Extended Abstract

Motivation GPU kernel optimization represents a fundamental bottleneck in high-performance computing and machine learning. Writing efficient GPU kernels requires extensive expertise in parallel programming, memory hierarchy optimization, and hardware-specific tuning—skills that are scarce and expensive to develop. While domain-specific languages (DSLs) like TileLang abstract away low-level details, they face a critical challenge: extreme scarcity of training data. TileLang, released just months ago, has only 24 examples in its official repository. This raises a compelling question: can modern machine learning techniques enable a small language model to learn such a low-resource programming language effectively?

Method We propose a three-stage progressive learning strategy that bootstraps from minimal examples to a comprehensive kernel generation system. First, we generate a synthetic dataset by combining manual kernel implementations with retrieval-augmented generation, growing from 37 seed kernels to 180 correct implementations. Second, we employ reasoning alignment through Kahneman-Tversky Optimization (KTO), training on post-hoc reasoning traces that explain the mapping from PyTorch operations to TileLang constructs. We choose this approach over an initial SFT stage because generating an SFT dataset of correct TileLang Kernels was too expensive for our budget. Finally, we apply reinforcement learning with verifiable rewards (RLVR) using Group Relative Policy Optimization (GRPO), directly optimizing for compilable, correct, and performant kernel generation through multi-turn exchanges.

Implementation & Results We fine-tuned Seed-Coder-8B ByteDance Seed et al. (2025) with our pipeline, achieving progressive improvements in compilation success. The base model produced 456 compilation errors out of 500 attempts (91% failure rate). After KTO alignment, errors decreased to 408 (81% failure rate), demonstrating that reasoning training improves syntactic understanding. GRPO further reduced compilation errors to 272 (54% failure rate), nearly halving the error rate. However, correctness remains very poor; our pass@5 metric did not improve as we expected. Performance evaluation on KernelBench levels 1-2 shows that while our model learns to generate syntactically valid TileLang code, it struggles with semantic correctness and optimization patterns.

Discussion & Conclusion Our results reveal both the promise and limitations of learning low-resource DSLs through progressive training. The dramatic reduction in compilation errors (91% to 54%) demonstrates that combining reasoning alignment with reinforcement learning can effectively teach syntactic patterns even with minimal data. However, the persistent correctness failures highlight a fundamental challenge: compilation success does not guarantee functional correctness. The GRPO model made rapid gains in mastering TileLang's syntax—boosting its compilation rate fivefold to 45.6%—but fell short on semantic correctness. This imbalance arose because compilation failures incur a strong, immediate penalty (1.0), whereas functional feedback is sparse (a single scalar reward) and offers no diagnostic guidance. Moreover, the environment's reset-on-failure rule and the limited number of GRPO training steps caused the model's learning to remain dominated by syntactic signals. Bridging this "correctness gap" would thus require a much longer training run to accrue enough informative, positive rewards for functional accuracy.

Can a Small Model Learn *TileLang*, a Very Low-resource DSL?

Nathan Paek
Department of Computer Science
Stanford University
`nathanjp@stanford.edu`

Sokserey Sun
Department of Computer Science
Stanford University
`sokserey@stanford.edu`

Zijian Luo
Department of Symbolic Systems
Stanford University
`zijianl@stanford.edu`

Abstract

We present a novel approach for teaching small language models to generate code in TileLang, an extremely low-resource domain-specific language for GPU kernel optimization with only 24 official examples. Through a three-stage pipeline combining synthetic data generation, reasoning alignment via preference optimization, and reinforcement learning with verifiable rewards, we achieve significant improvements in compilation success rates. Our fine-tuned Seed-Coder-8B model reduces compilation errors from 91% to 54%, demonstrating that progressive training can effectively teach syntactic patterns in low-resource settings. However, functional correctness remains elusive, with pass@5 scores below 2%, highlighting the gap between syntactic and semantic understanding in program synthesis.

1 Introduction

GPU kernel optimization represents a fundamental challenge in high-performance computing and machine learning. Writing efficient CUDA or GPU kernels requires extensive expertise in parallel programming, memory hierarchy optimization, and hardware-specific tuning—skills that are scarce and expensive to develop. This expertise gap has motivated the development of domain-specific languages (DSLs) that abstract low-level hardware details while maintaining performance. Languages like Triton Tillet et al. (2019), ThunderKittens Spector et al. (2024), and TileLang Wang et al. (2025) democratize GPU programming by providing higher-level abstractions.

However, these DSLs face a critical adoption challenge: the scarcity of training data for machine learning models. Unlike mainstream programming languages with millions of code examples, emerging DSLs have minimal documentation and few reference implementations. TileLang exemplifies this challenge—released only months ago with currently only 24 examples in its official repository (`https://github.com/tile-ai/tilelang`). This scarcity makes it nearly impossible to train language models using conventional supervised learning approaches.

Our work addresses the research question: *can we teach a small language model to write code in an extremely low-resource DSL?* We focus on TileLang as a representative case study, developing techniques that could generalize to other emerging programming languages. Our approach combines three key insights: (1) synthetic data generation through retrieval-augmented generation can bootstrap from minimal examples, (2) teaching models to reason about DSL primitives before code generation improves syntactic understanding, and (3) reinforcement learning with compilation feedback can further refine generation quality.

We evaluate our approach on KernelBench, a comprehensive benchmark spanning 250 GPU kernel optimization tasks. Our experiments reveal both successes and limitations: while we achieve dramatic improvements in compilation rates, functional correctness remains challenging.

2 Related Work

Recent efforts in automated GPU kernel generation have primarily focused on scaling the test-time compute of large language models with existing high-level languages. KernelBench Ouyang et al. (2024) introduced an open-source evaluation framework covering 250 PyTorch workloads and established baseline LLM performance metrics. KernelBench allows for fair comparison across different approaches and provides diverse optimization challenges spanning from basic operations to complex multi-kernel fusion tasks.

Building on KernelBench, SakanaAI developed an "AI CUDA Engineer" workflow with RAG components that automatically generates and validates kernels against benchmark tasks Lange et al. (2025). Their system achieves impressive results but at significant computational cost—approximately $15 in API credits per generated kernel. Additionally, their approach suffered from reward hacking, where models generated impractical kernels that technically passed tests but offered no real-world utility.

NVIDIA demonstrated a continuous chain-of-thought method for generating Triton kernels, showing efficiency gains on their GPUs Blog (2024). METR also created a kernel agent with blackbox LLMs, providing detailed analysis of KernelBench results and suggesting hardware-aware prompts to improve generation quality METR (2025). Mako's closed-source "kernel-agent" is another LLM-agent pipeline, though performance data and implementation details remain unavailable Mako (2025).

Evolutionary search algorithms have also shown promise for kernel optimization. Zhang et al. Zhang et al. (2024b) applied population-based perturbations to identify high-performance kernels, achieving some success on specific optimization patterns. The original KernelBench authors also demonstrate that repeated evolutionary search for kernels yields impressive performance gains `https://scalingintelligence.stanford.edu/blogs/fastkernels/`.

Some recent efforts have also trained specialized models for GPU kernel synthesis. Facebook's KernelLLM fine-tunes an 8 billion-parameter Llama 3.1 Instruct on 18,000 Triton kernels scraped from GitHub, automating PyTorch-to-Triton translations Fisches et al. (2025). Cognition's Kevin-32B employs a multi-turn RL regimen to generate and iteratively improve CUDA kernels Baronio et al. ([n. d.]). While these approaches validate the potential of LLM-driven kernel generation, they hinge on languages with preexisting data. CUDA is well-studied, and Triton, although a low-resource language, was released in 2021, so it is no longer a *very* low-resource language. In this context, our work diverges by targeting TileLang for which there are practically zero online examples.

3 Method

We used a three-stage progressive learning strategy designed to bootstrap from minimal examples to a comprehensive kernel generation system. Each stage addresses a specific challenge in learning low-resource DSLs: data scarcity, reasoning capability, and generation quality.

3.1 Stage 1: Synthetic Dataset Generation

We began with 37 correct TileLang kernels (10 from their GitHub repository + 17 written by us) covering fundamental operations including matrix multiplication, vector operations, and reductions. We annotated these initial kernels with comments explaining the mapping from PyTorch operations to TileLang constructs, serving as high-quality seed examples for our generation pipeline.

Then, to bootstrap the 37 kernels into a mini-dataset, we implemented a retrieval-augmented generation (RAG) system using DSPy Khattab et al. (2023) that treats our growing kernel collection as a dynamic knowledge base. The system operates through four key components. First, query processing analyzes input PyTorch operations to extract semantic features including operation types, tensor shapes, computational patterns, and memory access characteristics. These features enable effective similarity matching with existing kernels.

The similarity retrieval component computes embeddings for both queries and our kernel knowledge base using a pre-trained code embedding model. We retrieve the k=5 most semantically similar kernels based on cosine similarity. Then we use a context construction module to transform the retrieved kernels into structured in-context examples that demonstrate the mapping from PyTorch to TileLang.

Finally, the generation component generates new kernels by adapting patterns from retrieved examples. Through iterative application of this pipeline, we expanded our initial 37 kernels to 180 verified implementations. We hypothesize this worked because GPU kernel optimization typically involves recombining established techniques rather than inventing entirely new algorithms, and also the order in which we generated the kernels formed a natural "curriculum" from easy to hard.

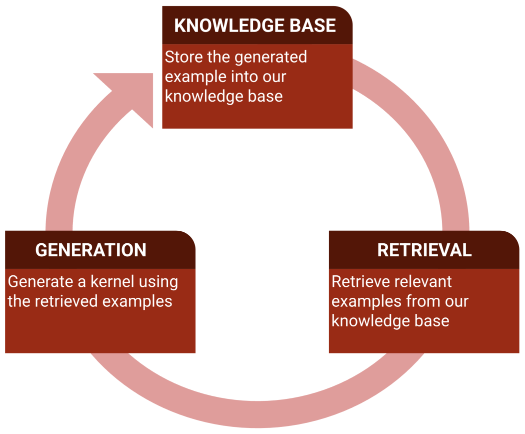

KNOWLEDGE BASE
Store the generated example into our knowledge base
GENERATION
Generate a kernel using the retrieved examples
RETRIEVAL
Retrieve relevant examples from our knowledge base

Figure 1: Our method of bootstrapping a small dataset of TileLang kernels.

3.2 Stage 2: Reasoning Alignment

While our synthetic data generation produced 180 verified TileLang kernels, this dataset was not enough for traditional supervised fine-tuning approaches. Modern code generation models typically require orders of magnitude more examples to learn syntactic patterns effectively. To address this data scarcity, we explored alternative training: teaching the model to reason about TileLang primitives and language decisions before generating code, specifically through preference alignment. Our approach was motivated by recent advances in reasoning training: Zhang et al. demonstrated that teaching models to produce good intermediate reasoning steps through preference optimization algorithms improves performance on coding tasks.

We chose Kahneman-Tversky Optimization (KTO) Ethayarajh et al. (2024) as our preference learning algorithm for several key reasons. Unlike Direct Preference Optimization (DPO) Rafailov et al. (2024), which requires paired comparisons between preferred and dispreferred outputs, KTO operates effectively with unpaired preference data, which worked perfectly for our correct reasoning vs. incorrect reasoning setting. Furthermore, recent evaluations demonstrate that KTO can match or exceed DPO's performance on logical reasoning tasks while requiring simpler data collection Saeidi et al. (2025), Kadlčík and Štefánik (2024). Also, KTO is an algorithm that can be used with an imbalanced dataset where there are more negative examples then positive examples.

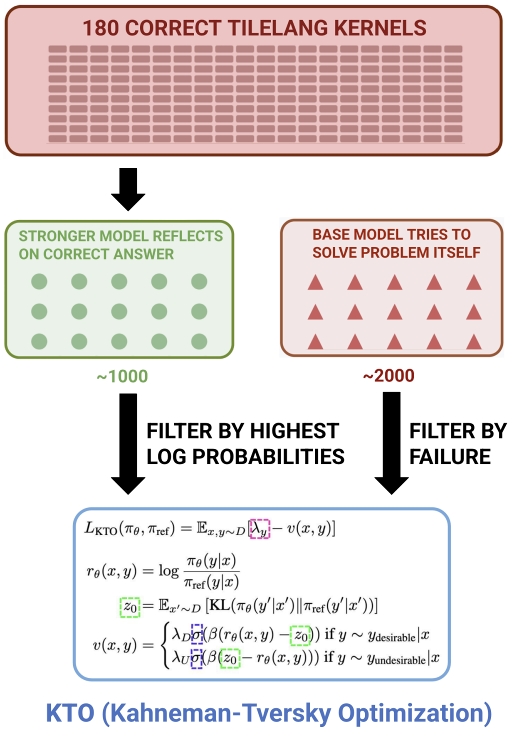
KTO reformulates the preference learning objective through the lens of prospect theory from behavioral economics. Rather than maximizing the log-likelihood ratio between preferred and dispreferred outputs as in DPO, KTO optimizes a value function that treats gains (positive examples) and losses (negative examples) asymmetrically. This formulation naturally handles the imbalanced nature of code generation, where there are many ways to write incorrect code but relatively few correct implementations. The KTO loss function incorporates a reference model to prevent distribution shift while encouraging the model to increase likelihood on positive examples and decrease it on negative ones, with different weighting schemes reflecting the asymmetric value humans place on gains versus losses.

To construct our reasoning alignment dataset, we developed a pipeline for generating and filtering synthetic reasoning traces. For each of our 180 verified kernels, we prompted a diverse pool of language models to generate detailed "post-hoc" step-by-step explanations of the kernel process, given the correct answer. We used both specialized reasoning models (DeepSeek-R1-Distill-Qwen-14B, DeepSeek-R1-Distill-Llama-70B) and general-purpose models (cogito-v1-preview-llama-8B, gpt-4.1-nano, gpt-4.1-mini, gpt-4o-mini). Each trace was required to reason about syntax decisions like the mapping from PyTorch operations to TileLang constructs and also implementation decisions like memory access patterns and thread organization.

To ensure quality and diversity in our positive examples, we sampled reasoning traces using different prompting strategies and filtered them based on their highest conditional log probability under our base model. We passed the reasoning trace through the base Seed-Coder-8B model and computed P(trace | task) for each candidate; then we retained those in the top quartile, resulting in 900 high-quality "good" reasoning traces. The intuition here is that we wanted to retain the reasoning traces with the highest likelihood of being something our base model could generate.

Then, we created "bad" reasoning traces. This step required careful consideration; simply generating random or syntactically incorrect reasoning would not teach the model to avoid realistic failure modes. Instead, we employed a two-step process. First, we had the base model attempt to solve new kernel optimization tasks independently, without access to our curated examples. These attempts typically produced plausible-sounding but flawed reasoning chains. Second, we systematically injected common reasoning errors into existing traces, such as incorrect memory access pattern analysis, flawed parallelization strategies, and misunderstandings of TileLang's execution model. We validated each negative example by confirming that following its reasoning led to compilation failures or incorrect results.

This process yielded a dataset of 2,400 reasoning traces, with a 3 to 8 ratio of positive to negative examples. By training on these traces through KTO, we hypothesized that the model would develop a deeper understanding of TileLang's primitives and computational model, improving its ability to generate syntactically correct kernels even with limited direct supervision on code generation.


180 CORRECT TILELANG KERNELS
STRONGER MODEL REFLECTS ON CORRECT ANSWER
BASE MODEL TRIES TO SOLVE PROBLEM ITSELF
~1000
~2000
FILTER BY HIGHEST LOG PROBABILITIES
FILTER BY FAILURE
KTO (Kahneman-Tversky Optimization)

$$L_{\text{KTO}}(\pi_\theta, \pi_{\text{ref}}) = \mathbb{E}_{x,y\sim D}[\boxed{\lambda_y} - v(x,y)]$$

$$r_\theta(x,y) = \log \frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)}$$

$$\boxed{z_0} = \mathbb{E}_{x'\sim D}\left[\text{KL}(\pi_\theta(y'|x')\|\pi_{\text{ref}}(y'|x'))\right]$$

$$v(x,y) = \begin{cases} \lambda_D \boxed{\sigma}(\beta(r_\theta(x,y) - \boxed{z_0})) \text{ if } y \sim y_{\text{desirable}}|x \\ \lambda_U \boxed{\sigma}(\beta(\boxed{z_0} - r_\theta(x,y))) \text{ if } y \sim y_{\text{undesirable}}|x \end{cases}$$

Figure 2: Pipeline to construct our reasoning dataset.

3.3 Stage 3: Reinforcement Learning with Verifiable Rewards

It's known that RL methods primarily act to draw out existing capabilities in base model Ferrag et al. (2025). We hypothesized our slight success with KTO was a sign that we could start reinforcement learning with verifiable rewards (RLVR), in order to transition the model's capabilities from reasoning to functional code generation. While prior fine-tuning stages provided a necessary alignment with the DSL's structure, generating kernels that are both correct and performant requires learning from the consequences of execution. We therefore formulate kernel optimization as an RL problem, where the agent's policy is refined based on rewards derived from compiling and running the code it produces.

We selected Group Relative Policy Optimization (GRPO) for this task, as its design is exceptionally well-suited to our problem's unique challenges. By learning from relative preferences within groups of solutions for a specific problem, GRPO effectively navigates the sparse reward landscape inherent in code generation; it can extract a learning signal even when attempts fail correctness tests, so long as some are demonstrably better than others. Furthermore, by treating each kernel optimization task as a distinct group, the algorithm encourages the development of problem-specific strategies rather than a one-size-fits-all policy.

To imitate the iterative nature of human code optimization, our implementation frames this process as a multi-turn RL problem. Rather than a single-shot generation, the model engages in a conversational refinement loop lasting up to four turns per episode. In each turn, it generates a kernel, receives structured feedback on its compilation and correctness, and then attempts a new solution informed by the outcome of its previous effort. The history of these attempts is fed back into the prompt, creating a rich, stateful context that allows the model to recover from initial errors and learn from its mistakes in a manner akin to a human debugging workflow.

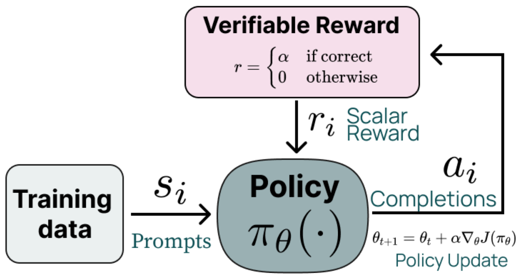

Verifiable Reward
$r = \begin{cases} \alpha & \text{if correct} \\ 0 & \text{otherwise} \end{cases}$
r_i Scalar Reward
a_i Completions
Training data
s_i Prompts
Policy $\pi_\theta(\cdot)$
$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\pi_\theta)$
Policy Update

Figure 3: RLVR (image credit to `https://allenai.org/tulu`)

The reward function is structured to create a clear learning hierarchy, guiding the model from basic syntax to functional correctness and finally to performance optimization. Based on the verifiable outcome of each generated kernel, the reward **R** is defined as:

$$R = \begin{cases} -1.0 & \text{if compilation fails} \\ -0.5 & \text{if compiles, but is incorrect} \\ 0.3 + \frac{T_{\text{baseline}}}{T_{\text{generated}}} & \text{if compiles and is correct} \end{cases}$$

This structure imposes a steep penalty for compilation failure, strongly discouraging invalid syntax. Once this is mastered, the model is incentivized to overcome the smaller but significant penalty for correctness errors. For fully successful kernels, a constant base reward is augmented with a linear speedup factor, directly driving the model's policy toward generating highly performant code.

Given the significant computational demands of this online training process, our configuration is calibrated for stability and sample efficiency. We employ a policy learning rate of 1×10^{-5} with a training batch size of 4 and 2 gradient accumulation steps. This configuration balances effective gradient updates with our memory constraints while enabling the multi-turn learning paradigm that is crucial for this complex code optimization task.

4 Experimental Setup

We evaluate 100 held-out problems from KernelBench levels 1 and 2, provided by the benchmark authors. Level 1 contains basic operations (element-wise transformations, simple reductions) while Level 2 requires complex patterns with operator fusion and memory optimization. Experiments were run on NVIDIA H100 GPUs via Modal.

Each generated kernel goes through three evaluations: compilation checking with the TileLang compiler, correctness verification against PyTorch references (relative tolerance 10^{-5}), and performance measurement (100 trials with warm-up).

We report three primary metrics: compilation success rate (percentage of error-free compilations), correctness rate (percentage of compiled kernels producing correct outputs), and pass@5 scores (probability of generating at least one correct solution in 5 attempts), following conventions we saw in related works.

Our KTO training uses learning rate 1×10^{-6} (within recommended range for $\beta = 0.1$), effective batch size 32 (gradient accumulation across 8 GPUs), 10 epochs, desirable weight 3.0, and undesirable weight 2.0.

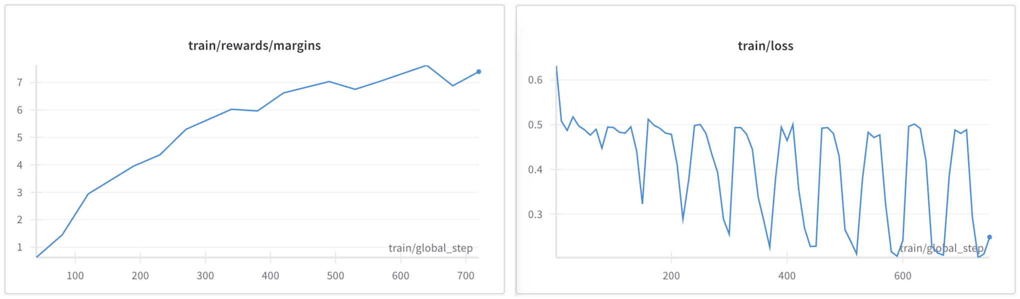


Figure 4: Reward and Loss of KTO Training

Figure 3 shows the training dynamics of our KTO stage. The reward margin between chosen and rejected traces increases monotonically from 1.0 to 7.0, indicating the model successfully learns to distinguish high-quality reasoning from flawed approaches. The training loss converges with expected oscillations characteristic of preference optimization. The increasing separation between chosen (positive) and rejected (negative) example rewards validates that our reasoning alignment effectively teaches the model to recognize correct TileLang optimization patterns.

For our GRPO training, we use a learning rate of $1e-5$, a batch size of 4, and 2 gradient accumulation steps (effective batch size of 8). To maximize sample efficiency, the model makes 4 PPO epochs over each batch of collected experiences. The training is framed as a multi-turn process where the agent has 4 attempts to refine its solution for a given problem, with 2 trajectories generated per problem to increase experience diversity. We set the generation temperature to 0.7 so that the model still explores a broader range of candidate kernels beyond its top predictions.

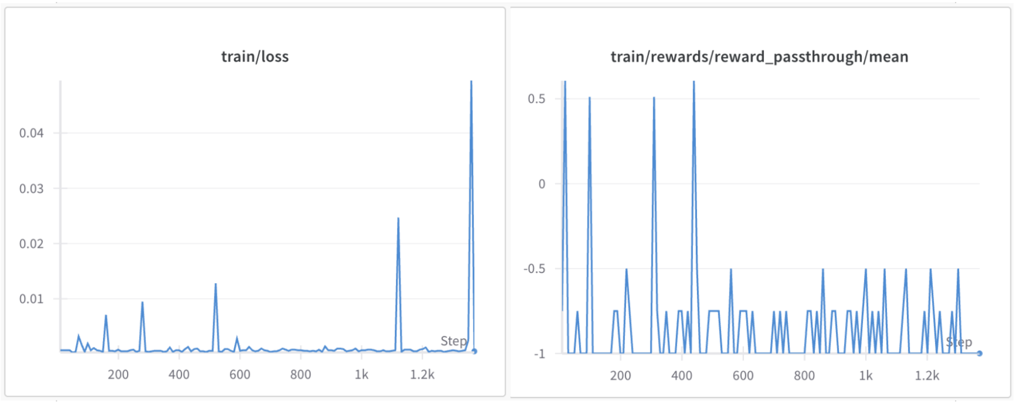


Figure 5: Loss and Reward of GRPO Training

Figure 4 illustrates the evolution of training loss and mean reward throughout the GRPO stage. The training loss remains essentially zero, reflecting the scarcity of fully correct kernel generations and resulting in very few meaningful policy updates. The mean reward curve shows occasional positive spikes, each corresponding to a task for which the model produces a compilable and correct kernel, while the vast majority of rewards lie between –1.0 and –0.5, indicating outputs that compile but fail to meet semantic requirements. This pattern confirms our expectation that GRPO primarily teaches the model to satisfy compilation constraints even when widespread correctness is unattainable.

We compare three model variants to isolate each training stage's contribution: base Seed-Coder-8B without TileLang training, the model after KTO reasoning alignment, and the final model after GRPO reinforcement learning.

5 Results

Our results show a progression in model capabilities across the three training stages, with huge improvements in compilation success but challenges in achieving functional correctness.

Run	**Compile Errors**	**Correctness Errors**	**pass@5**
Seed-Coder-8B-Base	456	44	0.00
Seed-Coder-8B-KTO	408	91	0.01
Seed-Coder-8B-KTO-GRPO	**272**	**233**	**0.01042**

Table 1: Compilation and correctness metrics across training stages. While compilation rates improve dramatically, correctness remains challenging.

5.1 Quantitative Evaluation

The base Seed-Coder-8B model, despite its general code generation capabilities, fails catastrophically on TileLang with a 91.2% compilation failure rate. This baseline performance underscores the challenge of zero-shot generation for low-resource DSLs. Even a capable code model cannot infer TileLang syntax from its general training plus a few in-context examples of TileLang.

We see that KTO reasoning alignment produces the first meaningful improvement, reducing compilation errors from 456 to 408 (improving compilation rate from 8.8% to 18.4%). This 10 percentage point improvement demonstrates that learning to reason about DSL primitives transfers to better syntactic understanding. The model begins to grasp TileLang's structure through reasoning traces, even without direct code generation training.

GRPO reinforcement learning yields the most dramatic gains, further reducing compilation errors to 272 (compilation rate of 45.6%). This represents a 5x improvement over the base model and shows the power of learning from compilation feedback. The model discovers syntactic patterns through trial and error, effectively learning TileLang's grammar through interaction with the compiler. However, correctness remains elusive across all stages (almost no improvement). We hypothesize our model excels at compilation but still struggles with correctness because our reward structure incentivizes this behavior by offering a relatively high reward for simple compilation, allowing the model to find a local optimum where it maximizes its score without attempting the much harder task of generating functionally correct code. The reward signal for compilation is dense and immediate—the code either compiles or it doesn't. In contrast, the signal for correctness is sparse. The model receives a positive reward for a correct kernel, but this single scalar value provides no gradient or guidance on why the code was correct. Furthermore, as an online algorithm, the model's low success rate for correctness starves it of positive examples to learn from, causing it to get "stuck" on problems, whereas the higher success rate for compilation provides a steady stream of data that reinforces this limited skill.

5.2 Qualitative Analysis

Examining generated kernels reveals interesting patterns in both successes and failures. The progression from base model to GRPO shows clear improvements in syntactic understanding and code structure.

Base model outputs often contain fundamental syntax errors such as mixing up TileLang primitives or even inventing non-existent TileLang keywords. For example, attempting to generate a simple convolution kernel, the base model produces code with undefined functions and incorrect tensor declarations. These outputs suggest the model tries to leverage its general programming knowledge but cannot adapt to TileLang's specific syntax.

After KTO training, the generated code shows some improvement in syntactic validity, but not too qualitatively noticeable. We do notice that the reasoning chains generated by the model seem to improve; they tend to be much longer than the base model's reasoning chains, probably due to the high-quality reasoning chains the model was trained to prefer. After GRPO however, we see much more visibly improved kernels. The model correctly uses TileLang primitives like `T.Kernel`, `T.Buffer`, and proper tensor indexing. However, semantic errors persist; buffer dimensions are often incorrect, and memory access patterns fail to account for TileLang's execution model. For a specific case study, see the Appendix.

6 Discussion

Our results show both the potential and limitations of learning low-resource programming languages through progressive training strategies. The dramatic improvement in compilation rates—from 8.8% to 45.6%—demonstrates that combining reasoning alignment with reinforcement learning can effectively teach syntactic patterns even with minimal training data. This suggests that the approach could generalize to other emerging DSLs facing similar data scarcity challenges.

The effectiveness of reasoning alignment (KTO) suggests that explicit reasoning about code generation can improve learning efficiency. Teaching models to think about programming concepts before generating code provides a form of inductive bias that accelerates syntactic learning. This approach could be particularly valuable for documentation-scarce languages where understanding the conceptual mapping from high-level operations to low-level implementations is crucial.

Our experience with GRPO and the multi-turn setup proved effective at teaching the model to recover from basic errors, mimicking a human-like debugging process for compilation issues. The model learned to iterate on its own mistakes when the feedback was clear and actionable.

However, the persistent correctness gap reveals the core challenge of our task: while the model effectively learned what valid TileLang code looks like, it failed to learn how that code must behave. This disparity is a direct result of the reward structure and the environment's mechanics. The feedback for compilation is dense and immediate—a harsh -1.0 penalty for failure—which provides a clear, strong signal to quickly master syntax. In contrast, the reward for functional correctness is sparse and uninformative. The model receives a single scalar value, which provides no insight into the underlying logical flaws of an incorrect program or the specific strengths of a correct one.

This dynamic is further reinforced by the environment's state-update rule. The agent is only permitted to advance to a new state if its generated kernel is both compiled and functionally correct. Any failure resets the agent to the previous known-good state, forcing it to retry. This entire learning process was further constrained by computational limitations. The finite number of GRPO training steps meant the model's learning was still dominated by the most frequent feedback signal—the compilation penalty. While the compilation rate improved fivefold to 45.6%, this figure indicates that the model was still actively learning the intricacies of TileLang syntax. It had not yet received a sufficient density of positive rewards for correct code to begin abstracting the principles of functional correctness. A significantly longer training run, beyond our computational budget, would have been necessary to see if the model could move past mastering syntax and begin to effectively address the more difficult challenge of semantic accuracy.

7 Conclusion

We presented a novel approach for teaching small language models to generate code in extremely low-resource domain-specific languages, using TileLang as a test case. Our three-stage pipeline demonstrates that progressive training—combining synthetic data generation, reasoning alignment, and reinforcement learning—can achieve significant improvements in compilation success even with minimal initial examples. The reduction in compilation errors from 91% to 54% is an important step toward automated DSL code generation.

Yet a program that compiles is not necessarily a program that works. The persistent gap between compilation success and semantic correctness indicates that binary compile-or-fail signals are too coarse; more informative supervision is needed to guide models toward functional behaviour rather than mere syntactic validity.

The rapid expansion of DSLs for accelerators in quantum, neuromorphic, and other emerging hardware platforms makes data-efficient code generation increasingly valuable. Our method offers a practical recipe for bootstrapping models on brand-new languages while also illustrating the obstacles that remain before small models can reliably produce correct and performant code.

Future work can move in many directions. First, applying the three-stage pipeline to additional emerging DSLs would test its generality and reveal how strongly each stage contributes in new settings. Second, systematic ablation studies are needed: removing or substituting the kernel-tuning optimization step, and comparing pure supervised fine-tuning with or without DPO, would clarify which components truly drive semantic correctness.Third, devising richer reward structures could help smaller models learn more effectively; partial-credit signals from unit-test coverage, execution traces, or hardware counters such as memory footprint and instruction count offer promising alternatives to binary compile success. Finally, scaling up data and model size may also unlock further gains. Generating a broader seed dataset with RAG and training on larger 32B models could determine whether increased capacity and denser rewards together close the gap between compilation and full semantic correctness.

8 Team Contributions

Nathan Paek: Dataset generation, KTO training, report writing.

Sokserey Sun: Dataset generation, GRPO infrastructure, report writing.

Zijian Luo: GRPO training, test sample generation and evaluation pipeline, report writing

Changes from Proposal Our original proposal targeted CUDA kernel generation, but we pivoted to TileLang for two key reasons. First, TileLang's extreme scarcity (24 examples) provided a more challenging test case for low-resource learning. Second, TileLang's simpler syntax allowed us to focus on fundamental learning challenges rather than CUDA's complex semantics.

References

Carlo Baronio, Pietro Marsella, Ben Pan, and Silas Alberti. [n. d.]. Multi-Turn RL Training for CUDA Kernel Generation. https://cognition.ai/blog/kevin-32b.

NVIDIA Developer Blog. 2024. Accelerating CUDA Kernel Generation with AI. `https://developer.nvidia.com/blog/ai-kernel-generation/` Blog post showcasing AI-generated CUDA and Triton kernels.

ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, Tao Sun, Jinhua Zhu, Shulin Xin, Dong Huang, Yetao Bai, Lixin Dong, Chao Li, Jianchong Chen, Hanzhi Zhou, Yifan Huang, Guanghan Ning, Xierui Song, Jiaze Chen, Siyao Liu, Kai Shen, Liang Xiang, and Yonghui Wu. 2025. Seed-Coder: Let the Code Model Curate Data for Itself. arXiv:2506.03524 [cs.CL] `https://arxiv.org/abs/2506.03524`

Kawin Ethayarajh, Winnie Xu, Niklas Muennighoff, Dan Jurafsky, and Douwe Kiela. 2024. KTO: Model Alignment as Prospect Theoretic Optimization. arXiv:2402.01306 [cs.LG] `https://arxiv.org/abs/2402.01306`

Mohamed Amine Ferrag, Norbert Tihanyi, and Merouane Debbah. 2025. Reasoning Beyond Limits: Advances and Open Problems for LLMs. arXiv:2503.22732 [cs.LG] `https://arxiv.org/abs/2503.22732`

Zacharias Fisches, Sahan Paliskara, Simon Guo, Alex Zhang, Joe Spisak, Chris Cummins, Hugh Leather, Joe Isaacson, Aram Markosyan, and Mark Saroufim. 2025. *KernelLLM*. `https://huggingface.co/facebook/KernelLLM` Corresponding authors: Aram Markosyan, Mark Saroufim.

Marek Kadlčík and Michal Štefánik. 2024. Self-training Language Models for Arithmetic Reasoning. arXiv:2407.08400 [cs.CL] `https://arxiv.org/abs/2407.08400`

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2023. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. arXiv:2310.03714 [cs.CL] `https://arxiv.org/abs/2310.03714`

Robert Tjarko Lange, Aaditya Prasad, Qi Sun, Maxence Faldor, Yujin Tang, and David Ha. 2025. *Agentic CUDA Kernel Discovery, Optimization and Composition*. Technical Report. SakanaAI. `https://pub.sakana.ai/ai-cuda-engineer/paper/`

Mako. 2025. GPU Kernel-Agent. Proprietary LLM-based GPU kernel synthesis pipeline (not public).

METR. 2025. Enhancing KernelBench: Analysis and Improvement Suggestions. `https://metr.io/blog/kernelbench-improvements` Blog post analyzing KernelBench and proposing enhancements.

Anne Ouyang, Simon Guo, and Azalia Mirhoseini. 2024. KernelBench: Can LLMs Write GPU Kernels? `https://github.com/ScalingIntelligence/KernelBench`. Dataset and benchmark suite for LLM-generated GPU kernels.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2024. Direct Preference Optimization: Your Language Model is Secretly a Reward Model. arXiv:2305.18290 [cs.LG] `https://arxiv.org/abs/2305.18290`

Amir Saeidi, Shivanshu Verma, Md Nayem Uddin, and Chitta Baral. 2025. Insights into Alignment: Evaluating DPO and its Variants Across Multiple Tasks. arXiv:2404.14723 [cs.CL] `https://arxiv.org/abs/2404.14723`

Benjamin F. Spector, Simran Arora, Aaryan Singhal, Daniel Y. Fu, and Christopher Ré. 2024. ThunderKittens: Simple, Fast, and Adorable AI Kernels. arXiv:2410.20399 [cs.LG] `https://arxiv.org/abs/2410.20399`

Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10–19. `https://doi.org/10.1145/3315508.3329973`

Lei Wang, Yu Cheng, Yining Shi, Zhengju Tang, Zhiwen Mo, Wenhao Xie, Lingxiao Ma, Yuqing Xia, Jilong Xue, Fan Yang, and Zhi Yang. 2025. TileLang: A Composable Tiled Programming Model for AI Systems. arXiv:2504.17577 [cs.LG] `https://arxiv.org/abs/2504.17577`

Xuan Zhang, Chao Du, Tianyu Pang, Qian Liu, Wei Gao, and Min Lin. 2024a. Chain of Preference Optimization: Improving Chain-of-Thought Reasoning in LLMs. arXiv:2406.09136 [cs.CL] https://arxiv.org/abs/2406.09136

Yijia Zhang, Zhihong Gou, Shijie Cao, Weigang Feng, Sicheng Zhang, Guohao Dai, and Ningyi Xu. 2024b. Automating Energy-Efficient GPU Kernel Generation: A Fast Search-Based Compilation Approach. arXiv:2411.18873 [cs.PF] https://arxiv.org/abs/2411.18873

9 Appendix

An example of incorrect and non-compilable Tilelang generated by the base model for Level 2 Problem 40 (matrix multiplication + scaling + residual addition). Some non-compilable artifacts include "`TL.float_16`", "`TL.float_32`", "`TL.tensor`", "`@primitive`" and "`TL.grid_launch`".

```
import torch
import torch.nn as nn
import tilelang as TL
from tilelang.decorators import jit_func, primitive

class MatVecModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.block_m = 256
        self.block_k = 16
        self.dtype = TL.float16
        self.acc_dtype = TL.float32
        self.kernel = None

    def _make_matvec_kernel(self):
        @jit_func
        @primitive
        def kern(A: TL.tensor[TL.float16, (self.block_m, self.block_k)],
                 B: TL.tensor[TL.float16, (self.block_k,)],
                 C: TL.tensor[TL.float32, (self.block_m,)]):
            with TL.grid_launch(self.block_m // 8, self.block_k // 4) as (bx, by):
                for im, km in TL.parallel_range(self.block_m, self.block_k):
                    i = bx * self.block_m + im
                    k = by * self.block_k + km
                    if i < A.shape[0] and k < B.shape[0]:
                        C[i] += A[i, k] * B[k]
                    else:
                        C[i] = 0.0
            return C
        return kern

    def forward(self, A, B):
        A = A.cuda().to(torch.float16).contiguous()
        B = B.cuda().to(torch.float16)

        if self.kernel is None:
            self.kernel = self._make_matvec_kernel()

        # Allocate output tensor with shape and dtype
        C = torch.zeros(A.shape[0], dtype=torch.float32, device='cuda')

        # Call kernel with arguments
        out = self.kernel(A, B, C)
        return out.to(torch.float32)
```

An example of compilable (but functionally incorrect) Tilelang generated by our final KTO-GRPO for Level 2 Problem 40. Though it compiles, the math it implements is wrong: it multiplies the matrices in the wrong orientation, treats the bias vector as an output tile, resets the accumulator every K-tile, and applies the scaling factor to each partial sum instead of to the finished dot product. In the code it wrote, only the last tile's contribution survives and the required $(Wx + b)(scale + 1)$ transformation is never produced.

```
import torch
import torch.nn as nn
import tilelang
import tilelang.language as T

def optimized_matrix_multiply_scale_add(M, N, K, block_M, block_N, block_K, dtype="float16"):
    @tilelang.jit(out_idx=-1)
    @T.prim_func
    def kernel(A: T.Tensor((M, K), dtype), B: T.Tensor((K, N), dtype), C: T.Tensor((M, N), dtype)):
        with T.Kernel(T.ceildiv(M, block_M), T.ceildiv(N, block_N), threads=256) as (bx, by):
            A_shared = T.alloc_shared((block_M, block_K), dtype)
            B_shared = T.alloc_shared((block_K, block_N), dtype)

            for k_block in T.Pipelined(T.ceildiv(K, block_K), num_stages=2):
                T.copy(A[bx * block_M: (bx + 1)
                     * block_M, k_block * block_K: (k_block + 1) * block_K], A_shared)
                T.copy(B[k_block * block_K: (k_block + 1) * block_K, by * block_N: (by + 1)
                     * block_N], B_shared)

                for i, j in T.Parallel(block_M, block_N):
                    acc = T.alloc_fragment((1,), dtype)
                    T.fill(acc, 0)
                    for k in T.serial(block_K):
                        acc += A_shared[i, k] * B_shared[k, j]
                    C[bx * block_M + i, by * block_N + j] = acc * scaling_factor
                         + C[bx * block_M + i, by * block_N + j]
    return kernel

class ModelNew(nn.Module):
    def __init__(self, in_features, out_features, scaling_factor):
        super(ModelNew, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.scaling_factor = scaling_factor
        self._kernel_cache = {}

    def _get_kernel(self, shape_params, dtype="float16"):
        key = (shape_params, dtype)
        if key not in self._kernel_cache:
            M, K, N = shape_params
            self._kernel_cache[key] = optimized_matrix_multiply_scale_add(M, K, N,
                 128, 128, 32, dtype)
        return self._kernel_cache[key]

    def forward(self, x):
        batch_size, in_features = x.shape
        x = x.to(device="cuda", dtype=torch.float16)
        kernel = self._get_kernel((batch_size, self.in_features, self.out_features))
        weight = self.matmul.weight.to(device="cuda", dtype=torch.float16)
        bias = self.matmul.bias.to(device="cuda", dtype=torch.float16)
        output = kernel(x, weight, bias)
        return output.to(torch.float32)
```