STParOpt: An End-to-End Framework for Execution-Aware Parallelism Inference and Optimized CUDA Migration

Meng Wang[1] Feiyang Wen[1] Jinshuo Liu[1]* Weiran Pian[1] Xinya Liu[1] Juan Deng[2] Jeff Z. Pan[3]

[1]School of Cyber Science and Engineering, Wuhan University
[2]School of Computer Science, Wuhan University
[3]School of Informatics, The University of Edinburgh

Abstract

Automatically migrating serial C programs to CUDA is important for high-performance computing, yet remains challenging due to the partial observability of data dependencies: critical memory access patterns are difficult to infer from source code alone. Static analyses are often conservative on irregular and pointer-intensive loops, while LLM-based approaches lack explicit execution evidence for memory-access-dependent optimization decisions. To address these limitations, we propose **STParOpt**, an execution-aware framework that combines static program structure with runtime memory-access features for joint parallelism and optimization inference. **STParOpt** augments static program graphs with stride-level execution features via cross-attention and uses entropy-guided fusion to adaptively weight static and dynamic evidence. Experiments on standard HPC benchmarks demonstrate that **STParOpt** outperforms both compiler-based and LLM-based baselines on parallelism detection and CUDA code generation. Notably, **STParOpt** achieves **96.63 $\pm$ 0.23%** parallelism detection accuracy across three seeds and **86.1%** end-to-end generation correctness, surpassing the strongest compared baselines by **2.06 pp** and **11.4 pp**, respectively.

1 Introduction

The growing demand for GPU-accelerated computing has made automated C-to-CUDA migration a critical bottleneck in scientific and engineering workflows, where vast quantities of legacy HPC code must be ported to GPU execution to meet modern performance targets. Manual migration demands deep expertise in parallel algorithm design and GPU memory hierarchy, making automation an urgent practical need. A central obstacle is parallelism reasoning: determining which loops can be safely parallelized and which optimization strategies should govern their GPU implementation. The core difficulty is not syntactic translation but a semantic gap—loop-carried dependencies, irregular memory access patterns, and optimization strategies such as memory coalescing cannot be reliably inferred from source text alone, as loop parallelizability depends on cross-iteration dependencies induced by runtime memory access patterns that static representations abstract away.

Existing approaches address this challenge partially, each capturing one dimension of the required evidence while leaving the other unresolved. Static compiler analyses [1, 2, 3] rely on conservative dependency approximations, overestimating hazards in the presence of irregular access patterns or pointer aliasing. Learning-based methods over static program graphs [4, 5] capture structural

*Corresponding author. Emails: Meng Wang (wang_meng@whu.edu.cn); Jinshuo Liu (liujinshuo@whu.edu.cn); Jeff Z. Pan (j.z.pan@ed.ac.uk).

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

priors but remain insensitive to execution-level signals that distinguish true dependencies from abstraction artifacts. LLM-based approaches [6, 7] attempt direct CUDA generation from source text, but lack explicit dependency analysis and cannot determine optimization strategies from fixed rules—these depend on runtime access patterns that vary across loop structures and input regimes. While recent work addresses CUDA kernel optimization [8, 9], these approaches either target existing kernels or require explicit human guidance, and do not address automated C-to-CUDA migration with optimization-aware parallelism inference. These limitations reveal a core challenge: *how to construct representations that jointly capture static program structure and dynamic execution behavior to infer both loop parallelizability and optimization strategies, enabling automated C-to-CUDA migration given executable harnesses and representative inputs.* We identify three instantiations of this challenge:

C1: Dependency ambiguity in irregular loops. Static structure cannot resolve dependency ambiguity in irregular and pointer-intensive loops, where runtime access patterns determine true parallelizability.

C2: Unreliable fusion of heterogeneous signals. Static and dynamic signals have unequal reliability across loop categories, requiring adaptive reconciliation rather than fixed combination.

C3: Optimization strategy inference under partial observability. Optimization strategies depend on runtime access patterns that vary across loop structures and cannot be captured by fixed rules.

To address these challenges, we propose **STParOpt** (Spatio-Temporal Parallelism and Optimization Inference), an end-to-end execution-aware framework for automated C-to-CUDA migration. For **C1**, we introduce **STER**, combining a Multi-dimensional Program Graph (MPG) with a Stride Pattern Encoder (SPE) that injects execution-level features into MPG via cross-attention. For **C2**, an **entropy-guided fusion mechanism** adaptively weights static and dynamic signals based on predictive uncertainty. For **C3**, a **hierarchical decision framework** jointly predicts parallelism and optimization strategies, translating them into structured constraints for LLM-guided CUDA generation. Together, these challenges motivate execution-aware representations for deciding which loops to parallelize and which optimizations to apply. Entropy-guided fusion provides targeted adaptive weighting when the evidence conflicts.

Our contributions are as follows:

- We propose **STParOpt**, an end-to-end framework that jointly infers loop parallelizability and optimization strategies from execution-aware program representations and generates CUDA migration candidates given an executable harness and representative inputs.
- We combine **STER**, entropy-guided fusion, and hierarchical decision making to jointly infer parallelism and optimization strategies from structural program relations and runtime execution behavior.
- We conduct comprehensive experiments on standard HPC benchmarks, demonstrating that **STParOpt** achieves $\mathbf{96.63 \pm 0.23\%}$ parallelism detection accuracy across three seeds and **86.1%** code correctness, outperforming the strongest compared baselines by **2.06 pp** and **11.4 pp**, respectively.

2 Related Work

Program Parallelization. Polyhedral compilers [1, 2] are effective for affine loops but overestimate dependencies under pointer aliasing. Skeleton-based approaches [10, 11, 12] automate translation via pattern matching but lack optimization awareness. Dynamic profilers [13, 14, 3] improve coverage but produce input-specific observations. Learning-based performance optimization has been explored for program parallelization [15, 16, 17, 18, 19, 20, 4, 5], distributed-training control [21], and data-access planning [22]; however, these methods do not jointly model static program structure and execution traces for loop-level parallelism and CUDA optimization inference. STParOpt addresses this by jointly encoding static and dynamic evidence for parallelism and optimization inference.

LLM-based Code Generation. NMT and LLM-based approaches [7, 6, 23] translate sequential code to parallel implementations but lack explicit dependency analysis. AST-based retrieval [24] and test-driven refinement [25] improve SQL generation; CUDA still requires dependency reasoning [26] and optimization guidance [9]. CUDAMaster [8] optimizes existing kernels via profiling-guided iteration,

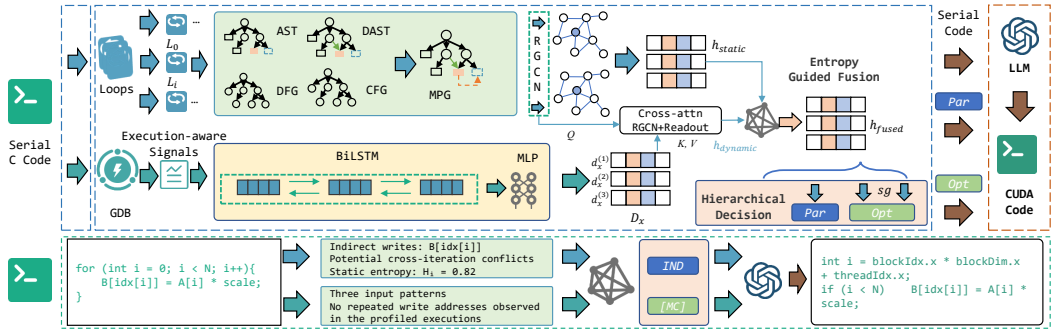

Figure 1: **STParOpt** architecture (top) and a running example (bottom). Cross-attention aggregates three pattern-specific embeddings per variable; intermediate graph processing is omitted. Entropy-guided fusion supports parallelism and optimization prediction for CUDA generation. sg marks training-only stop-gradient on the Opt input and g_i. In the bottom example, MC denotes coalesced reads of `A[i]` and `idx[i]`, not necessarily writes to `B[idx[i]]`.

targeting a complementary post-migration setting. STParOpt addresses the upstream migration stage by predicting optimization constraints from execution-aware representations and injecting them as structured specifications into the generator.

3 Method

3.1 Problem Formulation

Let C denote the input C program containing loops $\mathbb{L} = \{L_1, \ldots, L_n\}$. We formulate C-to-CUDA migration as a two-stage structured prediction problem under partial observability.

Stage 1: Semantic Inference. For each loop L_i, the model infers a parallelism type $y_i \in \{0, 1, 2\}$ and an optimization strategy $\mathbf{o}_i \in \{0, 1\}^5$ from static source code C and execution traces $\mathcal{T}$ collected under K input patterns. Inference is uncertainty-aware: the model maintains a predictive distribution $p(y_i \mid C)$ over the static branch and computes its entropy $H_i = -\sum_c p(y_i{=}c \mid C) \log p(y_i{=}c \mid C)$ as a proxy for dependency ambiguity. The two signals are reconciled via a learned gate $\alpha_i = \sigma(wH_i{+}b)$, yielding a fused representation:

$$\mathbf{h}_i^{\text{fused}} = (1 - \alpha_i)\, \mathbf{h}_i^{\text{static}} + \alpha_i\, \mathbf{h}_i^{\text{dynamic}},$$

where the gate adapts the balance using uncertainty in the static prediction. The joint prediction then operates on $\mathbf{h}_i^{\text{fused}}$:

$$p(\{y_i, \mathbf{o}_i\}_{i=1}^n \mid C, \mathcal{T}) = \prod_{i=1}^{n} p(y_i \mid \mathbf{h}_i^{\text{fused}}) \cdot p(\mathbf{o}_i \mid y_i, \mathbf{h}_i^{\text{fused}}).$$

We factorize predictions per target loop; cross-loop scheduling and data movement are outside the evaluated setting.

Stage 2: Strategy-Driven Generation. Given inferred decisions $\{y_i, \mathbf{o}_i\}$, a downstream LLM generates the CUDA program:

$$\hat{C} = \text{LLM}(C, \{y_i\}, \{\mathbf{o}_i\}),$$

where $\{y_i, \mathbf{o}_i\}$ serve as hard semantic constraints that determine the generation target, decoupling uncertainty-sensitive inference from syntax-constrained generation.

3.2 Overview

STParOpt addresses C-to-CUDA migration as a two-stage structured decision problem under partial observability, instantiated through three components (Figure 1). **STER** (§ 3.3) constructs execution-aware representations by augmenting a Multi-dimensional Program Graph with stride-level dynamic features via cross-attention. An **entropy-guided fusion** mechanism (§ 3.4) adaptively modulates

static and dynamic evidence based on predictive uncertainty, shifting reliance toward execution signals when static analysis is ambiguous. A **hierarchical decision framework** (§ 3.5) then jointly infers parallelism type and optimization strategies, translating them into structured constraints for downstream CUDA generation.

3.3 Execution-Aware Representation

Static structure alone cannot resolve dependency ambiguity in loops with complex memory access patterns, while dynamic traces without structural grounding fail to generalize across input regimes. **STER** addresses this by constructing a **Multi-dimensional Program Graph (MPG)** that encodes syntactic, data-flow, and control-flow relations, and enriching it with stride-level execution features from a **Stride Pattern Encoder (SPE)** via cross-attention, yielding a unified loop representation $\mathbf{h}_i$.

Multi-dimensional Program Graph. We construct $G_i = (\mathcal{V}_i, \mathcal{E}_i)$ for each loop L_i by parsing source code with *tree-sitter*, without requiring compilation. We partition AST nodes into statement, variable, and constant subsets with syntactic edges $\mathcal{E}_{\texttt{syn}}$, augment them with typed data-dependence edges $\mathcal{E}_{\texttt{dfg}}$ distinguishing `RAW`, `WAR`, and `WAW` hazards, and insert control nodes $\mathcal{V}_{\texttt{ctrl}}$ at loop exit and branch merge points with labeled edges $\mathcal{E}_{\texttt{ctrl}}$:

$$\mathcal{V}_i = \mathcal{V}_{\texttt{ast}} \cup \mathcal{V}_{\texttt{ctrl}}, \quad \mathcal{E}_i = \mathcal{E}_{\texttt{syn}} \cup \mathcal{E}_{\texttt{dfg}} \cup \mathcal{E}_{\texttt{ctrl}}. \tag{1}$$

This joint encoding enables the model to reason over both structural dependency patterns and control-flow constraints within a single relational representation. We assign each node v a learnable type-specific embedding $\mathbf{h}_v^{(0)} \in \mathbb{R}^{128}$ and apply a single-layer RGCN [27] to produce static node representations $\mathbf{h}_v^{\text{static}}$. We obtain a graph-level representation $\mathbf{h}_i^{\text{static}}$ via attention-based readout.

Stride Pattern Encoder. Let $\mathcal{X}_i$ denote the set of memory-accessing variables in L_i. For each variable $x \in \mathcal{X}_i$ and input pattern $k \in \{1, \ldots, K\}$, we record a memory-address sequence over $T = 50$ iterations using GDB-based instrumentation, where access expressions are automatically identified via a preprocessing step and per-iteration element addresses are captured (see Appendix A.2 for details). Crucially, we retain the input-pattern dimension rather than aggregating the $K = 3$ traces before attention. Each pattern is encoded independently by applying a BiLSTM to its stride sequence and an MLP to its pattern-specific domain features:

$$\mathbf{d}_x^{(k)} = \mathbf{W}_{\text{spe}} \left[\text{BiLSTM}(\boldsymbol{\Delta}_x^{(k)}) \,\|\, \text{MLP}(\mathbf{f}_x^{(k)}) \right] \in \mathbb{R}^d. \tag{2}$$

We stack the pattern-specific embeddings row-wise in $\mathbf{D}_x \in \mathbb{R}^{K \times d}$, whose k-th row is $(\mathbf{d}_x^{(k)})^\top$, with $K = 3$ and $d = 128$.

Cross-Attention Injection. We use multi-head cross-attention to aggregate the K pattern-specific traces associated with each MPG variable node. For each head $r \in \{1, \ldots, M\}$, the static variable representation provides one query, while the $K = 3$ rows of $\mathbf{D}_x$ provide the keys and values:

$$\mathbf{Q}_{x,r} = (\mathbf{h}_x^{\text{static}})^\top \mathbf{W}_{Q,r} \in \mathbb{R}^{1 \times d_h}, \tag{3}$$

$$\mathbf{K}_{x,r} = \mathbf{D}_x \mathbf{W}_{K,r}, \qquad \mathbf{V}_{x,r} = \mathbf{D}_x \mathbf{W}_{V,r} \in \mathbb{R}^{K \times d_h}, \tag{4}$$

$$\boldsymbol{\beta}_{x,r} = \text{softmax}_K \left(\frac{\mathbf{Q}_{x,r} \mathbf{K}_{x,r}^\top}{\sqrt{d_h}} \right) \in \mathbb{R}^{1 \times K}, \tag{5}$$

$$\mathbf{c}_{x,r} = \boldsymbol{\beta}_{x,r} \mathbf{V}_{x,r} \in \mathbb{R}^{1 \times d_h}, \tag{6}$$

$$\mathbf{h}_x^{\text{new}} = \mathbf{h}_x^{\text{static}} + \mathbf{W}_O [\mathbf{c}_{x,1} \| \cdots \| \mathbf{c}_{x,M}]^\top. \tag{7}$$

Here $M = 4$, $d_h = d/M = 32$, $\mathbf{W}_{Q,r}, \mathbf{W}_{K,r}, \mathbf{W}_{V,r} \in \mathbb{R}^{d \times d_h}$, and $\mathbf{W}_O \in \mathbb{R}^{d \times d}$. Softmax is computed across the three input patterns, not over a single key or across different variables. Each variable node therefore attends to the pattern-specific stride embeddings associated with that variable, adaptively selecting execution patterns that are most informative given its static structural context. The role of this attention differs from entropy-guided fusion: cross-attention selects and aggregates evidence *among input patterns*, whereas the graph-level gate α_i in § 3.4 controls the overall contribution of the dynamic representation relative to the static representation. We then apply a second RGCN layer to propagate enriched variable features to statement and control nodes: the first

RGCN layer encodes structural priors before injection, while the second completes the information flow from dynamic signals to dependency reasoning across the full graph. This produces dynamic representations $\{\mathbf{h}_v^{\text{dynamic}}\}$ and graph-level $\mathbf{h}_i^{\text{dynamic}}$.

3.4 Entropy-Guided Fusion

Static structure provides a global prior over dependency relations, while dynamic traces offer partial but input-sensitive observations. We use entropy-guided gating as a targeted mechanism for weighting the two representations when their evidence conflicts. The static branch's predictive uncertainty informs the learned balance; the execution-aware representation itself supplies the runtime evidence.

Entropy-Based Gating. We compute the predictive entropy of the static branch:

$$H_i = -\sum_{c \in \mathcal{Y}} p_{\text{static}}(c \mid C) \log p_{\text{static}}(c \mid C), \tag{8}$$

where higher H_i indicates greater structural ambiguity. Predictive entropy is used as a proxy for ambiguity in the static prediction. We use the static distribution because the gate is intended to adjust reliance on the source-only view; the dynamic branch also has an auxiliary classifier for training, but its entropy does not determine the gate.

We use a detached static prediction to compute H_i, preventing the fusion weight from introducing gradient leakage into the parallelism classifier. A learnable gate then modulates the contribution of dynamic evidence:

$$\alpha_i = \sigma(wH_i + b), \qquad \mathbf{h}_i^{\text{fused}} = (1 - \alpha_i)\, \mathbf{h}_i^{\text{static}} + \alpha_i\, \mathbf{h}_i^{\text{dynamic}}. \tag{9}$$

The gating parameters are learned jointly with the model, enabling adaptive calibration of modality contributions rather than fixed weighting.

When execution traces are unavailable, we set $\alpha_i = 0$, reducing the model to static-only inference. This soft gating avoids discontinuities in gradient flow that arise from hard masking, enabling stable joint optimization.

Auxiliary Supervision. Auxiliary three-class classifiers on the static and dynamic graph representations receive the same parallelism labels. These heads support branch supervision and analysis; the final parallelism decision uses the fused representation.

3.5 Hierarchical Decision and Strategy-Driven Generation

Parallelism detection and optimization selection are coupled but asymmetric: optimization strategies are only meaningful for parallelizable loops.

Parallelism Decision. Given $\mathbf{h}_i^{\text{fused}}$, we predict parallelism type $y_i \in \{0, 1, 2\}$ (independent, reduction, sequential) via:

$$p(y_i \mid \mathbf{h}_i^{\text{fused}}) = \text{softmax}(\mathbf{W}_y\, \mathbf{h}_i^{\text{fused}}), \quad y_i \in \{0\text{: independent, } 1\text{: reduction, } 2\text{: sequential}\}. \tag{10}$$

Conditional Optimization Decision. The optimization head produces five probabilities from a detached fused representation:

$$\mathbf{r}_i = \text{sigmoid}(\mathbf{W}_o\ \text{stopgrad}(\mathbf{h}_i^{\text{fused}})) \in (0, 1)^5. \tag{11}$$

During training, the sample weight $g_i = \text{stopgrad}(p_i^f[0] + p_i^f[1])$ weights each sample's binary cross-entropy before batch averaging, where p_i^f is the fused parallelism distribution above. Both the weight and the optimization head's input are detached. No ground-truth hard mask or additional class weighting is used. Binary strategy decisions $\hat{\mathbf{o}}_i$ are used for generation only for loops predicted as independent or reduction.

Training Objective. We jointly optimize all objectives via learnable loss weighting [28]:

$$\mathcal{L} = \frac{\mathcal{L}_{\text{par}}}{2\sigma_1^2} + \frac{\mathcal{L}_{\text{static}}}{2\sigma_2^2} + \frac{\mathcal{L}_{\text{dynamic}}}{2\sigma_3^2} + \lambda\, \mathcal{L}_{\text{opt}} + \sum_{j=1}^{3} \log \sigma_j, \tag{12}$$

Here $\sigma_1, \sigma_2, \sigma_3 > 0$ are learned uncertainty scales and $\lambda = 0.1$ weights optimization supervision. The three parallelism terms are batch-averaged cross-entropies; $\mathcal{L}_{\text{opt}}$ already includes the per-sample g_i. Complete reductions and gradient paths are defined in Appendix B.3.1.

Strategy-Driven Generation. Given inferred decisions $\{y_i, \hat{\mathbf{o}}_i\}$, we construct a structured prompt P_i comprising four components: a task definition P^{task}, the source C code C, the predicted parallelism type y_i as a preliminary analysis result, and a set of optimization directives P^{opt} derived from $\hat{\mathbf{o}}_i$. Formally:

$$P_i = P^{\text{task}} \parallel C \parallel P^{\text{par}}(y_i) \parallel P^{\text{opt}}(\hat{\mathbf{o}}_i), \tag{13}$$

where $P^{\text{par}}(y_i)$ encodes the predicted parallelism label as a natural language analysis result, and $P^{\text{opt}}(\hat{\mathbf{o}}_i)$ translates each active optimization dimension into a concrete strategy directive (see Appendix A.4). The CUDA program is then generated via $\hat{C} = \text{LLM}(P_i)$, treating the LLM as a conditional executor over structured semantic constraints rather than requiring it to discover optimization opportunities from source code alone [9].

4 Experiments

4.1 Dataset & Metrics

We construct a two-stage training corpus supporting both static and dynamic learning, and evaluate on two separate test sets corresponding to the two tasks.

For Loop Detection. We use the OMP_Serial dataset [2] consisting of 32,570 samples, split into train / dev / test at an 8:1:1 ratio, constructed from real-world OpenMP benchmarks with manually verified annotations. To support dynamic learning, we annotate 4,000 training samples and 500 dev samples with minimal executable implementations covering $K = 3$ input configurations. The entire test split (3,257 samples) is fully annotated. For **parallel loop detection**, we report **Accuracy** and **Macro-F1** over the three classes. For **optimization strategy prediction**, we report per-class F1 over five strategies (MC, WR, TL, SM, ST) and Macro-F1; annotation definitions and verification are in Appendix B.2. All learning-based baselines are retrained on the same dataset splits for fair comparison.

For CUDA Code Generation. We evaluate on Polybench and Rodinia, two widely used HPC benchmark suites, augmented with 30 non-parallelizable loops from the test split, totaling 79 programs spanning affine, irregular, and non-parallelizable loop categories. Each sample targets one serial loop, with any generated kernel validated independently; source applications may contain multiple stages. For **parallel code generation**, we evaluate **Compilation Rate (CR)**, **Execution Rate (ER)**, **Correctness (Co)**, **Fast** (speedup ≥ 1), and **Relative Speedup**, which jointly assess syntactic validity, executability, functional correctness, and performance gains. We report geometric-mean speedup over serial execution on correct CUDA outputs with valid timings. Relative speedup $S = T_{\text{Human}}/T_{\text{Model}}$ is used separately for the per-kernel comparison with Human CUDA (Appendix B.1).

4.2 Baselines

We compare **STParOpt** against two groups of baselines.

For **parallel loop detection**: traditional analyzers (Pluto [1], DiscoPoP [3]), learning-based methods (Graph2Par [4], AutoParLLM [5]), and LLM-based detection using Claude 4.5 Sonnet [29] and Qwen3-Coder [30] under four configurations: `-base`, `-static`, `-dynamic`, and `-s&d`.

For **CUDA code generation**: PPCG [2], neural translation systems (CodeRosetta [6], BabelTower [7], LASSI [23]), and LLM-based methods under `-gen` (direct generation) and `-det&gen` (detection-then-generation) configurations. Baseline label mapping details are provided in Appendix B.2.

4.3 Implementation Details

For **loop detection**, we use a 2-layer RGCN (hidden size 128) with a 1-layer BiLSTM (128), ReLU activation, and 0.3 dropout. We set $K = 3$, $T = 50$, and learning rate to 0.001 based on validation. Other hyperparameters (λ, w, b) are tuned on the dev set (see Appendix). STParOpt and its trainable

[2] `https://github.com/thelechen/OMP_serial_dataset`

Table 1: Main results on parallel loop detection and optimization strategy prediction. MC: memory coalescing, WR: warp reduction, TL: tiling, SM: shared memory, ST: streaming. STParOpt reports mean ± sample standard deviation over three seeds (42, 2026, 10086).

Method	Parallel Detection					Optimization Strategy Prediction					
	Acc	IND-F1	RED-F1	SEQ-F1	M-F1	Macro-F1	MC-F1	WR-F1	TL-F1	SM-F1	ST-F1
Pluto	0.646	0.609	0.679	0.654	0.647	—	—	—	—	—	—
DiscoPoP	0.718	0.641	0.877	0.703	0.740	—	—	—	—	—	—
Graph2Par	0.795	0.779	0.779	0.821	0.793	—	—	—	—	—	—
AutoParLLM	0.914	0.873	0.935	0.936	0.915	—	—	—	—	—	—
Qwen-base	0.909	0.881	0.883	0.946	0.903	0.678	0.755	0.731	0.679	0.686	0.538
Qwen-static	0.916	0.905	0.906	0.930	0.914	0.705	0.767	0.757	0.718	0.683	0.600
Qwen-dynamic	0.920	0.871	0.936	0.926	0.911	0.651	0.713	0.738	0.698	0.635	0.473
Qwen-s&d	0.931	0.931	0.943	0.924	0.933	0.731	0.779	0.787	0.723	0.750	0.615
Claude-base	0.932	0.922	0.922	0.945	0.930	0.731	0.833	0.751	0.719	0.701	0.652
Claude-static	0.942	0.944	0.942	0.939	0.942	0.766	0.865	0.765	0.747	0.685	0.766
Claude-dynamic	0.941	0.923	0.962	0.947	0.944	0.713	0.798	0.763	0.725	0.704	0.577
Claude-s&d	0.945	0.933	0.953	0.951	0.946	0.757	0.827	0.797	0.718	0.721	0.720
DeepSeek-base	0.854	0.803	0.820	0.919	0.847	0.732	0.757	0.707	0.693	0.725	0.778
DeepSeek-static	0.895	0.910	0.846	0.903	0.886	0.740	0.772	0.720	0.731	0.688	0.789
DeepSeek-dynamic	0.874	0.866	0.836	0.897	0.866	0.700	0.772	0.728	0.711	0.723	0.565
DeepSeek-s&d	0.913	0.946	0.877	0.899	0.908	0.723	0.803	0.728	0.696	0.753	0.636
GPT-4.1-base	0.905	0.886	0.907	0.921	0.904	0.709	0.805	0.750	0.698	0.713	0.577
GPT-4.1-static	0.919	0.907	0.923	0.927	0.919	0.723	0.808	0.772	0.706	0.718	0.609
GPT-4.1-dynamic	0.932	0.917	0.932	0.945	0.931	0.738	0.828	0.770	0.751	0.720	0.622
GPT-4.1-s&d	0.944	0.916	0.968	0.960	0.947	0.761	0.840	0.785	0.777	0.749	0.652
STParOpt	**0.9663** ±0.0023	**0.9592** ±0.0036	**0.9841** ±0.0011	**0.9637** ±0.0049	**0.9690** ±0.0019	**0.8405** ±0.0043	**0.8865** ±0.0029	**0.8689** ±0.0059	**0.7688** ±0.0060	**0.8402** ±0.0093	**0.8383** ±0.0106

ablations are trained with three seeds: $\{42, 2026, 10086\}$. Across-seed summaries report mean ± sample standard deviation.

For **code generation**, we use temperature 0.3; model versions and the shared inference protocol are in Appendix B.3. Experiments are conducted on a machine with RTX 4090 GPU, Intel i7-14700KF CPU, and 64GB RAM. C and CUDA codes are compiled with GCC 11.4 and NVCC 12.4.

4.4 Main Results

4.4.1 Loop Detection

We evaluate **STParOpt** against compiler-based, learning-based, and LLM-based baselines on parallel loop detection and optimization strategy prediction. Table 1 reports results across all methods.

STParOpt leads both tasks among the evaluated methods. STParOpt reaches **96.63 ± 0.23%** Acc and **96.90 ± 0.19%** M-F1 across three seeds, exceeding the strongest detection baseline in Table 1, Claude-s&d (**94.5%** Acc), by **2.13 pp**. On optimization strategy prediction, **STParOpt** achieves Macro-F1 of **84.05 ± 0.43%**, exceeding the strongest optimization baseline in Table 1, Claude-static (**76.6%**), by **7.45 pp**. Additional GPT-5.6-Sol and DeepSeek-V4-Pro comparisons are reported in Appendix C.5; the strongest added detection baseline reaches 94.57% Acc.

Static and dynamic signals are complementary. Across all four LLMs, `s&d` achieves higher detection accuracy than either `static` or `dynamic` alone, supporting the complementarity of the two signals. Optimization prediction gains vary by model and strategy: Qwen-s&d improves MC-F1, SM-F1, and WR-F1 over Qwen-base by **2.4 pp**, **6.4 pp**, and **5.6 pp**, respectively, but combined evidence does not improve every optimization metric for every model.

4.4.2 CUDA Code Generation

We evaluate all methods on three benchmarks: Polybench [3] (**30** affine loops), Rodinia [4] (**19** irregular HPC kernels), and **30** non-parallelizable loops sampled from the test split, totaling **79** samples. The overlap audit found no systematic train–generation leakage (Appendix B.1). All methods are evaluated end-to-end: given a serial C loop, each method must determine parallelizability, generate CUDA if applicable, and return C otherwise. For `det&gen` configurations, the detection step uses the `s&d` variant. Results are shown in Table 2.

[3] `https://github.com/cavazos-lab/PolyBench-ACC`

[4] `https://github.com/yuhc/gpu-rodinia`

Table 2: CUDA generation on 79 samples. CoP counts correct outputs by subset. Geomean is speedup over serial on each method's correct CUDA outputs with valid timings, excluding failures and sequential returns; timing sets may differ.

Method	Overall				CoP			Geomean (×)
	CR	ER	Co	Fast	Poly./30	Rod./19	Seq./30	
PPCG	0.759	0.747	0.734	0.684	28	0	30	1.40
CodeRosetta	0.203	0.114	0.076	0.063	6	0	0	1.31
BabelTower	0.152	0.101	0.038	0.025	3	0	0	0.88
LASSI	0.747	0.734	0.646	0.544	21	12	18	2.77
Qwen-gen	0.747	0.747	0.658	0.608	21	9	22	2.66
GPT-4.1-gen	0.709	0.709	0.671	0.620	20	10	23	3.49
Claude-gen	0.772	0.772	0.671	0.620	21	10	22	3.67
Qwen-det&gen	0.797	0.797	0.747	0.671	22	12	25	3.51
GPT-4.1-det&gen	0.772	0.772	0.747	0.696	21	13	25	3.65
Claude-det&gen	0.823	0.823	0.747	0.709	21	14	24	4.87
STParOpt-no-opt	0.899	0.873	0.848	0.772	25	16	26	5.42
STParOpt	**0.899**	**0.886**	**0.861**	**0.810**	**25**	**17**	**26**	**15.50**

Table 3: Ablations and pattern-aggregation controls on the OMP_Serial test set. Each cell reports mean (top) and sample standard deviation (bottom) over seeds 42, 2026, and 10086. **Bold** indicates the best mean in each column.

Variant	Parallel Detection					Optimization Strategy Prediction					
	Acc	IND-F1	RED-F1	SEQ-F1	M-F1	Macro-F1	MC-F1	WR-F1	TL-F1	SM-F1	ST-F1
STParOpt (full)	**0.9663** ±0.0023	**0.9592** ±0.0036	**0.9841** ±0.0011	**0.9637** ±0.0049	**0.9690** ±0.0019	**0.8405** ±0.0043	**0.8865** ±0.0029	**0.8689** ±0.0059	**0.7688** ±0.0060	**0.8402** ±0.0093	**0.8383** ±0.0106
w/o SPE	0.9155 ±0.0049	0.8711 ±0.0099	0.9309 ±0.0112	0.9444 ±0.0060	0.9155 ±0.0061	0.7662 ±0.0051	0.8185 ±0.0030	0.7818 ±0.0030	0.7183 ±0.0091	0.7768 ±0.0044	0.7356 ±0.0346
w/o Entropy Fusion	0.9505 ±0.0004	0.9397 ±0.0120	0.9770 ±0.0023	0.9467 ±0.0096	0.9545 ±0.0011	0.8072 ±0.0098	0.8703 ±0.0013	0.8518 ±0.0018	0.7434 ±0.0090	0.7968 ±0.0087	0.7739 ±0.0314
w/o Stop-Gradient	0.9555 ±0.0016	0.9469 ±0.0020	0.9776 ±0.0009	0.9519 ±0.0051	0.9588 ±0.0012	0.8161 ±0.0019	0.8819 ±0.0013	0.8592 ±0.0039	0.7550 ±0.0092	0.7958 ±0.0039	0.7884 ±0.0100
Shuffled Trace	0.9453 ±0.0047	0.9469 ±0.0004	0.9668 ±0.0026	0.9338 ±0.0110	0.9492 ±0.0037	0.7920 ±0.0027	0.8539 ±0.0034	0.8103 ±0.0050	0.7343 ±0.0075	0.7916 ±0.0025	0.7698 ±0.0222
Mean Pooling	0.9510 ±0.0008	0.9448 ±0.0048	0.9738 ±0.0051	0.9451 ±0.0030	0.9546 ±0.0011	0.8001 ±0.0153	0.8626 ±0.0006	0.8325 ±0.0225	0.7355 ±0.0144	0.7602 ±0.0250	0.8096 ±0.0468
Gated Residual	0.9545 ±0.0010	0.9463 ±0.0015	0.9748 ±0.0028	0.9514 ±0.0014	0.9575 ±0.0010	0.8031 ±0.0093	0.8671 ±0.0033	0.8357 ±0.0230	0.7434 ±0.0152	0.7660 ±0.0190	0.8032 ±0.0357
Concat + Projection	0.9594 ±0.0013	0.9512 ±0.0023	0.9821 ±0.0047	0.9552 ±0.0011	0.9628 ±0.0019	0.8143 ±0.0081	0.8684 ±0.0029	0.8483 ±0.0115	0.7599 ±0.0035	0.7863 ±0.0123	0.8087 ±0.0151

STParOpt achieves the highest correctness in Table 2. STParOpt reaches **86.1%** Co and **81.0%** Fast, outperforming the strongest LLM baseline Claude-det&gen (**74.7%** Co) by **+11.4 pp** and Claude-gen (**67.1%** Co) by **+19.0 pp**. Removing optimization directives preserves CR but reduces ER/Co by **1.3 pp** and Fast by **3.8 pp**; geometric-mean speedup falls from **15.50**× to **5.42**×.

Detection benefits generation correctness. Across all three LLMs, `det&gen` improves Co over `gen` (**+7.6–8.9 pp**), supporting the value of parallelism inference before generation. NMT-based methods (CodeRosetta, BabelTower) achieve CoP = 0 on Rodinia and cannot reject non-parallelizable loops, while **STParOpt** correctly returns serial code on **26**/30 sequential loops. Backbone-portability and additional-model results appear in Appendices C.4 and C.5.

4.5 Ablation Study

Effect of Pattern-Level Cross-Attention. Table 3 compares mean pooling, concatenation + projection, and gated residual injection, keeping SPE, downstream RGCN, and $K = 3$ embeddings fixed. These controls average the patterns, project their concatenation, or gate a mean-pooled update; cross-attention instead assigns static-conditioned weights to individual patterns. It exceeds the strongest control, concatenation + projection, by **0.69 pp** in Acc and **2.62 pp** in optimization Macro-F1.

Effect of SPE. Removing SPE reduces Acc/optimization Macro-F1 by **5.08/7.43 pp** and TL-F1/SM-F1 by **5.05/6.34 pp**, supporting stride-based memory-access encoding.

Effect of Entropy-Guided Fusion. Removing entropy-guided fusion reduces Acc/optimization Macro-F1 by **1.58/3.33 pp**, supporting adaptive evidence weighting; Section 4.6.2 examines conflicting signals.

Effect of Stop-Gradient. Removing stop-gradient reduces Acc/optimization Macro-F1 by **1.08/2.44 pp**, consistent with gradient isolation limiting cross-task interference.

Effect of Stride Sequence Ordering. Shuffling strides reduces Acc/optimization Macro-F1 by **2.10/4.85 pp**, supporting the value of temporal access coherence.

4.6 Analysis

4.6.1 Performance on Irregular Loops

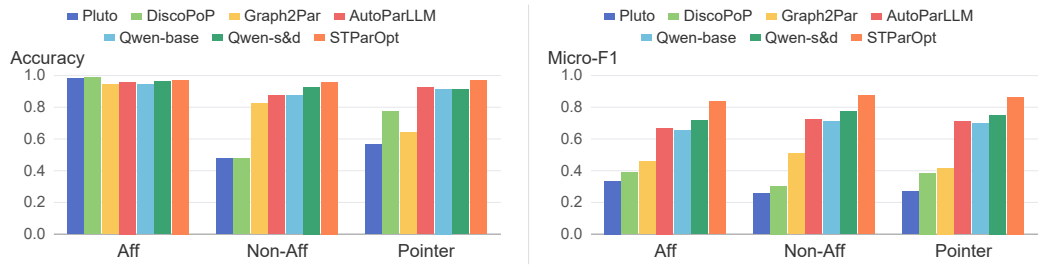

Figure 2: Micro-F1 and Accuracy on affine, non-affine, and pointer-intensive loops. STParOpt bars show three-seed means. **STParOpt** achieves the highest Micro-F1 across all three categories; polyhedral methods (Pluto, DiscoPoP) perform poorly on Non-Aff and Pointer patterns.

We stratify the test set into affine, non-affine, and pointer-intensive loops and report Micro-F1 and Accuracy in Figure 2. Polyhedral methods perform poorly on irregular patterns (Non-Aff F1: **0.2566/0.3027**; Pointer F1: **0.2682/0.3855**), while **STParOpt** achieves the highest Micro-F1 across all three categories (Table 8). Among LLM baselines, `s&d` consistently improves Micro-F1 over `base`, supporting the benefit of execution-aware signals across loop types.

4.6.2 Static-Dynamic Disagreement Analysis

We partition the OMP_Serial test set according to whether static structural evidence and dynamic runtime evidence indicate consistent (*agreement*) or conflicting (*disagreement*) dependence status before fusion. These evidence-level groups are not defined by equality of the branch classifiers' final labels. Table 4 summarizes the groups computed separately for each seed.

Table 4: Static-dynamic disagreement analysis on 3257 OMP_Serial test loops. Entries show mean (sample SD) over three seeds; the last three columns are accuracy (%). $\bar{H}_i$ is mean static predictive entropy. Overall values are computed per seed before averaging.

Group	**Size**	**Share (%)**	$\bar{H}_i$	**Static**	**Dynamic**	**STParOpt**
Agreement	2420.0 (16.1)	74.30 (0.49)	0.314 (0.011)	98.46 (0.15)	98.00 (0.08)	99.82 (0.09)
Disagreement	837.0 (16.1)	25.70 (0.49)	0.872 (0.011)	72.20 (1.04)	69.25 (0.81)	87.41 (0.93)
Overall	3257	100	0.457 (0.009)	91.71 (0.38)	90.62 (0.28)	**96.63** (0.23)

The disagreement group exhibits higher entropy ($\bar{H}_i = \mathbf{0.872 \pm 0.011}$ vs. $\mathbf{0.314 \pm 0.011}$), indicating greater static prediction uncertainty in cases with conflicting evidence. Within this group, static and dynamic branches achieve $\mathbf{72.20 \pm 1.04\%}$ and $\mathbf{69.25 \pm 0.81\%}$ accuracy, respectively, while **STParOpt** reaches $\mathbf{87.41 \pm 0.93\%}$. The per-seed gains are $\mathbf{15.21 \pm 0.18}$ **pp** and $\mathbf{18.16 \pm 0.18}$ **pp**, supporting the role of fusion in resolving ambiguity that neither branch handles as accurately on its own.

4.6.3 Impact on Optimization Quality

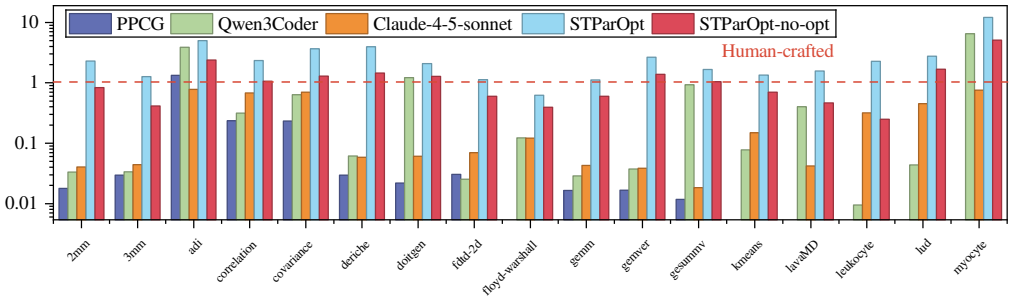

Figure 3: Relative speedup over Human CUDA on 17 benchmarks selected for visualization, not an aggregate evaluation set. The complete 48-kernel performance analysis appears in Appendix C.3. The dashed line marks $S = 1$; unavailable PPCG results are shown as zero.

We compare **STParOpt** against **STParOpt-no-opt**, which uses identical parallelism predictions but replaces optimization directives with a generic prompt. Figure 3 is illustrative; conclusions use the full results in Appendix C.3, not the 17-case subset. After excluding nine Human CUDA references no faster than serial, the 39-kernel performance analysis gives geometric-mean speedups of approximately **30.70**$\times$ for STParOpt, **13.64**$\times$ for no-opt, and **20.80**$\times$ for Human CUDA. This post-repair performance analysis is separate from the untouched automatic outputs in Table 2.

5 Threats to Validity

Evaluation covers 79 programs at per-loop, single-generated-kernel granularity, not repository-level multi-file or end-to-end multi-kernel migration. Harnesses and representative inputs may require human preparation. Finite traces and validation do not prove dependence freedom for unseen inputs (Appendix A.2). Advanced tiling, complex memory-hierarchy tuning, kernel fusion, and multi-GPU execution remain outside the five-strategy scope.

6 Conclusion

We propose **STParOpt**, which jointly models static program structure and dynamic execution behavior to infer loop parallelizability and latent CUDA optimization strategies, enabling end-to-end C-to-CUDA migration. **STER** reduces dependency ambiguity using execution-aware representations, entropy-guided fusion adaptively reconciles static and dynamic signals, and the hierarchical decision framework translates inferred strategies into LLM-guided CUDA generation. Experiments on standard HPC benchmarks demonstrate that **STParOpt** consistently outperforms both learning-based and LLM-based baselines across affine, non-affine, and pointer-intensive loops, with the largest gains on irregular patterns where static analysis fails. These findings suggest that runtime evidence helps determine not only which loops to parallelize but also how to optimize their CUDA implementations. Future work targets repository-level migration, inter-procedural analysis, broader CUDA optimization, and OpenMP/SYCL.

Acknowledgements

This work was supported by the National Natural Science Foundation of China (NSFC) [No. 62372334] and the National Key R&D Plan of China [No. 2020YFA0607902 and No. 2023YFB3106900].

The authors declare no competing interests.

References

[1] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN*

Conference on Programming Language Design and Implementation, PLDI '08, page 101–113, New York, NY, USA, 2008. Association for Computing Machinery.

[2] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4), January 2013.

[3] Zhen Li, Ali Jannesari, and Felix Wolf. Discovery of potential parallelism in sequential programs. In *2013 42nd International Conference on Parallel Processing*, pages 1004–1013, 2013.

[4] Le Chen, Quazi Ishtiaque Mahmud, Hung Phan, Nesreen Ahmed, and Ali Jannesari. Learning to parallelize with openmp by augmented heterogeneous ast representation. In D. Song, M. Carbin, and T. Chen, editors, *Proceedings of Machine Learning and Systems*, volume 5, pages 442–456. Curan, 2023.

[5] Quazi Ishtiaque Mahmud, Ali TehraniJamsaz, Hung D Phan, Nesreen K. Ahmed, and Ali Jannesari. Autoparllm: Gnn-guided automatic code parallelization using large language models, 2023.

[6] Ali TehraniJamsaz, Arijit Bhattacharjee, Le Chen, Nesreen K. Ahmed, Amir Yazdanbakhsh, and Ali Jannesari. Coderosetta: Pushing the boundaries of unsupervised code translation for parallel programming. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 100965–100999. Curran Associates, Inc., 2024.

[7] Yuanbo Wen, Qi Guo, Qiang Fu, Xiaqing Li, Jianxing Xu, Yanlin Tang, Yongwei Zhao, Xing Hu, Zidong Du, Ling Li, Chao Wang, Xuehai Zhou, and Yunji Chen. BabelTower: Learning to auto-parallelized program translation. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 23685–23700. PMLR, 17–23 Jul 2022.

[8] Yuxuan Han, Meng-Hao Guo, Zhengning Liu, Wenguang Chen, and Shi-Min Hu. Making llms optimize multi-scenario cuda kernels like experts, 2026.

[9] Matyáš Brabec, Jiří Klepl, Michal Töpfer, and Martin Kruliš. Tutoring llm into a better cuda optimizer, 2026.

[10] Gleison Mendonça, Breno Guimarães, Péricles Alves, Márcio Pereira, Guido Araújo, and Fernando Magno Quintão Pereira. Dawncc: Automatic annotation for data parallelism and offloading. *ACM Trans. Archit. Code Optim.*, 14(2), May 2017.

[11] Cedric Nugteren and Henk Corporaal. Bones: An automatic skeleton-based c-to-cuda compiler for gpus. 11(4), December 2014.

[12] Jan Kwiatkowski and Dzanan Bajgoric. Automatic parallelization of ansi c to cuda c programs. In Roman Wyrzykowski, Jack Dongarra, Ewa Deelman, and Konrad Karczewski, editors, *Parallel Processing and Applied Mathematics*, pages 459–470, Cham, 2018. Springer International Publishing.

[13] Karl-Filip Faxen, Konstantin Popov, Sverker Jansson, and Lars Albertsson. Embla - data dependence profiling for parallel programming. In *2008 International Conference on Complex, Intelligent and Software Intensive Systems*, pages 780–785, 2008.

[14] Xiangyu Zhang, Armand Navabi, and Suresh Jagannathan. Alchemist: A transparent dependence distance profiling infrastructure. In *2009 International Symposium on Code Generation and Optimization*, pages 47–58, 2009.

[15] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. Compiler-based graph representations for deep learning models of code. In *Proceedings of the 29th International Conference on Compiler Construction*, CC 2020, page 201–211, New York, NY, USA, 2020. Association for Computing Machinery.

[16] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F.P. O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 177–187, New York, NY, USA, 2009. Association for Computing Machinery.

[17] Daniel Fried, Zhen Li, Ali Jannesari, and Felix Wolf. Predicting parallelization of sequential programs using supervised learning. In *2013 12th International Conference on Machine Learning and Applications*, volume 2, pages 72–77, 2013.

[18] Yuanyuan Shen, Manman Peng, Shiling Wang, and Qiang Wu. Towards parallelism detection of sequential programs with graph neural network. *Future Generation Computer Systems*, 125:515–525, 2021.

[19] Le Chen, Quazi Ishtiaque Mahmud, and Ali Jannesari. Multi-view learning for parallelism discovery of sequential programs. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 295–303, 2022.

[20] R. Harel, T. Kadosh, N. Hasabnis, et al. Pragformer: Data-driven parallel source code classification with transformers. *International Journal of Parallel Programming*, 53(2), 2025.

[21] Zaigang Gong, Siyu Chen, Qiangsheng Dai, Ying Feng, Geng Niu, and Jinghui Zhang. RL-Freeze: Accelerating distributed freezing training with reinforcement learning. *Data Intelligence*, 7(2):416–439, 2025.

[22] Chahrazed B. Bachir Belmehdi, Abderrahmane Khiat, and Nabil Keskes. Predicting an optimal virtual data model for uniform access to large heterogeneous data. *Data Intelligence*, 6(2):504–530, 2024.

[23] Matthew T. Dearing, Yiheng Tao, Xingfu Wu, Zhiling Lan, and Valerie Taylor. Lassi: An llm-based automated self-correcting pipeline for translating parallel scientific codes. In *2024 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops)*, page 136–143. IEEE, September 2024.

[24] Zhili Shen, Pavlos Vougiouklis, Chenxin Diao, Kaustubh Vyas, Yuanyi Ji, and Jeff Z. Pan. Improving retrieval-augmented text-to-SQL with AST-based ranking and schema pruning. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 7865–7879. Association for Computational Linguistics, 2024.

[25] Wenbo Xu, Haifeng Zhu, Liang Yan, Chuanyi Liu, Peiyi Han, Shaoming Duan, and Jeff Z. Pan. TS-SQL: Test-driven self-refinement for text-to-SQL. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 2864–2889. Association for Computational Linguistics, 2025.

[26] Daniel Nichols, Joshua H. Davis, Zhaojun Xie, Arjun Rajaram, and Abhinav Bhatele. Can large language models write parallel code? In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '24, page 281–294, New York, NY, USA, 2024. Association for Computing Machinery.

[27] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In Aldo Gangemi, Roberto Navigli, Maria-Esther Vidal, Pascal Hitzler, Raphaël Troncy, Laura Hollink, Anna Tordai, and Mehwish Alam, editors, *The Semantic Web*, pages 593–607, Cham, 2018. Springer International Publishing.

[28] Alex Kendall, Yarin Gal, and Roberto Cipolla. Multi-task learning using uncertainty to weigh losses for scene geometry and semantics. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[29] Anthropic. Claude 4.5 sonnet, 2025. Accessed: Jan 16, 2026.

[30] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025.

A Method Details

A.1 Dependence Edge Construction

RAW, WAR, and WAW edges are constructed via a single-pass static dataflow analysis over the AST, without requiring compilation. Specifically, after parsing the loop body with *tree-sitter*, each `identifier` node is annotated as a definition (`is_def`) or use (`is_use`) based on its syntactic role (left-hand side of assignment vs. read context). Nodes are then sorted by source position (`start_byte`) to simulate sequential execution order. A single pass over the sorted nodes maintains per-variable state $\{\texttt{last_def}, \texttt{reads_since_def}\}$ and emits edges as follows:

- **RAW** (read-after-write): emitted from `last_def` to the current use node when a variable is read after a prior definition.
- **WAW** (write-after-write): emitted from `last_def` to the current definition node when a variable is written again.
- **WAR** (write-after-read): emitted from each node in `reads_since_def` to the current definition node when a variable is written after being read.

This analysis operates purely at the source/AST level and requires no compiler toolchain, header files, or complete translation units, making it applicable to isolated loop fragments. We acknowledge that this approach is conservative for pointer aliasing and indirect array accesses; such cases are precisely where the dynamic branch provides complementary evidence.

A.2 Trace Collection

Trace collection assumes an executable harness and representative inputs; preparing them may require human effort. LLM-based harness generation was not evaluated.

For each memory-accessing variable $x \in \mathcal{X}_i$ and input pattern k, we instrument the loop with GDB to record the actual memory addresses accessed per iteration. Prior to trace collection, a preprocessing step parses the loop body using *tree-sitter* to identify array access expressions (e.g., `A[i]`, `B[idx[i]]`) and pointer dereferences (e.g., `*p`), and automatically generates variable-specific GDB evaluation commands. GDB execution is terminated after $T = 50$ iterations via a conditional breakpoint. Stride sequences are derived as $\Delta_x^{(k,\tau)} = a_x^{(k,\tau+1)} - a_x^{(k,\tau)}$, where $a_x^{(k,\tau)}$ is the address of the element accessed by variable x at iteration τ under input pattern k.

Listing 1: GDB script for dynamic trace collection. This is a simplified illustrative version; our actual implementation uses the Python GDB API for robust error handling and structured JSON output.

```
# Step 1: Compile with debug symbols, no optimization
gcc -g -O0 loop.c -o loop_debug

# Step 2: Generate K=3 input patterns (sequential, alias, random)
python3 gen_inputs.py --mode sequential  > input_0.txt
python3 gen_inputs.py --mode alias       > input_1.txt
python3 gen_inputs.py --mode random      > input_2.txt

# Step 3: Run GDB for each input pattern k in {0,1,2}
for k in 0 1 2; do
gdb -batch -quiet \
    -ex "set confirm off" \
```

```
    -ex "set pagination off" \
    -ex "set $iter = 0" \
    -ex "break loop.c:LOOP_LINE" \
    -ex "commands
        silent
        set $iter = $iter + 1
        # --- Auto-generated by tree-sitter preprocessing ---
        # Example 1: for "A[i]"   -> base + index
        printf "%p\n", (void *)(A + i)
        # Example 2: for "B[idx[i]]" -> nested index
        printf "%p\n", (void *)(B + idx[i])
        # Example 3: for "*p" -> dereferenced pointer value
        printf "%p\n", (void *)*(int **)&p
        # -------------------------------------------------
        if $iter >= 50
            quit
        end
        continue
    end" \
    -ex "run < input_${k}.txt" \
    ./loop_debug > trace_${k}.txt
done

# Step 4: Parse addresses and compute stride sequences
python3 parse_traces.py --inputs trace_0.txt trace_1.txt trace_2.txt \
    --output strides.json
```

Key implementation details. The tree-sitter preprocessing step automatically generates the correct address expression for each variable: for indexed arrays (`A[i]`), we compute `(void*)(A + i)`; for nested indexing (`B[idx[i]]`), we recursively evaluate the index expression; for pointer dereferences (`*p`), we use pointer casting to obtain the actual memory address being accessed rather than the stack address of the pointer variable itself. In our actual implementation (not shown for brevity), we use the GDB Python API for more robust error handling and to produce structured JSON output directly.

Frontend Failure Handling. For complex builds, we retain compiler flags, macros, include paths, and source mappings through `compile_commands.json`, using a profiling binary built with `-g` and `-O0`/`-Og`. If dynamic tracing fails, we use static-only analysis ($\alpha_i = 0$), never interpreting a missing trace as independence. If parsing or execution still fails, the unit is marked unsupported. Repository-level migration additionally requires multi-file integration, cross-kernel data movement, synchronization, global-state handling, and scheduling, which are not evaluated here.

Candidate Adoption and Deployment. A candidate passing the available validation inputs is accepted in our evaluation, but the prototype neither overwrites/deploys the source loop nor automatically switches implementations at runtime. The original serial code remains unchanged; incomplete traces, abstention, or failed validation prevent candidate adoption. An accepted candidate can still fail on an unseen input activating an unobserved dependence, without guaranteed detection. Runtime guards dispatching to serial execution remain future work: safe dispatch requires sound, input-checkable conditions, combining execution evidence with symbolic alias/index/control-flow analysis while controlling guard overhead.

A.3 Domain Features

For each variable x and input pattern k, we extract four scalar features that characterize dependency-relevant memory access behaviors:

$$f_1^{(k)} = \mathrm{var}(\boldsymbol{\Delta}_x^{(k)}), \qquad \text{(stride regularity)} \tag{14}$$

$$f_2^{(k)} = \frac{|\{\tau : \Delta_x^{(k,\tau)} = 0\}|}{T}, \qquad \text{(alias degree)} \tag{15}$$

$$f_3^{(k)} = \frac{|\{\tau : \Delta_x^{(k,\tau)} \cdot \Delta_x^{(k,\tau-1)} < 0\}|}{T-1}, \qquad \text{(access directionality)} \tag{16}$$

$$f_4^{(k)} = \frac{|\{\tau : a_x^{(k,\tau)} = a_x^{(k,\tau-1)}\}|}{T-1}. \qquad \text{(cross-iteration reuse)} \tag{17}$$

While both f_2 and f_4 relate to zero-stride patterns, f_2 captures local stride stagnation within a single iteration, whereas f_4 reflects cross-iteration temporal reuse of the same absolute address. Empirically, the correlation between f_2 and f_4 is below 0.3 across our dataset, confirming that the two features provide complementary signals for dependency inference. Features remain pattern-specific, yielding $\mathbf{f}_x^{(k)} \in \mathbb{R}^4$ for each of the K runs; they are encoded independently and retained as the K rows of $\mathbf{D}_x$ before cross-attention.

A.4 All Prompts

A.4.1 Parallel Loop Detection Prompts

All detection prompts share the same output format requirement: the model must return a JSON object with a single field `label` $\in \{0, 1, 2\}$, where 0 denotes independent, 1 reduction, and 2 sequential.

Base Configuration (`-base`).

```
You are an expert in parallel programming. Given the
following C loop, determine whether it can be parallelized.
Return a JSON object with field "label": 0 for independent
parallel, 1 for reduction, 2 for sequential (not
parallelizable). Loop: {source_code}
```

Static Configuration (`-static`). The base prompt is augmented with the MPG structure serialized as a JSON object, including typed nodes (statement, variable, constant, control) and typed edges (RAW, WAR, WAW, syn, ctrl):

```
You are an expert in parallel programming. Given the
following C loop and its program graph, determine whether it
can be parallelized. Program Graph (JSON): {mpg_json} Loop:
{source_code} Return a JSON object with field "label": 0 for
independent parallel, 1 for reduction, 2 for sequential.
```

Dynamic Configuration (`-dynamic`). The base prompt is augmented with per-variable runtime memory access traces across $K = 3$ input patterns:

```
You are an expert in parallel programming. Given the
following C loop and its runtime memory access traces,
determine whether it can be parallelized. Memory access
traces: {var_name}: {addr_1}, {addr_2}, ..., {addr_T}
(repeated for each variable and input pattern) Loop:
{source_code} Return a JSON object with field "label": 0
for independent parallel, 1 for reduction, 2 for sequential.
```

Full Configuration (`-s&d`). Combines both the MPG JSON and runtime traces:

```
You are an expert in parallel programming. Given the
following C loop, its program graph, and runtime memory access
traces, determine whether it can be parallelized. Program
Graph (JSON): {mpg_json} Memory access traces: {var_name}:
{addr_1}, {addr_2}, ... Loop: {source_code} Return a JSON
object with field "label": 0 for independent parallel, 1 for
reduction, 2 for sequential.
```

A.4.2 Optimization-Strategy Prediction

Optimization prediction uses the corresponding base, static, dynamic, or combined evidence above and requests five binary recommendations using the definitions in Appendix B.2. The output schema is:

```
{"memory_coalescing": 0, "warp_reduction": 0, "tiling": 0,
 "shared_memory": 0, "streaming": 0}
```

Each field must be 0 or 1; multiple strategies may be active together. Sequential loops receive an all-zero optimization vector.

A.4.3 CUDA Code Generation Prompts

Direct Generation (`-gen`)**.**

```
You are an expert CUDA programmer. Convert the following
serial C loop to a CUDA kernel. If the loop cannot
be parallelized, return the original C code. Loop:
{source_code}
```

Detection-then-Generation (`-det&gen`)**.** The LLM first predicts the parallelism label using the `-s&d` detection prompt, then generates CUDA code conditioned on that prediction in a second inference call:

```
You are an expert CUDA programmer. The following C loop has
been analyzed as: {predicted_label_description} Convert it to
a CUDA kernel accordingly. If the loop is sequential, return
the original C code. Loop: {source_code}
```

STParOpt Generation ($P^{\text{task}} \| P^{\text{par}} \| P^{\text{opt}} \| C$)**.** The full STParOpt prompt comprises four components. P^{task} provides the task definition and output requirements. $P^{\text{par}}(y_i)$ encodes the predicted parallelism type as a natural language analysis result, e.g., "*Static and dynamic analysis indicates this loop is* ***independently parallelizable***". $P^{\text{opt}}(\hat{\mathbf{o}}_i)$ lists the recommended optimization strategies predicted by the hierarchical decision module:

```
You are an expert CUDA programmer. Convert the following
serial C loop to an optimized CUDA kernel. Analysis result:
{parallelism_description} Recommended optimizations: the
following optimization strategies are suggested based on
the loop's execution characteristics. Apply one or more if
appropriate: {active_optimization_list} e.g., 1. Memory
coalescing: ensure consecutive threads access consecutive
memory addresses. 2. Shared memory: use shared memory to
cache frequently accessed data. If the loop is sequential,
return the original C code. Loop: {source_code}
```

where `{active_optimization_list}` enumerates the predicted strategies from {memory coalescing, warp reduction, tiling, shared memory, streaming}.

B More Details of Experiment Setting

B.1 Metrics Details

Type Consistency. All generation metrics are conditioned on type consistency: a generated output is considered valid only if the predicted parallelization type matches the ground truth. Concretely, if a parallelizable loop is predicted as sequential (no code generated), or a non-parallelizable loop is predicted as parallel (code should not have been generated), all metrics (CR, ER, Co, Fast) are set to zero for that instance regardless of output quality. This ensures end-to-end evaluation where detection errors propagate directly to generation. For compiler-based baselines (PPCG, Pluto) that do not generate output for loops they cannot parallelize, we treat the absence of output as returning the original C code, which is counted as correct for non-parallelizable loops and incorrect for parallelizable loops.

Speedup. Let $R_{m,i} = T_{\text{serial},i}/T_{m,i}$. For method m, geometric-mean speedup is

$$G_m = \exp\left(\frac{1}{|\mathcal{V}_m|}\sum_{i\in\mathcal{V}_m}\log R_{m,i}\right),$$

where $\mathcal{V}_m$ contains that method's correct CUDA outputs with positive valid timings. Failed outputs and serial returns do not contribute CUDA timings; correct but slower CUDA outputs remain included. Each method retains its own correct-output set, so these sets need not be nested. Common-output controls are reported separately in Appendix C.5.2. Relative speedup $S = T_{\text{Human}}/T_{\text{Model}}$ is reported per kernel for the Human CUDA comparison. The automatic generation results in Table 2 and the post-repair performance analysis in Table 9 use distinct protocols.

Dataset Overlap. We audit the 13,841-loop training corpus against PolyBench/Rodinia using file names, function-name keywords, and normalized hashes of tree-sitter-extracted loops. Normalization removes comments, replaces identifiers and numeric literals, and collapses whitespace. After filtering generic matches and obvious idioms, four patterns involving 14 training loops remained (0.101%). Manual review identified short idioms rather than shared benchmark kernel bodies; the audit found no evidence of systematic leakage.

B.2 Baseline Label Mapping

We map baseline outputs to our unified label space $\{0\text{: independent}, 1\text{: reduction}, 2\text{: sequential}\}$ as follows. DiscoPoP's "reduction" annotations are mapped to 1, and its "parallel" suggestions to 0; all other loops are labeled 2. PPCG and Pluto only target race-free parallel loops; if they generate code or schedule a loop for parallel execution, we assign label 0, otherwise 2. Graph2Par and AutoParLLM produce direct class predictions that are used without remapping. LLM-based baselines output JSON with field `label` $\in \{0, 1, 2\}$ as described in Appendix A.4.

Optimization Labels. MC requires coalescible global-memory accesses across adjacent threads; WR an associative cross-thread reduction; TL dependence-safe locality with a feasible tile working set; SM feasible block-local reuse; and ST independent, streamable memory-bound chunks suitable for pipelining or transfer–computation overlap. GPT-4.1, Claude 4.5 Sonnet, and Qwen3-Coder independently produce five-label JSON annotations, consolidated by per-label majority vote and checked for valid binary fields and all-zero optimization labels on sequential loops.

Three CUDA/parallel-programming annotators independently audited a random 1,000/13,841 loops (7.2%), blind to the LLM decisions. Two-of-three vote followed by discussion established the human reference. Exact LLM–human-reference agreement was 89.7% (897/1,000); 103 samples contained 105 label-level disagreements, all corrected in the audited subset. Macro/micro Cohen's κ was 0.864/0.925; these compare the LLM ensemble with the adjudicated reference, not the annotators with each other.

B.3 Implementation Details & Hyperparameters

Model Architecture. The MPG encoder uses two RGCN layers with hidden size 128, ReLU activation, and dropout 0.3. The first produces the static representations; after cross-attention injection at variable nodes, the second propagates execution-aware features to statement and control nodes.

The dynamic branch uses a 1-layer BiLSTM with hidden size 128 followed by a 2-layer MLP for domain feature projection. BiLSTM is adopted over self-attention-based encoders because stride sequences are fixed-length ($T = 50$) and exhibit directional temporal dependencies; BiLSTM captures these with lower parameter cost and no positional encoding overhead.

Cross-attention injection uses four heads with a total key/value dimension of 128 (32 dimensions per head). For each variable, $\mathbf{D}_x \in \mathbb{R}^{3\times128}$ contains its three pattern-specific embeddings. Each head uses $\mathbf{Q}_{x,r} \in \mathbb{R}^{1\times32}$ and $\mathbf{K}_{x,r}, \mathbf{V}_{x,r} \in \mathbb{R}^{3\times32}$; concatenating four contexts yields 128 features before output projection and residual injection. Each head normalizes attention weights across the three input patterns, enabling static-conditioned trace selection before the resulting dynamic context is injected through a residual connection. This pattern-selection mechanism is distinct from entropy-guided fusion, whose scalar α_i controls the graph-level balance between the complete static and dynamic representations.

All learnable embeddings are initialized with Xavier uniform.

Training Configuration. We train with Adam optimizer, learning rate 1×10^{-3}, weight decay 1×10^{-5}, and batch size 32. Training runs for up to 100 epochs with early stopping based on dev

set M-F1. We repeat training of STParOpt and its trainable ablations with seeds $\{42, 2026, 10086\}$. Across-seed standard deviations use the sample estimator with denominator $n-1$ ($n=3$).

Hyperparameter Search. The following hyperparameters are tuned on the dev set via grid search:

Table 5: Hyperparameter search ranges and selected values.

Hyperparameter	Search Range	Selected
Sequence length T	$\{10, 20, 50, 100\}$	50
Input patterns K	$\{1, 3, 5\}$	3
Learning rate	$\{1\times10^{-3}, 5\times10^{-4}, 1\times10^{-4}\}$	1×10^{-3}
Opt. loss weight λ	$\{0.01, 0.1, 0.5, 1.0\}$	0.1
Fusion gate scale w	$\{0.5, 1.0, 2.0\}$	1.0
Fusion gate bias b	$\{-1.0, 0.0, 1.0\}$	0.0

Code Generation. The API identifiers are `qwen3-coder-next`, `claude-4-5-sonnet-20250929`, `deepseek-v3`, and `gpt-4.1`; the additional comparisons use `gpt-5.6-sol` and `deepseek-v4-pro`. All use temperature 0.3, thinking disabled, provider-default top-p, and no API seed. The output budget is 8,192 tokens for C inputs of at most 3.2K tokens and 16,384 otherwise.

All LLM generation methods use three valid candidates per program (pass@3). API failures and malformed outputs are retried up to three times without counting toward this budget. Selection is automatic: Fast $>$ correct $>$ runnable $>$ compilable. CR/ER/Co record whether any candidate passes the corresponding check; Fast and speedup use the fastest correct candidate, subject to the type-consistency requirement above. STParOpt's primary backend is Qwen3-Coder. LASSI reimplements the published pipeline with the same backend and settings, adapting its OpenMP-oriented input prompt to serial C.

Experiments are conducted on a machine with Intel i7-14700KF CPU, NVIDIA RTX 4090 GPU, and 64 GB RAM. Source C programs are compiled with GCC 11.4 and generated CUDA kernels with NVCC 12.4.

B.3.1 Loss Definitions

Let $\mathcal{B}$ be a training batch, $y_i \in \{0,1,2\}$ its parallelism labels, and $\mathbf{o}_i \in \{0,1\}^5$ its optimization labels. Denote the fused, static, and dynamic three-class distributions by p_i^f, p_i^s, and p_i^d, respectively. The auxiliary distributions are computed from the corresponding graph representations. The three parallelism losses are

$$\begin{aligned}\mathcal{L}_{\text{par}} &= -\frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}}\log p_i^f[y_i],\\ \mathcal{L}_{\text{static}} &= -\frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}}\log p_i^s[y_i],\\ \mathcal{L}_{\text{dynamic}} &= -\frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}}\log p_i^d[y_i].\end{aligned} \tag{18}$$

For optimization probabilities $\mathbf{r}_i = \text{sigmoid}(\mathbf{W}_o\,\text{stopgrad}(\mathbf{h}_i^{\text{fused}}))$, we first average binary cross-entropy over the five labels, then apply each sample's detached parallelizability weight:

$$\ell_i^{\text{opt}} = -\frac{1}{5}\sum_{j=1}^{5}\left[o_{ij}\log r_{ij} + (1-o_{ij})\log(1-r_{ij})\right], \tag{19}$$

$$g_i = \text{stopgrad}(p_i^f[0] + p_i^f[1]), \qquad \mathcal{L}_{\text{opt}} = \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} g_i \ell_i^{\text{opt}}. \tag{20}$$

The gate is applied *before* batch reduction, not multiplied into an already averaged loss. We use no additional class weights, ground-truth parallelism mask, or normalization by $\sum_i g_i$.

The total objective is Eq. 12, with positive learned scales $\sigma_1, \sigma_2, \sigma_3$ and $\lambda = 0.1$. The optimization loss updates its head but does not backpropagate through g_i or through the detached fused input into

the parallelism heads or shared encoders. The parallelism losses train those representations and their respective classifiers. Separately, static probabilities are detached when computing the entropy used by the fusion gate, as described in Section 3.4.

B.4 Human-Crafted Baseline

The human-crafted baseline refers to the reference CUDA implementations provided in the official Polybench-ACC [5] and Rodinia [6] repositories, which are community-maintained and represent optimized hand-written GPU kernels. Relative speedup is computed as $S = T_{\text{Human}}/T_{\text{Model}}$, where T_{Human} is the wall-clock execution time of the repository's reference CUDA kernel and T_{Model} is the execution time of the generated kernel, both measured on the same hardware (RTX 4090) with identical input sizes, 10 warmup runs, and 100 timed repetitions. $S > 1$ indicates that the generated kernel outperforms the human-written reference.

B.5 Inference Overhead

STParOpt involves two distinct computational phases.

Offline Trace Collection. For each loop, we instrument the program with GDB and record memory access addresses for up to $T = 50$ iterations per input pattern. GDB execution is terminated immediately after T iterations are collected, avoiding full program execution. Latency includes GDB startup, reaching the loop, and trace collection:

P50	P90	P99	Mean	Observed max
120 ms	380 ms	4.2 s	200 ms	10 s

For large programs, candidate-loop functions and their dependencies can be reduced to minimal executable units and analyzed directly. Traces are collected once offline and reused across all subsequent inference calls.

Online Inference. Given pre-collected traces, the per-loop inference latency is decomposed as follows:

Table 6: Per-component inference latency averaged over the OMP_Serial test set (3,257 loops) on RTX 4090.

Component	Latency (ms)
I/O (source code reading)	1.5
MPG construction	7.0
RGCN encoding	1.5
Stride computation	0.1
SPE encoding (BiLSTM + MLP)	1.0
Variable–node alignment	0.1
Entropy-guided fusion + decision	0.5
I/O (result saving)	0.8
Total	**12.5**

C Additional Results

C.1 Full Per-Strategy P/R/F1

Memory Coalescing and Warp Reduction achieve the highest F1 (**0.8865** $\pm$ **0.0029** and **0.8689** $\pm$ **0.0059**), consistent with their well-represented training patterns and observability from stride sequences. Stream achieves the highest recall (**0.9048** $\pm$ **0.0000**) at the cost of lower precision (**0.7811** $\pm$ **0.0183**), suggesting the model tends to over-predict streaming opportunities. Tiling shows the

[5] https://github.com/cavazos-lab/PolyBench-ACC

[6] https://github.com/yuhc/gpu-rodinia

Table 7: Precision, recall, F1, and accuracy for each optimization strategy predicted by **STParOpt** on the OMP_Serial test set. Results are mean ± sample standard deviation over seeds $\{42, 2026, 10086\}$.

Optimization Strategy	Precision	Recall	F1-Score	Accuracy
Memory Coalescing	0.8955 ± 0.0014	0.8776 ± 0.0044	0.8865 ± 0.0029	0.9400 ± 0.0014
Shared Memory	0.8191 ± 0.0028	0.8624 ± 0.0165	0.8402 ± 0.0093	0.9873 ± 0.0006
Warp Reduction	0.8987 ± 0.0165	0.8412 ± 0.0112	0.8689 ± 0.0059	0.9637 ± 0.0019
Stream	0.7811 ± 0.0183	0.9048 ± 0.0000	0.8383 ± 0.0106	0.9977 ± 0.0002
Tiling	0.7984 ± 0.0172	0.7419 ± 0.0230	0.7688 ± 0.0060	0.9818 ± 0.0004

lowest F1 (**0.7688 ± 0.0060**); its limited sample count (133 positive instances) may contribute to this long-tail challenge in imbalanced strategy prediction.

C.2 Full Aff/Non-Aff/Pointer Results

Table 8: Micro-F1 and Accuracy stratified by loop type (Aff / Non-Aff / Pointer) on the OMP_Serial test set. **STParOpt** results are mean ± sample standard deviation over seeds $\{42, 2026, 10086\}$; baseline results are from the original evaluation. Micro-F1 uses $2\mathrm{TP}/(2\mathrm{TP} + \mathrm{FP} + \mathrm{FN})$, with counts aggregated within each loop type.

Method	Aff		Non-Aff		Pointer	
	F1	Acc	F1	Acc	F1	Acc
Pluto	0.3354	0.9798	0.2566	0.4784	0.2682	0.5671
DiscoPoP	0.3902	0.9877	0.3027	0.4784	0.3855	0.7719
Graph2Par	0.4563	0.9439	0.5069	0.8253	0.4174	0.6422
AutoParLLM	0.6695	0.9585	0.7210	0.8726	0.7102	0.9240
Qwen-base	0.6532	0.9450	0.7097	0.8758	0.6961	0.9159
Qwen-s&d	0.7172	0.9607	0.7743	0.9279	0.7510	0.9106
STParOpt	**0.8364** ±0.0045	**0.9727** ±0.0076	**0.8761** ±0.0035	**0.9599** ±0.0042	**0.8646** ±0.0043	**0.9684** ±0.0077

Table 8 reports Micro-F1 and Accuracy stratified by loop type for all methods. Polyhedral methods (Pluto, DiscoPoP) achieve near-perfect Accuracy on Aff loops (**0.9798/0.9877**) but perform poorly on Non-Aff and Pointer patterns, reflecting their limitation to affine loop structures. Graph2Par improves over polyhedral methods on irregular patterns but still lags substantially behind LLM-based approaches. Among LLM baselines, Qwen-s&d consistently improves Micro-F1 over Qwen-base across all three categories, supporting the benefit of execution-aware signals across loop types. **STParOpt** achieves the highest Micro-F1 across all three categories: **0.8364 ± 0.0045** on Aff (**+11.92 pp** over Qwen-s&d), **0.8761 ± 0.0035** on Non-Aff (**+10.18 pp**), and **0.8646 ± 0.0043** on Pointer (**+11.36 pp**). The stark contrast between high Accuracy and low F1 for polyhedral methods reflects their conservative bias toward sequential predictions, which inflates Accuracy on class-imbalanced data while severely under-detecting parallelizable loops.

C.3 Full Speedup Table

Table 9 reports serial-relative speedup and $S = T_{\text{Human}}/T_{\text{Model}}$ on 48 kernels (30 Polybench and 18 Rodinia). All outputs affected by header-related build errors were repaired for this performance analysis; algorithms and kernel logic were unchanged. These repaired results do not replace the automatic CR/ER/Co/Fast results in Table 2. Values of $S < 1$ mean slower-than-Human execution; zero or missing entries do not constitute valid speedup measurements.

C.3.1 Sensitivity to Slow Human References

Nine references do not outperform serial execution: atax, durbin, jacobi-1d, trisolv, bfs, myocyte, nw, particlefilter, and srad. We exclude these same reference cases for every method and recompute geometric means on the remaining 39-kernel subset. Each method uses its available valid timings

Table 9: Per-kernel speedup over serial execution and relative speedup S over Human CUDA. This performance-only analysis includes manual fixes of header-related build errors, without changing algorithms or kernel logic, and is separate from the untouched end-to-end outputs in Table 2.

Kernel	**PPCG**		**Qwen3Coder**		**Claude-4.5-sonnet**		**STParOpt**		**STParOpt-no-opt**		**Human**
	Speedup	S	Speedup	S	Speedup	S	Speedup	S	Speedup	S	Speedup
Polybench											
2mm	2.82	0.02	5.25	0.03	6.38	0.04	361.77	2.28	131.55	0.83	158.35
3mm	6.57	0.03	7.48	0.03	9.77	0.04	281.07	1.27	91.68	0.41	221.82
adi	9.73	1.33	28.32	3.88	30.18	4.14	36.37	4.98	17.46	2.39	7.30
atax	0.01	0.01	0.01	0.01	0.01	0.02	0.23	0.25	0.11	0.11	0.92
bicg	0.02	0.01	0.04	0.01	0.04	0.01	0.66	0.22	0.30	0.10	3.03
cholesky	1.81	0.64	2.62	0.93	2.84	1.01	4.74	1.68	2.16	0.76	2.83
correlation	8.64	0.24	11.59	0.32	24.93	0.68	86.12	2.34	38.98	1.06	36.73
covariance	9.50	0.23	25.74	0.63	28.41	0.70	149.07	3.67	52.60	1.30	40.60
deriche	0.53	0.03	1.09	0.06	1.16	0.06	70.56	3.96	25.82	1.45	17.82
doitgen	0.47	0.02	26.12	1.22	28.49	1.33	44.72	2.08	27.45	1.28	21.47
durbin	0.00	0.43	1.06	129.52	1.14	139.72	1.96	239.53	0.89	108.88	0.01
fdtd-2d	2.37	0.03	1.98	0.03	5.48	0.07	88.25	1.13	46.71	0.60	78.22
floyd-warshall	–	–	30.87	0.12	32.88	0.13	156.48	0.62	99.70	0.40	251.99
gemm	0.67	0.02	1.17	0.03	1.75	0.04	45.43	1.12	24.39	0.60	40.69
gemver	0.04	0.02	0.08	0.04	0.08	0.04	5.74	2.65	2.99	1.38	2.17
gesummv	0.01	0.01	1.00	0.93	1.07	1.00	1.79	1.67	1.12	1.04	1.08
gramschmidt	10.61	0.70	14.73	0.97	15.72	1.03	26.11	1.72	11.87	0.78	15.21
heat-3d	4.24	0.13	4.55	0.14	4.93	0.15	8.17	0.25	3.71	0.11	33.13
jacobi-1d	0.00	0.09	0.00	0.13	0.00	0.14	0.00	0.24	0.00	0.11	0.02
jacobi-2d	3.52	0.09	4.36	0.11	5.16	0.13	68.91	1.76	31.32	0.80	39.25
lu	6.60	0.08	0.81	0.01	5.48	0.07	91.09	1.14	41.41	0.52	79.73
ludcmp	0.09	0.00	10.95	0.18	11.65	0.19	19.48	0.32	8.85	0.14	61.55
mvt	0.05	0.02	0.09	0.03	0.09	0.03	1.54	0.45	0.70	0.21	3.40
nussinov	0.31	0.02	11.65	0.87	12.44	0.92	20.83	1.55	9.47	0.70	13.46
seidel-2d	26.28	0.32	50.79	0.61	54.28	0.66	90.61	1.10	41.19	0.50	82.66
symm	4.46	0.02	350.11	1.70	378.94	1.84	631.56	3.06	287.07	1.39	206.17
syr2k	9.27	0.03	7.95	0.03	14.05	0.05	219.79	0.76	99.90	0.34	290.00
syrk	1.48	0.02	1.00	0.02	1.99	0.03	31.06	0.48	14.12	0.22	65.39
trisolv	0.01	0.35	0.30	19.92	0.32	21.38	0.54	35.84	0.24	16.29	0.01
trmm	3.43	0.03	162.20	1.30	173.34	1.39	288.49	2.31	131.13	1.05	124.77
Rodinia											
b+tree_kernel_cpu	–	–	0.45	0.01	0.49	0.02	8.18	0.27	3.72	0.12	30.04
backprop_backprop_kernel	–	–	0.01	0.01	0.01	0.01	0.26	0.09	0.12	0.04	2.72
bfs_bfs	–	–	0.07	0.08	0.08	0.09	1.27	1.46	0.58	0.67	0.87
cfd_euler3d_cpu	–	–	66.09	11.97	71.63	12.97	119.09	21.56	54.13	9.80	5.52
heartwall_kernel	–	–	7.14	0.33	7.72	0.36	12.82	0.60	5.83	0.27	21.37
hotspot3D_3D	–	–	0.86	0.02	0.93	0.02	15.48	0.29	7.03	0.13	54.17
hotspot_hotspot_openmp	–	–	0.05	0.03	0.05	0.03	0.85	0.51	0.39	0.23	1.66
kmeans_kmeans_clustering	–	–	4.89	0.33	5.27	0.35	19.95	1.34	10.42	0.70	14.90
lavaMD_kernel_cpu	–	–	1.38	0.48	1.48	0.51	4.53	1.56	1.34	0.46	2.90
leukocyte_detect_main	–	–	4.65	0.33	5.01	0.36	31.62	2.26	3.50	0.25	13.98
lud_lud_omp	–	–	7.51	0.06	8.11	0.06	348.93	2.77	212.16	1.68	126.02
myocyte_solver	–	–	0.03	1.81	0.03	1.95	0.21	12.15	0.09	5.09	0.02
nn_nn_openmp	–	–	12.33	3.27	13.28	3.52	221.98	58.90	100.90	26.77	3.77
nw_needle	–	–	0.33	0.69	0.35	0.74	5.88	12.42	2.67	5.65	0.47
particlefilter_ex_particle	–	–	0.03	0.83	0.03	0.89	2.24	67.80	0.49	14.92	0.03
pathfinder_pathfinder	–	–	9.23	0.74	9.96	0.80	167.67	13.41	76.21	6.09	12.51
srad_main	–	–	0.53	0.95	0.58	1.02	9.64	17.08	4.38	7.76	0.56
streamcluster	–	–	8.12	4.23	7.45	3.88	22.12	11.52	10.67	5.56	1.92
Count (>1)	16	1	31	10	33	13	41	34	37	20	–

within that subset; results in Table 10 are approximate because they are recomputed from the displayed per-kernel speedups.

STParOpt remains approximately $1.48\times$ faster than Human CUDA and $2.25\times$ faster than no-opt under this analysis. Separately, the automatic-generation timings underlying Table 2 show gains across several optimization-sensitive kernels rather than only 2mm: the serial-relative speedups of STParOpt/no-opt are $361.77/131.55\times$ on 2mm, $281.07/13.86\times$ on 3mm, $631.56/28.71\times$ on symm, and $70.56/2.58\times$ on deriche. Both variants have valid timings for these cases and differ only in optimization directives; some baselines have no valid timing on part of these high-gain kernels. Speedup is workload-dependent, and these claims are limited to the evaluated workloads.

C.4 Generation-Backbone Portability

We replace only the generation backbone while keeping STParOpt predictions, optimization directives, prompts, temperature, and output budget unchanged. Table 11 reports the same 79-program evaluation. Qwen3-Coder was selected for open-weight availability and lower inference cost, not because the framework depends on one provider.

Table 10: Geometric-mean speedup over serial after excluding Human CUDA references with speedup ≤ 1. This is a sensitivity analysis of the post-repair performance results, not an alternative correctness rate.

Method	Geometric mean ($\times$)
PPCG	1.30
Qwen3-Coder	3.63
Claude 4.5 Sonnet	4.42
STParOpt-no-opt	13.64
Human CUDA	20.80
STParOpt	**30.70**

Table 11: STParOpt generation-backbone portability. Rates are percentages over 79 programs; Geomean is speedup over serial on each backbone's correct outputs with valid timings.

Backbone	CR	ER	Co	Fast	Geomean ($\times$)
Qwen3-Coder	89.87	88.61	86.08	81.01	15.50
GPT-4.1	91.14	91.14	87.34	83.54	15.54
Claude 4.5 Sonnet	93.67	92.41	91.14	87.34	15.90

These backbones improve feasibility and correctness while the aggregate speedup changes little. On the 41 kernels with valid outputs under all three backbones, their geometric means are $14.37\times$, $15.41\times$, and $14.47\times$, respectively. This is consistent with the backbone affecting realization of correct CUDA code, while performance also depends on the workload and predicted optimization strategies.

C.5 Additional Model Comparisons

We additionally evaluate GPT-5.6-Sol and DeepSeek-V4-Pro on the same prediction dataset and full 79-program generation suite, using the original evaluation protocols. Prediction and generation results are reported separately below.

C.5.1 Loop Detection and Optimization Prediction

Table 12: Additional-model prediction results (%). LLM rows use the original inference settings; STParOpt reports three-seed mean $\pm$ sample SD.

Model	Setting	Acc	Par Macro-F1	OPT Macro-F1
DeepSeek-V4-Pro	base	86.28	85.97	71.60
	static	88.76	88.53	73.72
	dynamic	89.87	89.84	75.78
	s&d	92.02	91.95	78.17
GPT-5.6-Sol	base	90.51	90.56	71.92
	static	91.77	91.82	74.24
	dynamic	93.06	93.32	74.46
	s&d	94.57	95.00	77.48
STParOpt	—	$\mathbf{96.63} \pm 0.23$	$\mathbf{96.90} \pm 0.19$	$\mathbf{84.05} \pm 0.43$

Dynamic evidence is at least as informative as static evidence alone in these comparisons, and combining both gives the best results for each model. Their best OPT Macro-F1 values remain 6.57pp and 5.88pp below STParOpt for GPT-5.6-Sol and DeepSeek-V4-Pro, respectively, supporting the benefit of task-specific learning over static and execution evidence.

C.5.2 CUDA Generation

All methods use the same 79 programs, pass@3 protocol, and compilation, execution, correctness, and performance checks. Table 13 compares direct generation, detection-then-generation, and STParOpt with each backbone.

Table 13: Additional-model CUDA generation results. Rates are percentages over 79 programs; Geomean is speedup over serial on each method's correct outputs with valid timings.

Backbone	Method	CR	ER	Co	Fast	Geomean (×)
DeepSeek-V4-Pro	gen	73.42	73.42	68.35	64.56	2.96
	det&gen	77.22	77.22	72.15	67.09	4.79
	STParOpt	89.87	87.34	86.08	82.28	15.69
GPT-5.6-Sol	gen	74.68	74.68	70.89	67.09	3.88
	det&gen	82.28	82.28	74.68	70.89	4.90
	STParOpt	91.14	91.14	88.61	84.81	15.79

Relative to det&gen with the same backbone, STParOpt improves Co/Fast by 13.92/13.92pp with GPT-5.6-Sol and 13.92/15.19pp with DeepSeek-V4-Pro. Their geometric means increase from $4.90\times$ to $15.79\times$ and from $4.79\times$ to $15.69\times$, respectively.

Because each method's geometric mean uses its own correct outputs, we also compare each pair on kernels for which both methods produce correct CUDA with valid timings. STParOpt achieves $20.21\times$ versus $5.69\times$ on 31 GPT-5.6-Sol kernels and $21.48\times$ versus $5.79\times$ on 30 DeepSeek-V4-Pro kernels. The correct-output sets are not nested; all method-specific correct outputs remain included in Table 13.

Gains occur across all four generation metrics, with larger absolute increases in Co/Fast than in CR/ER. In det&gen, the LLM derives the parallelization decision from prompted evidence and chooses the CUDA optimizations. STParOpt supplies an execution-aware parallelism prediction and explicit optimization recommendations to the generator. This difference is consistent with the correctness and performance gains observed across both generators.

C.6 Optimization Directive Adoption Analysis

Among the 48 benchmarks, **STParOpt** predicts at least one optimization strategy for **37** kernels; the remaining **11** are predicted as requiring conservative handling due to irregular or sequential structure. Memory coalescing (MC) and shared memory (SM) are the most frequently predicted strategies (35/48 each), reflecting their broad applicability to memory-bound HPC kernels. **10** kernels receive all five directives simultaneously and **15** receive four-strategy combinations, indicating that the framework frequently predicts complementary multi-strategy optimizations rather than isolated directives.

To verify that predicted directives are realized in generated code, we inspect each kernel for optimization-specific patterns: coalesced indexing for MC, warp-level primitives for WR, tiling structures for TL, shared-memory allocation for SM, and CUDA stream usage for ST. Among the 37 strategy-predicted kernels, **84.6%** of directives are realized on average. SM and MC exhibit the highest adoption rates (**91.4%** and **89.1%**), while TL is more challenging (**72.5%**) due to kernel-specific boundary conditions. Without optimization directives (**STParOpt-no-opt**), adoption rates drop below **40%** across all strategies, confirming that predicted constraints drive the generated optimizations rather than implicit LLM priors.

C.7 Sensitivity Analysis

Trace Length. We evaluate the sensitivity of **STParOpt** to the trace length $T \in \{10, 20, 50, 100\}$, which controls the number of loop iterations recorded per variable per input pattern. Figure 4 reports Accuracy and Macro-F1 on the OMP_Serial test set.

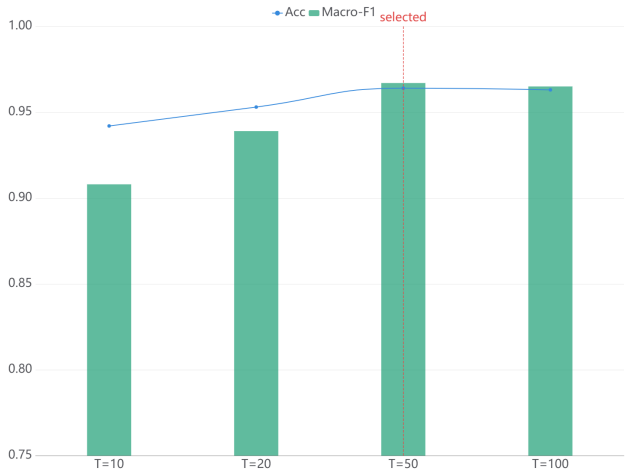

Figure 4: Sensitivity to trace length T. The dashed line marks the selected value ($T = 50$).

Both metrics improve consistently from $T = 10$ to $T = 50$, with the largest gain on Macro-F1 (**+5.9 pp**), confirming that longer traces provide richer stride pattern information for optimization strategy inference. Performance plateaus between $T = 50$ and $T = 100$ (Acc: **0.964** vs. **0.963**; Macro-F1: **0.967** vs. **0.965**), indicating that $T = 50$ captures sufficient behavioral information while avoiding unnecessary overhead.

Input Patterns and Hidden Dependencies. On 50 loop–input stress cases, including eight input-triggered hidden dependencies, the three fixed modes detect 3/8 dependencies; expanded inputs detect 7/8 and achieve 94.0% three-class accuracy overall. Conservative abstention declines parallelization in 10/50 cases and eliminates unsafe parallel decisions among these eight hidden-dependency cases. This is migration-time abstention, not automatic runtime recovery; the result concerns the observed cases, not a dependence-freedom guarantee for all unseen inputs.

D Case Study

D.1 Correct Parallelism Detection

The following loop performs a pointer-based indirect memory access, a pattern that confounds static analyzers but is correctly identified as independently parallelizable by **STParOpt**.

```
for (int i = 0; i < N; i++) {
    B[idx[i]] = A[i] * scale;
}
```

Pluto and DiscoPoP conservatively label this loop as sequential due to potential write-after-write hazards on `B` introduced by the indirect index `idx[i]`. AutoParLLM, operating on static graph representations alone, similarly fails to resolve the aliasing ambiguity. **STParOpt**'s SPE observes that `idx` accesses are non-repeating across iterations (alias degree $f_2 = 0$, stride regularity $f_1 \approx 4$; see Appendix A.3 for feature definitions), providing evidence that no write conflict occurs in the profiled executions. The entropy-guided fusion up-weights this dynamic evidence (high $H_i = 0.82$), leading to a correct *independent* prediction and the following generated CUDA kernel:

```
__global__ void kernel(float *A, float *B, int *idx,
                       float scale, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N)
        B[idx[i]] = A[i] * scale;
}
```

The generated kernel compiles and produces numerically correct results across all test inputs, achieving a speedup of **3.2**$\times$ over the serial baseline on a representative input of size $N = 10^6$.

D.2 Correct Sequential Rejection

The following loop contains a loop-carried dependency through pointer aliasing, making it non-parallelizable.

```
for (int i = 1; i < N; i++) {
    A[i] = A[i-1] * alpha + B[i];
}
```

LLM-based baselines (Qwen-gen, Claude-gen) incorrectly predict this loop as independently parallelizable and generate CUDA kernels that produce incorrect results due to the unresolved loop-carried dependence on `A`. **STParOpt**'s SPE detects a strong cross-iteration reuse pattern ($f_4 = 0.98$; see Appendix A.3) and non-zero stride overlap, providing unambiguous evidence of a RAW dependency. With low static entropy ($H_i = 0.21$), the static branch confidently predicts *sequential*, which is confirmed by the dynamic branch. **STParOpt** correctly rejects parallelization and returns the original C code, avoiding the data-race errors produced by all LLM-only baselines, whose generated kernels pass compilation but fail numerical verification.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction explicitly state three contributions (STParOpt framework, STER representation, and experimental results), each of which is supported by corresponding sections in the paper. The claimed accuracy ($96.63 \pm 0.23\%$) and code correctness (86.1%) are directly reported in Table 1 and Table 2. Limitations are discussed in the Threats to Validity section.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Section 5 discusses the 79-program evaluation scope, harness/input prerequisites, input-specific traces and finite validation, and the five-strategy optimization scope. Appendices B.5 and A.2 specify profiling costs, frontend failures, and candidate-adoption semantics; runtime guards and repository-level multi-file migration remain future work, not implemented deployment guarantees.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best

judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: This paper does not include theoretical results, proofs, or formal theorems. The contributions are empirical: a framework design, an execution-aware representation learning method, and experimental evaluations on standard HPC benchmarks.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The paper provides sufficient information for reproducibility. Model architecture details (2-layer RGCN, 1-layer BiLSTM, hidden size 128, dropout 0.3) are reported in Section 4.3 and Appendix B.3. All hyperparameters and their search ranges are listed in Table 5. The GDB trace collection procedure is described with a complete script in Appendix A.2. All prompts used for LLM-based baselines and STParOpt generation are provided in Appendix A.4. Training uses seeds $\{42, 2026, 10086\}$, and the experiments use publicly available datasets (OMP_Serial, PolyBench, Rodinia). Baseline label mapping rules are provided in Appendix B.2.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example

(a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
(b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
(c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: The project code, documentation, and data-access instructions are publicly available at `https://github.com/B551-Whu-NLP/STParOpt`. The repository covers MPG construction, SPE encoding, entropy-guided fusion, and evaluation for loop detection and CUDA generation. PolyBench, Rodinia, and OMP_Serial are public datasets referenced in the paper; hyperparameters, training configurations, GDB tracing, and LLM prompts are documented in the appendix.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: All training and evaluation details are provided in Section 4.3 and Appendix B.3, including model architecture (2-layer RGCN, 1-layer BiLSTM, hidden size 128), optimizer (Adam, lr=1×10^{-3}), dropout (0.3), batch size (32), early stopping (patience=10), and training seeds $\{42, 2026, 10086\}$. Data splits (8:1:1) and dynamic annotation procedure are described in Section 4. Hyperparameter search ranges and selected values are listed in Table 5. LLM temperature (0.3) and hardware configuration are reported in Section 4.3.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: STParOpt and its trainable ablations are trained with seeds $\{42, 2026, 10086\}$. Table 1 reports mean $\pm$ sample standard deviation across these three runs for STParOpt. This variation reflects repeated training, not repeated API generation.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
   - If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Hardware configuration is reported in Section 4.3: Intel i7-14700KF CPU, NVIDIA RTX 4090 GPU (24 GB), and 64 GB RAM. Compiler versions (GCC 11.4, NVCC 12.4) are also specified. Offline trace collection for the 4,500 annotated training loops takes approximately **2 hours**. Model training takes approximately **5 hours** on the RTX 4090. Test set inference (3,257 loops) completes in approximately **5 minutes** (12.5 ms per loop), as detailed in Appendix B.5. LLM-based CUDA generation averages **15 seconds per sample** via the Qwen3-Coder API.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.

- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: This research conforms with the NeurIPS Code of Ethics. The work focuses on automated C-to-CUDA migration for HPC applications and does not involve human subjects, sensitive data, or potential for direct harm. All datasets used are either publicly available or constructed from open-source code repositories.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: STParOpt lowers barriers to GPU acceleration for scientific computing, benefiting researchers without deep GPU expertise. The primary risk is incorrect parallelization producing wrong numerical results, which is mitigated by correctness verification in our evaluation pipeline.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: The paper poses no significant risks for misuse, as it addresses HPC code migration and does not involve sensitive data or dual-use models.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: All assets used in this paper are properly credited via citations. PolyBench is available under the MIT License; Rodinia is available under the University of Virginia open-source license. OMP_Serial is licensed under the MIT License. BabelTower is publicly available on GitHub but does not specify an explicit license; we use it for evaluation purposes only and cite the original paper. tree-sitter is MIT licensed. LLM APIs (Qwen3-Coder, Claude 4.5, GPT-4.1) are accessed via commercial APIs under their respective terms of service.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [N/A]

Justification: This paper does not introduce a new dataset, pretrained model, or other standalone research asset. The accompanying code repository is provided to support reproducibility of the proposed method and experiments.

Guidelines:

- The answer [N/A] means that the paper does not release new assets.

- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: This paper does not involve crowdsourcing or research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: This paper does not involve crowdsourcing or research with human subjects, and therefore does not require IRB approval.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: LLMs are a core component of the proposed framework. Specifically, Qwen3-Coder is used as the backbone for constraint-driven CUDA code generation (Section 3.5), where it receives structured prompts comprising predicted parallelism type and optimization directives. GPT-4.1, Claude 4.5 Sonnet, and Qwen3-Coder are also evaluated as baselines under multiple configurations. All LLM usage details, including prompt templates and inference settings (temperature 0.3), are documented in Appendix A.4.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.