structure is represented by a main symbol indicating the loop type, with superscripts and subscripts specifying the loop index and iteration range. LEIR supports various loop types: serial loops (L), parallel loops (P), vectorized loops (V), unrolled loops (U), and thread/block-binding loops (B). Notably, indices for binding loops are mapped to CUDA intrinsics: $\{bx, by, bz\}$ for `blockIdx` and $\{tx, ty, tz\}$ for `threadIdx`. In this case, $B^{719}_{tx=0}$ denotes the outermost loop bound to `threadIdx.x` with a range of 719, while $L^{549}_{a=0}$ represents a nested serial loop with a range of 549.

Equation structure. The equation part specifies the computation performed under the given loop nest and consists of three elements: element-wise computation, tensor variables, and delimiters to define the computational scope and logical sequence.

(1) *Element-wise computation.* We formalize the algebraic computations using a set of functional operators derived from TVM TIR, including standard arithmetic (e.g., +, −), transcendental functions (e.g., exp, log), and conditional intrinsics (e.g., if_then_else), all applied in a purely element-wise manner. The example illustrates the core matric multiplication operation: $D = D + A \times C$.

(2) *Variable.* Each variable is defined by an identity symbol (e.g., D), with subscripts for indices and superscripts for metadata. The metadata includes the data type (e.g., $f64$ for float64) and memory hierarchy (e.g., g for global, s for shared, l for local memory). For example, $D^{f64,g}_{tx,a,c}$ denotes a double-precision tensor variable stored in global memory.

(3) *Delimiters.* Two types of delimiters are employed to organize the program structure. Specifically, the square brackets ([and]) in the example bind the computation logic to the four-level loop nest. The internal semicolon (;) marks the completion of the expression, ensuring a clear logical sequence for operations.

Implicit Initialization. To maintain an uninterrupted algebraic flow, LEIR employs implicit initialization for common reduction patterns. The identity element is automatically inferred from the operator: summations default to 0, products to 1, and extremes (max/min) to $\pm\infty$. Consequently, the accumulator $D^{f64,g}_{tx,a,c}$ is initialized to 0 without requiring explicit code, maintaining a concise algebraic representation.

3 Dataset Construction

To construct a high-quality dataset for grounded, step-level supervision of tensor program transformation with reliable reasoning, we design a multi-stage pipeline as shown in Figure 2. Four stages are included: (i). The *PyTorch-to-LEIR Translator* captures the computation in a PyTorch program and then converts it into our LEIR. (ii). The *One-step Strategy-driven transformation* stage derives the applicable strategy set for each IR, applies strategies individually to perform single-step transformations, and generates a corresponding reasoning trace for each transformed LEIR to explain the optimization rationale. (iii). The transformed LEIR is then lowered to TVM TIR via an *LEIR-to-TIR Translator*, enabling execution with a mature compiler backend. (iv). Finally, the *Verification and Filtering* stage validates the correctness and semantic equivalence of transformed LEIRs, and applies a designed filtering mechanism to regulate the strategy distribution in the final dataset. While the translators provide the necessary infrastructure, the Transformation and Filtering stages constitute the core mechanisms, ensuring the reliability of reasoning traces and the data quality.

This section is organized as follows: Sec. 3.1 outlines the overarching dataset composition; Sec. 3.2 details the transformation stage;

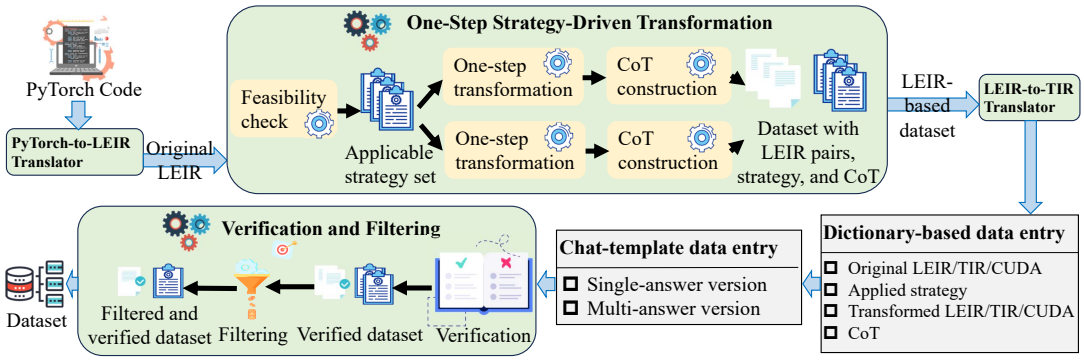


Figure 2: Pipeline of dataset construction.

Sec. 3.3 describes the specific data formats utilized for archival and training; and Sec. 3.4 covers verification and filtering mechanisms.

3.1 Dataset Composition

The dataset comprises source PyTorch programs constructed through a hierarchical approach. We first establish two fundamental building blocks: (1). single-operator programs, mainly based on KernelBench [30] level-1 dataset, covering computational backbones (e.g., matrix multiplication), nonlinear activations, lightweight operations (e.g., transpose), normalization and pooling, and common loss functions. (2) popular architectures, such as Matrix Multiplication and Softmax pipelines and attention modules (e.g., multi-head attention, multi-group attention). Based on these building blocks, we further construct composite programs by sampling and assembling 2-5 components from the aforementioned categories, covering the majority of KernelBench [30] level-2 programs and additional randomly composed cases. To further enhance data diversity, we randomize the input and output shapes (e.g., with dimension sizes up to 16,384), and vary data types (e.g., float16, float32, and float64). By instantiating the 189 distinct program types with these varied configurations, we ultimately produce a diverse dataset comprising 6,335 unique PyTorch programs, providing comprehensive coverage of representative tensor computation workloads.

Based on the dataset composition, we employ a PyTorch-to-LEIR translator to convert these PyTorch programs into our LEIR. Since PyTorch operations abstract away low-level execution details (e.g., loop nesting), we align the underlying program structures of our IR with the corresponding vanilla TVM TIR implementations, ensuring correctness, semantic equivalence, and executability.

3.2 One-step Strategy-driven Transformation

This subsection details the one-step strategy-driven transformation stage, which shifts from traditional end-to-end mapping to step-level supervision with reliable reasoning. Two benefits are included: (i) step-level supervision guides the LLM through individual transformation, reducing learning difficulty and enabling generalization to different combinations of optimizations; (ii) the strategy-driven design ensures the independence and composability of each transformation strategy, avoiding the lack of interpretability in end-to-end optimizations. To implement this stage, we employ *a three-phase transformation process*: a feasibility check of strategy preconditions, a one-step transformation to generate target LEIR, and a CoT construction to trace the transformation logic.

Feasibility Check. To ensure the validity of the generated candidate, each tensor program undergoes a feasibility check to identify applicable transformation strategies. We define nine essential preconditions, such as pattern-matching checks (e.g., identifying softmax for online softmax), with all preconditions and their corresponding strategy mappings in Appendix A. By checking these preconditions, we establish a set of feasible strategies for each LEIR.
One-step Transformation. Based on the feasible strategy set for each original LEIR, we generate one-step transformed LEIRs via a decompose-modify-reassemble workflow to control the scope of modifications and improve reproducibility. The decomposition follows the structure of our LEIR, enabling strategies at different levels (i.e., graph, operator, memory, and mathematical) to modify only the relevant components before reassembling them into a transformed LEIR. For example, the log simplification strategy works solely on the equations, while the loop reorder strategy acts only on the loops. When a strategy admits multiple valid outcomes (e.g., different loop split factors), we randomly sample one variant to enhance the diversity of transformations covered by each strategy.
CoT Construction. After applying the strategy, a corresponding reasoning trace is synthesized to formalize the underlying transformation logic. As shown in Fig. 3(a), each reasoning trace comprises two components: (i) a brief description of the applied strategy to provide a high-level semantic anchor for the transformation, and (ii) an instance-specific explanation to delineate the targeted expressions and components (e.g., loop or equation segments) and document their reassembly into the resulting modified expressions.

This structured CoT design yields several benefits for learning and interpretability. First, it improves the LLM's understanding of program representations and enhances generalization to unseen LEIRs by explicitly constructing the reasoning with the underlying LEIR structure. Second, it facilitates the activation of tensor program optimization knowledge acquired during pretraining by presenting transformations in a strategy-centric and interpretable form. Third, the reasoning traces are directly derived from the actual transformation process rather than post-hoc rationales, and can be explicitly mapped to different-level modifications, ensuring full transparency and traceability.

After transformation, the LEIR-to-TIR translator maps both the original and transformed LEIRs into TVM TIRs, leveraging the TVM backend to generate executable CUDA kernels for end-to-end performance evaluation.

Give the following IR of Matmul3D:
'B^{719}_{tx=0}L^{549}_{a=0}L^{2252}_{c=0}L^{2193}_{d=0}[D^{f64,g}_{tx,a,c}=D^{f64,g}_{tx,a,c}+A^{f64,g}_{tx,a,d}*C^{f64,g}_{d,c};];', where the known variables are 'A' with the dtype torch.float64 and shape [719, 549, 2193], 'C' with the dtype torch.float64 and shape [2193, 2252], and 'D' with the dtype torch.float64 and shape [719, 549, 2252]. Do not change the names, shapes or dtypes of these known variables in the IR.
Please give me a **numerically equivalent transformed** IR that produces exactly the same outputs for any floating-point inputs (bitwise identical), and also provide applied strategy in this transformed IR.
Return the answer **only** as a valid JSON object with the following keys:'transformed_IR', 'applied_strategies'.

<think>The strategy loop reorder is used on the given IR to rearrange the nesting orders of loops within a loop nest. Under the given IR, the loops 'B^{719}_{tx=0}L^{549}_{a=0}L^{2252}_{c=0}L^{2193}_{d=0}' in the expression 'B^{719}_{tx=0}L^{549}_{a=0}L^{2252}_{c=0}L^{2193}_{d=0}[D^{f64,g}_{tx,a,c}=D^{f64,g}_{tx,a,c}+A^{f64,g}_{tx,a,d}*C^{f64,g}_{d,c};];' can be reordered as the loops 'B^{719}_{tx=0}L^{2252}_{c=0}L^{549}_{a=0}L^{2193}_{d=0}' in the expression 'B^{719}_{tx=0}L^{2252}_{c=0}L^{549}_{a=0}L^{2193}_{d=0}[D^{f64,g}_{tx,a,c}=D^{f64,g}_{tx,a,c}+A^{f64,g}_{tx,a,d}*C^{f64,g}{d,c};];'.</think>
<answer>{'transformed_IR': 'B^{719}_{tx=0}L^{2252}_{c=0}L^{549}_{a=0}L^{2193}_{d=0}[D^{f64,g}_{tx,a,c}=D^{f64,g}_{tx,a,c}+A^{f64,g}_{tx,a,d}*C^{f64,g}_{d,c};];', 'applied_strategy': 'loop_reorder'}</answer>

(a) Single-answer

Give the following IR of Matmul3D:
...(the same IR information as (a))...
Please give me at least 2 different **numerically equivalent transformed** IRs that produces exactly the same outputs for any floating-point inputs (bitwise identical), and also provide applied strategy for each transformed IR.
Return the answer list **only** as a valid JSON object, and each entry with the following keys: 'idx', 'transformed_IR', 'applied_strategies'.

<think>These **2 transformed IRs** can be individually analyzed as follows:
0. ...(analysis removed due to space limitation)...
1. ...(analysis removed due to space limitation)...</think>
<answer>[{'idx': 0, 'transformed_IR': 'B^{719}_{tx=0}L^{9}_{h=0}L^{2}_{i=0}L^{61}_{a=0}L^{1126}_{c=0}L^{2193}_{d=0}[D^{f64,g}_{tx,h*61+a,i*1126+c}=D^{f64,g}_{tx,h*61+a,i*1126+c}+A^{f64,g}_{tx,h*61+a,d}*C^{f64,g}_{d,i*1126+c};];', 'applied_strategy': 'loop_tiling'},
{'idx': 1, 'transformed_IR': 'B^{719}_{tx=0}L^{549}_{a=0}L^{4}_{h=0}L^{563}_{c=0}L^{2193}_{d=0}[D^{f64,g}_{tx,a,h*563+c}=D^{f64,g}_{tx,a,h*563+c}+A^{f64,g}_{tx,a,d}*C^{f64,g}_{d,h*563+c};];', 'applied_strategy': 'loop_split'}]</answer>

(b) Multiple-answer

Figure 3: Single-answer and multi-answer examples for Matmul3D in the chat-template dataset.

3.3 Dataset Format

After the IR-to-TIR translation, we organize our data into two distinct formats: a comprehensive *dictionary-based* repository for archival purposes and a *chat-template* dataset for model training.

Dictionary-based Dataset. In this format, each data entry contains the original and transformed versions of the LEIR, TIR, and CUDA code, along with the applied strategy and the corresponding CoT. While this dictionary supports various program representations for future research, this paper mainly focuses on our LEIR.

Chat-template Dataset. Based on the dictionary repository, we construct a chat-template dataset, containing two specialized variants for training: (1).*Single-answer format* pairs an original LEIR with one specific transformation; (2). *Multi-answer format* incorporates a randomized subset of multiple transformations for an original LEIR. This multi-answer design not only encourages diverse optimization reasoning but also accommodates various multi-step optimization scenarios, such as providing multiple candidates for node selection in beam search.

As illustrated in Figure 3, both variants follow a standardized prompt-label architecture. The prompt integrates the original LEIR, essential metadata (e.g., program name, input/output shapes and data types), task specifications, and format requirements. The label comprises the CoT trace and the final answer. Specifically, in the multiple-answer variant, the CoT summarizes the number of transformed IRs and provides a numbered reasoning trace for each.

3.4 Verification and Filtering Mechanism

To guarantee dataset correctness and maintain a balanced strategy distribution, we implement a two-fold pipeline consisting of empirical verification and strategy-difficulty-aware filtering.

Verification. To ensure the reliability of our dataset, we subject all original and transformed LEIRs to a rigorous verification process. For each program, we execute three independent trials using randomized input tensors and compare the outputs against the baseline. Only programs that exhibit consistent numerical equivalence across all trials are retained for the final dataset.

Filtering. While the verification ensures functional correctness, it does not guarantee a high-quality distribution of transformation patterns. Without explicit control over strategy distribution, LLMs tend to disproportionately favor simplistic transformations. This bias stems from an asymmetry between simple and complex strategies. Simple strategies (e.g., log simplification) are frequently encountered during pre-training and can be activated with minimal supervision, resulting in lower predictive entropy and higher generation confidence. In contrast, complex strategies (e.g., loop split with index remapping and range recalculation) exhibit higher variability and structural diversity, which increases predictive uncertainty and causes the model to systematically avoid them. To mitigate this bias and encourage the LLMs to master sophisticated reasoning, we implement a difficulty-aware rebalancing approach.

Strategy difficulty formulation. To operationalize this rebalancing, we formalize the difficulty of each strategy along three dimensions:

- Preconditions (K): the number of essential constraints identified during the feasibility check in Sec. 3.2;
- Parameter modification (P): the number of existing components modified in the original IR, covering six aspects (i.e., expressions, loop axes, range adjustments, equations, variables, and index calculation segments);
- Synthesis depth (S): the number of unique categories of newly introduced elements (including new variables, index calculations, and expressions). Each category contributes a fixed value of 1 to the depth score.

These three dimensions reflect an ascending complexity, ranging from static constraint checking (K) and structural modification (P) to the generative synthesis of new logic (S). Therefore, the final difficulty score is formulated as:

$$\text{difficulty score} = 0.1K + 0.5(P-1) + S \tag{1}$$

where $P-1$ accounts for the baseline modification inherent in any transformation (e.g., selecting the target expression).

Strategy balancing. Based on the calculated scores, we categorize strategies into three levels and apply differentiated filtering, detailed in Appendix A. For *easy* strategies (score<1), we remove all instances from the multiple-answer dataset, whereas for the single-answer version, we retain only 20% of simplification-oriented strategies (e.g., log simplification) and a mere 4% of their inverse expansion counterpart (e.g., expand log simplification). These different ratios are based on our observation that LLMs can typically generalize to expansion tasks after mastering the corresponding

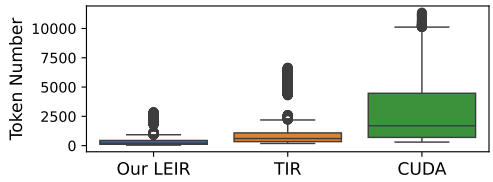


Figure 4: Token consumption of 6335 tensor programs across LEIR, TIR, and CUDA.

simplification logic. For *medium* strategies (1 ≤score< 2.5), we solely cap their occurrences at 2,000 in the multi-answer version. Finally, all *difficult* strategies (score ≥ 2.5) are fully retained to maximize the LLM's exposure to complex optimization logic.

Dataset Summary. Following the verification and filtering stages, the 35,878 entries initially generated during the strategy-driven phase were refined to a final collection of 24,953 high-quality samples. The curated dataset includes 7,537 single-answer instances and 17,416 multi-answer instances. In the final distribution (treating each strategy in multi-answer samples independently), difficult strategies account for the majority at 87.32%, while medium and easy strategies constitute 12.17%and 0.51%, respectively. This composition prioritizes high-complexity reasoning while retaining a minimal baseline to consolidate and activate the model's existing knowledge of fundamental optimizations.

4 Evaluation

In this section, we present an experimental study of LEIR and the Step-TP dataset. Some technical details are deferred to the appendix. The goal of the experimental study is to empirically validate and answer the following main questions for our design.

1. Can LEIR achieve better token efficiency than CUDA and TIR?
2. Can Step-TP enable executable, grounded transformations across a diverse set of strategies?
3. Can Step-TP support long-horizon optimization and exhibit generality across different GPUs?

Overview. The results answer these questions affirmatively, validating the effectiveness of LEIR and Step-TP.

4.1 Experimental Setup

Testbed. All experiments are conducted on a machine with 1536GB of host memory and eight NVIDIA H20-3e (140GB memory each). Unless otherwise specified, this platform serves as the default environment. To further evaluate hardware adaptability, we also perform some evaluations on a machine with eight NVIDIA A100 GPUs (80GB memory each) and 800GB of host memory.

Models. The experiments are conducted on a range of Qwen3 models with different parameter scales, including Qwen3-1.7B, Qwen3-8B, Qwen3-14B, and Qwen3-32B. The training setups are detailed in Appendix B

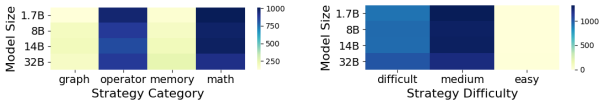
4.2 Token Efficiency

To evaluate the context-efficiency of different IRs, we compare the token consumption of 6335 tensor programs across our LEIR, TVM TIR, and CUDA using the Qwen3 model's AutoTokenizer. As illustrated in Figure 4, LEIR consistently exhibits the highest structural density. Specifically, the mean token count for LEIR is 499.3, which is significantly lower than that of TVM TIR (1244.2) and CUDA (2897.3). The disparity is even more pronounced at the upper tail of the distribution: while the longest CUDA kernel consumes over 11,000 tokens and risks exhausting the effective context window of many models, the maximum length for the equivalent LEIR representation remains under 2,900 tokens.

These results demonstrate that LEIR reduces the average token footprint by approximately 60% compared to TIR and 83% compared to CUDA. This efficiency ensures that complex optimization processes can be encoded within prompt-length limits, allowing the model to focus its reasoning capacity on strategy selection rather than parsing redundant syntax.

Model	Diff Pass	Build Pass	Exec Pass	Equal Pass
Qwen3-1.7B	99%	87%	87%	73%
Qwen3-8B	100%	95%	94%	88%
Qwen3-14B	100%	96%	96%	90%
Qwen3-32B	100%	96%	95%	92%

Table 2: Single-attempt pass rates on single-step tensor program transformations for models trained on StepTP.



(a) Category (b) Dfficulty

Figure 5: Number of strategies applied by LLMs of different sizes across 2248 test cases, grouped by (a) strategy category and (b) difficulty level.

4.3 Single-step Transformations

We evaluate models trained on our dataset via single-step transformations, highlighting two key dataset properties: (i) enabling faithfully grounded, executable program transformations, and (ii) preliminarily supporting a diverse set of tensor-program-level strategies.

Setup. We evaluate all models trained on our dataset across 2248 test cases, constructed from 180 distinct tensor programs by varying input/output data types and shapes. All test cases are strictly held out from training. Each model is prompted using our single-answer format, requiring it to autonomously select a valid transformation.

Metrics. We employ four progressively stricter metrics to evaluate the transformations, where each success is counted only after passing three independent verification trials to ensure stability: (i) Different pass, to ensure the generated LEIR is syntactically modified; (ii) Build pass, to confirm the transformed LEIR is compilable into CUDA; (iii) Execute pass, to verify the successful execution; and (iv) Equal pass, to validate equivalence with the original program.

Result Analysis. Table 2 demonstrates that models fine-tuned on our Step-TP dataset exhibit exceptional fidelity to executable and equivalent program transformations. Specifically, all models achieve near-perfect different pass rates, with Qwen3-32B reaching a 92% equal pass rate and even the 1.7B model achieving 73%. The consistently high success rates in build and execute passes suggest that our reasoning traces effectively guide the models to maintain functional correctness during complex IR modifications. To further examine the effect of repeated sampling, we also evaluate Step-TP with more independent attempts in Appendix B. Qwen3-8B trained on our dataset improves from 88.08% equal Pass with a single

Method	Avg. # Samples	Max. # Samples	Avg. Speedup	Median Speedup	Max. Speedup	Search Efficiency
Greedy Search	35.91	41	20.79	1.78	193.55	0.58
Beam Search	103.17	114	42.90	4.57	561.82	0.60
BFS Search	28.27	31	18.27	2.11	173.13	0.65
DFS Search	28.11	31	20.97	1.68	242.62	0.75
MCTS	17.41	21	24.96	4.54	171.9	1.43
Chain Search	16.63	21	23.62	2.05	286.92	1.42
Chain Search wo Parent	15.95	21	20.25	4.96	96.14	1.63
Chain Search on A100	16.85	21	22.01	3.62	265.35	1.31

Table 3: Performance comparison and search efficiency of various search algorithms guided by Qwen3-32B trained on Step-TP.

Method	Avg. # Strategy	Max. # Strategy
Greedy Search	3.6	8
Beam Search	5.23	10
BFS Search	2.45	4
DFS Search	2.49	4
MCTS	3.65	9
Chain Search	3.57	7
Chain Search wo Parent	3.43	7
Chain Search on A100	3.33	9

Table 4: Number of strategies of multi-step optimization trajectories guided by Qwen3-32B trained on Step-TP.

attempt to 95.55% with 10 attempts and 96.22% with 16 attempts. This underscores the superior quality and robust generalization of the Step-TP dataset.

We further analyze the diversity of strategies autonomously selected by the models. As shown in Figure 5(a), the category-wise distribution (graph, operator, memory, math) closely mirrors the underlying dataset ratio (8:9:5:21), confirming that models successfully cover all strategy categories proportionally. As shown in Figure 5(b), easy strategies occur least frequently, and medium and difficult strategies appear at comparable ratesindicating that the models do not avoid difficult strategies in favor of simpler alternatives. Moreover, we observe that the 32B model utilizes 31 unique strategies at all levels, while other scales cover 29, representing over 70% of the 43 total available strategies. This diverse coverage, aligned with our data distribution, confirms that the models have successfully mastered a broad spectrum of representative optimization patterns.

4.4 Multi-step Optimizations

We evaluate models trained on our dataset via multi-step transformations, highlighting three key dataset properties: (i) supporting long-horizon complex optimization by combining multiple strategies step by step, (ii) enabling efficient search through high-quality step-level supervision, and (iii) exhibiting generality across different GPU environments. We provide a detailed case study of the highest-performing test case in Appendix B.

Setup. We evaluate Qwen3-32B, trained on Step-TP, across 100 distinct tensor programs strictly held out from training. The model is tasked with generating runtime-performance-optimized, equivalent LEIRs. To implement multi-step optimization, we deploy seven distinct search algorithms: Greedy Search, Breadth-First Search (BFS), Depth-First Search (DFS), Beam Search, Monte Carlo Tree Search (MCTS), and Chain-based Search (with/without parent nodes). The detailed setups are provided in Appendix B.

Metrics. We evaluate results using four metrics: (i) # Samples: the total number of candidate LEIRs verified during the entire optimization process; (ii) Speedup: the runtime improvement of the transformed LEIR relative to the original LEIR , computed as $\text{Runtime}_{\text{original}}/\text{Runtime}_{\text{transformed}}$; (iii) # Strategies: the count of distinct strategy types applied along the final optimization trajectory, where each type is counted once regardless of different applications ; and (iv) Search efficiency: the average speedup achieved per verified sample, calculated as Average Speedup/# Samples.

Result Analysis. We evaluate the ability of our dataset Step-TP across all search algorithms. Table 3 details the number of samples, the speedup, and the search efficiency, while Table 4 tracks the structural complexity of the optimization trajectories via the number of strategies. The results illustrate key properties:

(i) Supporting Long-horizon Complex Optimization. As shown in Table 3, across all search algorithms, the model trained on Step-TP consistently achieves substantial performance improvements across all search paradigms. For instance, Beam Search achieves an average speedup of 42.90× and a peak speedup of 561.82×. Notably, even under the most restrictive search budgets (e.g., Chain-based Search without parent nodes), the model maintains a high median speedup of 4.96×. These results indicate performance gains are not driven by isolated outliers but demonstrate that our dataset captures high-quality optimization patterns, enabling algorithms to successfully compose sequences of transformations.

Table 4 further reveals that optimal schedules require an average of 2.45 to 5.23 distinct strategy types, with a maximum depth of 10 (e.g., in Beam Search). This demonstrates the support of our dataset for diverse tensor-program-level strategies, which serve as the essential building blocks for composing these optimal trajectories.

Together, these observations confirm that our dataset provides the necessary structural knowledge to navigate non-trivial, long-horizon optimization landscapes.

(ii) Enabling Efficient Search. As shown in Table 3, the search efficiency ranging from 0.58 to 1.63 indicates that the majority of verified transformations contribute meaningfully to the final speedup. In particular, even search algorithms without backtracking or branching techniques achieve significant performance. For instance, Greedy Search and Chain-based Search variants achieve average speedups of 20.25× to 25.97× while requiring as few as 16 to 36 samples. The success of these short-insight paradigms suggests that Step-TP provides high-quality step-level supervision that enables the model to effectively identify high-potential transformation paths, reducing the cost of exhaustive trial and error.

(iii) Exhibiting generality across different GPUs. As shown in Tables 3 and 4, Chain-based Search on A100 GPU maintains a strong speedup performance and complex trajectories up to 9 strategies, extending beyond the default H20-3e GPUs. This confirms that

Dataset Variant	Equal Pass	Difficult:Medium:Easy strategy Ratio
F TIR wo CoT	77%	42:51:7
F LEIR wo CoT	83%	44:50:6
UF LEIR with CoT	88%	25:28:47
Step-TP	88%	42:51:7

Table 5: Single-step transformation ablation results for Qwen3-8B trained on four dataset variants on 2248 test cases.

Dataset Variant	Avg. #Samples	Avg. Speedup	Search Efficiency
F TIR wo CoT	8.10	0.72	0.08
F LEIR w/o CoT	7.31	0.75	0.10
UF LEIR with CoT	13.66	8.14	0.59
Step-TP	13.88	12.10	0.87

Table 6: Multi-step optimization ablation results for Qwen3-8B trained on four dataset variants on 100 distinct test cases using chain-based search with parent nodes.

the optimization knowledge captured by Step-TP remains effective across different GPU generations.

4.5 Ablation Study

We conduct ablation studies to examine how three key Step-TP dataset designs affect the performance of trained models: the LEIR representation, structured CoT supervision, and strategy filtering.
Setup. Four controlled dataset variants are constructed from the same source programs and transformation pipeline. Specifically, we train Qwen3-8B on: (i) TensorIR-based data without CoT but with strategy filtering (*F TIR wo CoT*); (ii) LEIR-based data without CoT but with strategy filtering (*F LEIR wo CoT*); (iii) LEIR-based data with CoT but without strategy filtering (*UF LEIR with CoT*); and (iv) the full Step-TP dataset with all three components enabled.
Result Analysis. We compare the four dataset variants from two perspectives: single-step transformations, which measure transformation fidelity and strategy difficulty, and multi-step optimization, which measures optimization effectiveness and search efficiency. Table 5 reports single-step transformation performance in terms of Equal Pass and the difficulty distribution of the applied strategies, while Table 6 reports the average number of verified samples, average speedup, and search efficiency during multi-step optimization. These results lead to the following observations:

(i) LEIR reduces representation-induced transformation errors. As shown in Table 5, replacing TensorIR with LEIR improves equal pass from 77% to 83% when CoT is removed and strategy filtering is kept. This suggests that LEIR helps the model generate semantically equivalent transformations by exposing the relevant loop and equation structures more directly, instead of requiring the model to reason through compiler-oriented TensorIR boilerplate.

(ii) Structured CoT is critical for long-horizon optimization. In single-step transformation, Table 5 shows that adding CoT improves equal pass from 83% to 88%, indicating better semantic preservation. In multi-step optimization, Table 6 shows a much larger improvement, with average speedup increasing from 0.75 to 12.10 and search efficiency from 0.10 to 0.87. This suggests that structured CoT helps the model learn composable transformation logic, enabling effective optimization trajectories beyond locally valid IR edits.

(iii) Strategy filtering reduces the bias toward simplistic transformations. As shown in Table 5, adding strategy filtering keeps equal pass unchanged at 88%, but changes the generated strategy distribution from 25:28:47 to 42:51:7 for difficult, medium, and easy strategies, respectively. This indicates that filtering suppresses the model's preference for easy transformations without sacrificing single-step correctness. This shift becomes more meaningful in multi-step optimization, where Table 6 shows higher average speedup from 8.14 to 12.10 and higher search efficiency from 0.59 to 0.87.

5 Conclusion

We introduce Step-TP, a step-level post-training dataset for LLM-based tensor program optimization that provides verifiable, compositional supervision for single-step transformation decisions. By combining a token-efficient intermediate representation (LEIR) with atomic strategy decomposition and deterministic equivalence checking, Step-TP enables models to reason about precise optimization steps, rather than relying on outcome-only shortcuts.

ACKNOWLEDGEMENT

This work was supported in part by a collaborative research grant from Ant Group and grants from Hong Kong RGC under the contracts 17204423, 17205824, 17204625, C7004-22G (CRF), CRS_PolyU501/23, and T43-513/23-N (TRS).

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.
[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).
[3] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 193–205.
[4] Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. 2025. Kevin: Multi-Turn RL for Generating CUDA Kernels. *arXiv preprint arXiv:2507.11948* (2025).
[5] Tyler A Chang and Benjamin K Bergen. 2024. Language model behavior: A comprehensive survey. *Computational Linguistics* 50, 1 (2024), 293–350.
[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.
[7] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems* 31 (2018).
[8] Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, et al. 2023. Large language models for compiler optimization. *arXiv preprint arXiv:2309.07062* (2023).
[9] Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. 2024. Meta large language model compiler: Foundation models of compiler optimization. *arXiv preprint arXiv:2407.02524* (2024).
[10] Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691* (2023).
[11] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems* 35 (2022), 16344–16359.

[12] Juncheng Dong, Yang Yang, Tao Liu, Yang Wang, Feng Qi, Vahid Tarokh, Kaushik Rangadurai, and Shuang Yang. 2025. Stark: Strategic team of agents for refining kernels. *arXiv preprint arXiv:2510.16996* (2025).

[13] Jingzhi Fang, Yanyan Shen, Yue Wang, and Lei Chen. 2021. ETO: Accelerating optimization of DNN operators by high-performance tensor program reuse. *Proceedings of the VLDB Endowment* 15, 2 (2021), 183–195.

[14] Pratik Fegade, Tianqi Chen, Phillip B Gibbons, and Todd C Mowry. 2024. ACRoBat: Optimizing auto-batching of dynamic deep learning at compile time. *Proceedings of Machine Learning and Systems* 6 (2024), 14–30.

[15] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, et al. 2023. Tensorir: An abstraction for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 804–817.

[16] Junfeng Gong, Zhiyi Wei, Junying Chen, Cheng Liu, and Huawei Li. 2025. From large to small: Transferring cuda optimization expertise via reasoning graph. *arXiv preprint arXiv:2510.19873* (2025).

[17] Hanpeng Hu, Junwei Su, Juntao Zhao, Yanghua Peng, Yibo Zhu, Haibin Lin, and Chuan Wu. 2024. CDMPP: A device-model agnostic framework for latency prediction of tensor programs. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 1054–1074.

[18] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 47–62.

[19] Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2019. Optimizing DNN computation with relaxed graph substitutions. *Proceedings of Machine Learning and Systems* 1 (2019), 27–39.

[20] Hyeonjin Kim, Sungwoo Ahn, Yunho Oh, Bogil Kim, Won Woo Ro, and William J Song. 2020. Duplo: Lifting redundant memory accesses of deep neural networks for gpu tensor cores. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 725–737.

[21] Lingcheng Kong, Jiateng Wei, Hanzhang Shen, and Huan Wang. 2025. Concur: Conciseness makes state-of-the-art kernel generation. *arXiv preprint arXiv:2510.07356* (2025).

[22] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. 2022. Automatic horizontal fusion for GPU kernels. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 14–27.

[23] Tianle Li, Ge Zhang, Quy Duc Do, Xiang Yue, and Wenhu Chen. 2024. Long-context llms struggle with long in-context learning. *arXiv preprint arXiv:2404.02060* (2024).

[24] Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. 2025. Cuda-l1: Improving cuda optimization via contrastive reinforcement learning. *arXiv preprint arXiv:2507.14111* (2025).

[25] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[26] Mengfan Liu, Wei Wang, and Chuan Wu. 2025. Optimizing distributed deployment of mixture-of-experts model inference in serverless computing. In *Ieee infocom 2025-ieee conference on computer communications*. IEEE, 1–10.

[27] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the association for computational linguistics* 12 (2024), 157–173.

[28] Massinissa Merouani, Afif Boudaoud, and Riyadh Baghdadi. 2025. Looperset: A large-scale dataset for data-driven polyhedral compiler optimization. *arXiv preprint arXiv:2510.10209* (2025).

[29] Maxim Milakov and Natalia Gimelshein. 2018. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867* (2018).

[30] Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. 2025. KernelBench: Can LLMs write efficient GPU kernels?, 2025. *URL https://arxiv. or g/abs/2502.10517* (2025).

[31] Mangpo Phothilimthana, Sami Abu-El-Haija, Kaidi Cao, Bahare Fatemi, Michael Burrows, Charith Mendis, and Bryan Perozzi. 2023. Tpugraphs: A performance prediction dataset on large tensor computational graphs. *Advances in Neural Information Processing Systems* 36 (2023), 70355–70375.

[32] Guicheng Qi, Junwei Su, Liqi Yang, Tao Li, Tingwen Xie, Yerui Sun, Yuchen Xie, and Chuan Wu. 2026. HetAuto: Cross-Cluster Auto-Parallelism for Heterogeneous Distributed Training. In *Proceedings of the 21st European Conference on Computer Systems*. 759–779.

[33] Daniel Snider and Ruofan Liang. 2023. Operator fusion in XLA: analysis and evaluation. *arXiv preprint arXiv:2301.13062* (2023).

[34] Songqiao Su, Xiaofei Sun, Xiaoya Li, Albert Wang, Jiwei Li, and Chris Shum. 2025. CUDA-L2: Surpassing cuBLAS Performance for Matrix Multiplication through Reinforcement Learning. *arXiv preprint arXiv:2512.02551* (2025).

[35] Annabelle Sujun Tang, Christopher Priebe, Rohan Mahapatra, Lianhui Qin, and Hadi Esmaeilzadeh. 2025. REASONING COMPILER: LLM-Guided Optimizations for Efficient Model Serving. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

[36] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 10–19.

[37] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[38] Vasily Volkov and James W Demmel. 2008. Benchmarking GPUs to tune dense linear algebra. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE, 1–11.

[39] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. 2021. {PET}: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 37–54.

[40] Lei Wang, Yu Cheng, Yining Shi, Zhengju Tang, Zhiwen Mo, Wenhao Xie, Lingxiao Ma, Yuqing Xia, Jilong Xue, Fan Yang, et al. 2025. TileLang: A Composable Tiled Programming Model for AI Systems. *arXiv preprint arXiv:2504.17577* (2025).

[41] Lei Wang, Lingxiao Ma, Shijie Cao, Quanlu Zhang, Jilong Xue, Yining Shi, Ningxin Zheng, Ziming Miao, Fan Yang, Ting Cao, et al. 2024. Ladder: Enabling efficient {Low-Precision} deep learning computing through hardware-aware tensor transformation. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 307–323.

[42] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Zhijao Jia, Dheevatsa Mudigere, Ying Zhang, Anthony Kewitsch, and Manya Ghobadi. 2022. Topoopt: Optimizing the network topology for distributed dnn training. *arXiv preprint arXiv:2202.00433* (2022).

[43] Jiin Woo, Shaowei Zhu, Allen Nie, Zhen Jia, Yida Wang, and Youngsuk Park. 2025. Tritonrl: Training llms to think and code triton without cheating. *arXiv preprint arXiv:2510.17891* (2025).

[44] Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Man Kit Ao, Praveen Velliengiri, Xupeng Miao, Oded Padon, and Zhihao Jia. 2025. Mirage: A {Multi-Level} superoptimizer for tensor programs. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 21–38.

[45] Haofeng Xu, Junwei Su, Yukun Tian, Lansong Diao, Zhengping Qian, and Chuan Wu. 2026. GAC: Stabilizing Asynchronous RL Training for LLMs via Gradient Alignment Control. *arXiv preprint arXiv:2603.01501* (2026).

[46] Zi Yang, Lei Qiu, Fang Lyu, Ming Zhong, Zhilei Chai, Haojie Zhou, Huimin Cui, and Xiaobing Feng. [n. d.]. IR-OptSet: An Optimization-Sensitive Dataset for Advancing LLM-Based IR Optimizer. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

[47] Yi Zhai, Sijia Yang, Keyu Pan, Renwei Zhang, Shuo Liu, Chao Liu, Zichun Ye, Jianmin Ji, Jie Zhao, Yu Zhang, et al. 2024. Enabling Tensor Language Model to Assist in Generating {High-Performance} Tensor Programs for Deep Learning. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 289–305.

[48] Yi Zhai, Yu Zhang, Shuo Liu, Xiaomeng Chu, Jie Peng, Jianmin Ji, and Yanyong Zhang. 2023. Tlp: A deep learning-based cost model for tensor program tuning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 833–845.

[49] Jie Zhao, Xiong Gao, Ruijie Xia, Zhaochuang Zhang, Deshi Chen, Lei Chen, Renwei Zhang, Zhen Geng, Bin Cheng, and Xuefeng Jin. 2022. Apollo: Automatic partition-based operator fusion through layer by layer optimization. *Proceedings of Machine Learning and Systems* 4 (2022), 1–19.

[50] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, et al. 2021. AKG: automatic kernel generation for neural processing units using polyhedral transformations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1233–1248.

[51] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 863–879.

[52] Lianmin Zheng, Ruochen Liu, Junru Shao, Tianqi Chen, Joseph E Gonzalez, Ion Stoica, and Ameer Haj Ali. 2021. Tenset: A large-scale program performance dataset for learned tensor compilers. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

[53] Liyan Zheng, Haojie Wang, Jidong Zhai, Muyan Hu, Zixuan Ma, Tuowei Wang, Shuhong Huang, Xupeng Miao, Shizhi Tang, Kezhao Huang, et al. 2023. {EINNET}: Optimizing tensor programs with {Derivation-Based} transformations. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 739–755.

[54] Size Zheng, Siyuan Chen, Siyuan Gao, Liancheng Jia, Guangyu Sun, Runsheng Wang, and Yun Liang. 2023. Tileflow: A framework for modeling fusion dataflow via tree-based analysis. In *Proceedings of the 56th Annual IEEE/ACM International*

Symposium on Microarchitecture. 1271–1288.
[55] Yuchen Zhong, Junwei Su, Chuan Wu, and Minjie Wang. 2025. Heta: Distributed Training of Heterogeneous Graph Neural Networks. *Proceedings of the VLDB Endowment* 18, 9 (2025), 2790–2803.
[56] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, et al. 2022. {ROLLER}: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 233–248.

A Strategy

A.1 Tensor-Program-Level Strategy Categories

Four Categories of Strategies. Based on the semantic dimension primarily modified by a strategy, we categorize tensor-program-level optimizations into four classes:

(1). Graph level. These optimizations act on the organization of multiple expressions. An example is the expression reorder strategy, where independent expressions are permuted without violating data dependencies.

(2). Operator level. These strategies target the loop–equation structure within a single expression. For example, the loop reorder strategy swaps the nesting order of LEIR iterators, such as transforming $B^{719}_{tx=0}L^{549}_{a=0}$ into $L^{549}_{a=0}B^{719}_{tx=0}$.

(3). Memory level. These optimizations primarily alter the logical storage and layouts of tensor variables. the storage scope strategy rebinds a tensor $E^{f64,g}_a$ in global memory to a local memory scope $E^{f64,l}_a$ to optimize data proximity.

(4). Mathematical level. These strategies directly rewrite algebraic formulations to change execution logic while preserving numerical semantics. For example, the online softmax strategy reformulates the global exponential sum into incremental update equations, decomposing a monolithic reduction into recursive algebraic steps within a loop.

In total, we have identified 43 distinct strategies across these levels, comprising 8 graph-level, 9 operator-level, 5 memory-level, and 21 mathematical-level strategies, detailed in Sec A.3.

A.2 Preconditions for Strategy Filtering

Given that strategies are constrained by specific program structures, we define nine essential preconditions as follows: (1). The pattern match check detects a specific computation pattern (e.g., softmax computation for online softmax strategy); (2). The dependency check verifies the existence of computation-order dependencies among expressions (e.g., for expression reorder strategy); (3). The operation identity check ensures that repeated computation operations exist (e.g., for common subexpression elimination); (4). The loop nest consistency check ensures that expressions share the same loop structure (e.g., for operator fusion strategy); (5). The equation count check ensures a sufficient number of equations exist (e.g., for operator fission strategy); (6). The loop axis count check verifies that the number of loop axes is sufficient (e.g., for loop reorder strategy); (7). The loop range factorization check ensures that the selected loop range can be split appropriately (e.g., for loop split strategy); (8). The reduction axis check prevents applying illegal strategies (e.g., loop binding) to reduction axes; (9). The intermediate variable check validates whether the tensor variables are inputs or outputs (e.g., for set storage scope strategy).

A.3 Strategies and their Difficulty

- Easy: operator fission, factorization, expand factorization, cancellation, expand cancellation, apart, together, powsimp, expand powsimp, logsimp, expand log, collect, expand collect;
- Medium: operator fusion, compute inline, expression splitting, expression reorder, loop reorder, loop unrolling, loop parallelization, loop vectorization, loop binding, exponential split, ultiplicative split, additive split;
- Dfficult:tensor concat to fuse operators, tensor split to decouple operators, common subexpression elimination, loop tiling, loop split, loop fusion, reduction factorization, cache read write, layout transformation, set storage scope , set storage layout, precompute indices, partially equivalent then correct, normal loop max to prefix max, normal loop summation on exp to prefix summation on exp, online softmax, flashattention wo tiling, normal matmul to prefix matmul based on online softmax;

B Evaluation

Training. All models are fine-tuned using LoRA with a rank of 8, LoRA alpha set to 32, and a dropout rate of 0.05. LoRA adapters are applied to all linear layers. We use a learning rate of $1e-4$, a weight decay of 0.1, and a warmup ratio of 0.05. The batch size is set to 64 for Qwen3-1.7B and 16 for the larger models due to memory constraints. All models are trained for 3 epochs. LoRA fine-tuning is implemented using the ms-swift framework.

B.1 Single-step Transformation

Setup. The temperature is set to 0 during inference. Crucially, we do not specify any strategies, requiring the model to autonomously select a valid transformation.

B.2 Multi-step Transformation

Setup. The temperature is set to 0.3 during inference. For accurate performance measurement, we employ TVM's built-in `time_evaluator` to record execution time. Specifically, each kernel is executed for three warmup runs to mitigate transient hardware effects, followed by three measurement repetitions to ensure statistical stability, with the average latency reported.

The model is tasked with generating mathematically equivalent, runtime-performance-optimized IRs to achieve higher speedups. Each test case prompt provides six key pieces of information: (i) the search algorithm, (ii) the current LEIR and related metadata, (iii) the exploration history if required by the search algorithm, with at most one parent node, (iv) the target GPU specifications, (v) all 43 potential strategies, and (vi) the task description. An example prompt is provided as follows:

"Breadth-first-based optimization is used on a given IR to improve performance. Each IR is a state, and has a parent transformation and speedup performance.

Give the current IR of Gemm Swish Divide Clamp Tanh Clamp: $B^{728}_{tx=0}B^{1243}_{bxa=0}L^{2022}_{c=0}[C^{f32,g}_{tx,bxa} = C^{f32,g}_{tx,bxa} + A^{f32,g}_{tx,c} * J^{f32,g}_{bxa,c};]; B^{728}_{tx=0}L^{1243}_{a=0}[C^{f32,g}_{tx,a} = C^{f32,g}_{tx,a} + K^{f32,g}_{a};]; B^{728}_{tx=0}L^{1243}_{a=0}[F^{f32,g}_{tx,a} = 0.5 * C^{f32,g}_{tx,a} / (1 + exp(-C^{f32,g}_{tx,a}))];]; B^{728}_{tx=0}L^{1243}_{a=0}[G^{f32,g}_{tx,a} = min(max(F^{f32,g}_{tx,a}, -1.0), 1.0);]; B^{728}_{tx=0}L^{1243}_{a=0}$

$[M^{f32,g}_{tx,a} = exp(G^{f32,g}_{tx,a}) - exp(-G^{f32,g}_{tx,a});]; B^{728}_{tx=0}L^{1243}_{a=0}[N^{f32,g}_{tx,a} = exp(G^{f32,g}_{tx,a}) + exp(-G^{f32,g}_{tx,a});]; B^{728}_{tx=0}L^{1243}_{a=0}[H^{f32,g}_{tx,a} = M^{f32,g}_{tx,a}/N^{f32,g}_{tx,a};]; B^{728}_{tx=0}L^{1243}_{a=0}[I^{f32,g}_{tx,a} = min(max(H^{f32,g}_{tx,a}, -1.0), 1.0);];$, where the known variables are 'A' with the dtype torch.float32 and shape [728, 2022], 'J' with the dtype torch.float32 and shape [1243, 2022], 'K' with the dtype torch.float32 and shape [1243], and 'I' with the dtype torch.float32 and shape [728, 1243]. Do not change the names, shapes or dtypes of these known variables in the IR.

History:

The parent IR: same as the root IR, depth:0, speedup value: 1.

The speedup value of the current IR: 30.358607118641192, the depth is 1, and the current IR is obtained from the parent IR using the strategy 'loop_binding'.

Target hardware: NVIDIA H20-3e GPU.

CUDA binding rules: Loop axes bound to block (along x,y,z axis, max dimension value: $2^31 - 1$, 65535, 65535) MUST be renamed with prefixes bx, by, bz and unique, respectively, followed by other unique lowercase letters. Loop axes bound to thread (along x,y,z axis, max dimension value: 1024, 1024, 64) MUST be renamed with prefixes tx, ty, tz, respectively, followed by other unique lowercase letters.

Memory usage rules: Data indexed by block-level loops may be placed in shared (s) or global (g) memory. Data indexed by thread-level loops may be placed in local (l), shared (s) or global (g) memory.

The following strategies and any other mathematical strategies can be considered: operator fusion, operator fission, compute inline, expression splitting, tensor concat to fuse operators, tensor split to decouple operators, common subexpression elimination, expression reorder, loop reorder, loop tiling, loop split, loop fusion, loop unrolling, loop parallelization, loop vectorization, loop binding, reduction factorization, cache read write, layout transformation, set storage scope, set storage layout, precompute indices, factorization, expand factorization, cancellation,expand cancellation, apart, together, powsimp, expand powsimp, expand log, logsimp, collect, expand collect, partially equivalent then correct, normal loop max to prefix max, exponential split, multiplicative split, additive split, normal loop summation on exp to prefix summation on exp, online softmax, flashattention wo tiling, normal matmul to prefix matmul based on online softmax.

Task:

Please give me at least 2 different **numerically equivalent, runtime-performance-optimized** IRs that produce exactly the same outputs for any floating-point inputs (bitwise identical) to achieve higher speedup values (should be more than 1), and also provide applied strategy for each transformed IR.

Return the answer list **only** as a valid JSON object, and each entry with the following keys: 'idx', 'transformed_IR', 'applied_strategies'.

CRITICAL:

1. Before you suggest each new transformation, you MUST identify what has been changed in the current IR compared to the root IR. For each new optimization, you MUST build it ON TOP OF these existing changes, namely ON TOP OF the current IR. The strategies MUST be used on the current IR! You MUST compare your modified parts in each transformed IR with the current IR. If they are identical strings, your answer is WRONG. If the unmodified part in each transformed IR is different from the current IR, your answer is WRONG.

2.Don't repeat the current or parent IRs! You MUST NOT revert to the parent IR: In particular, you are NOT allowed to apply any reverse or undo operation that reconstructs the current IR from its parent IR, including inverse transformations such as operator fusion <-> operator fission, loop tiling <-> loop fusion, loop split <-> loop fusion, apart <-> together, collect <-> expand collect, or similar reversals."

Search Algorithm. To implement multi-step optimization, we deploy seven distinct search algorithms categorized by their exploration strategies: (i) Greedy Search, which generates multiple candidate transformations per step and selects the locally best one; (ii) Breadth-First Search (BFS) and (iii) Depth-First Search (DFS), representing exhaustive breadth and depth explorations; (iv) Beam Search, which generates multiple transformations per step and maintains the top-k candidates as the search frontier; (v) Monte Carlo Tree Search (MCTS), which balances exploration and exploitation using rollouts and value estimation to guide transformation selection; and (vi) Chain-based Search (with/without parent nodes), which sequentially refines transformations step by step, optionally considering the parent node.

For greedy search, beam search, MCTS, and two chain-based search variants, the maximum number of iterations is set to 20. For DFS and BFS, the maximum depth is set to 20 andthe number of generated LEIRs for each node is set to 2. For greedy search, the breadth is set to 2. For beam search, the number of generated LEIRs for each node is set to 3, and the k value is set to 2 in order to maintain the top-k candidates.

Case Study. We analyze a complex tensor program featuring matrix multiplication, scaling, and residual addition(i.e., $B^{457}_{tx=0}L^{2265}_{a=0}L^{3520}_{c=0}[D^{f32,g}_{tx,a} = D^{f32,g}_{tx,a} + A^{f32,g}_{tx,c} * I^{f32,g}_{a,c};]; B^{457}_{tx=0}L^{2265}_{a=0}[D^{f32,g}_{tx,a} = D^{f32,g}_{tx,a} + J^{f32,g}_{a};]; B^{457}_{tx=0}L^{2265}_{a=0}[E^{f32,g}_{tx,a} = D^{f32,g}_{tx,a} * H^{f32,g}_{a};]; B^{457}_{tx=0}L^{2265}_{a=0}[F^{f32,g}_{tx,a} = E^{f32,g}_{tx,a} + C^{f32,g}_{tx,a};];$), achieving a 561.82× speedup through a 15-step optimization trajectory. The process begins with a layout transposition (Step 1, 8.34×) to align memory access. This is followed by intensive memory hierarchy and parallelism adjustments—including storage scope specification and loop binding—to reach 288.75× (Steps 2–6). Subsequent multi-level tiling and unrolling further optimize data locality and register pressure to 430.68× (Steps 7–12), concluding with fine-grained hardware binding to maximize GPU utilization (Steps 13–15). The final optimized LEIR is: $B^{457}_{tx=0}L^{3520}_{c=0}[M^{f32,l}_{c,tx} = A^{f32,g}_{tx,c};]; B^{457}_{tx=0}L^{3520}_{c=0}[K^{f32,g}_{c,tx} = M^{f32,l}_{c,tx};]; B^{457}_{tx=0}U^{3520}_{c=0}[N^{f32,g}_{c*457+tx} = K^{f32,g}_{c,tx};]; B^{457}_{tx=0}B^{2265}_{bxa=0}U^{3520}_{c=0}[D^{f32,s}_{tx,bxa} = D^{f32,s}_{tx,bxa} + N^{f32,g}_{c*457+tx} * I^{f32,g}_{bxa,c};]; B^{457}_{tx=0}B^{2265}_{bza=0}[D^{f32,s}_{tx,bza} = D^{f32,s}_{tx,bza} + J^{f32,g}_{bza};]; B^{457}_{tx=0}B^{2265}_{bza=0}[E^{f32,g}_{tx,bza} = D^{f32,s}_{tx,bza} * H^{f32,g}_{bza};]; B^{457}_{tx=0}B^{5}_{bxf=0}L^{453}_{a=0}[F^{f32,g}_{tx,bxf*453+a} = E^{f32,g}_{tx,bxf*453+a} + C^{f32,g}_{tx,bxf*453+a};];$.