DSL-MONKEYS: SELF-GENERATED IN-CONTEXT EXAMPLES FOR LOW-RESOURCE GPU DSL KERNELS

Nathan Paek[1,*] Simon Guo[1,*] Vishnu Sarukkai[1,*] Willy Chan[1,*] William Hu[1]
Ethan Boneh[1] Simran Arora[2] Ludwig Schmidt[1] Kayvon Fatahalian[1,†] Azalia Mirhoseini[1,†]

[1]Computer Science Department, Stanford University [2]Together AI

[*]Equal contribution [†]Equal advising
Corresponding authors: <nathanjp, simonguo, vsarukkai, willyc>@stanford.edu

ABSTRACT

Domain-specific languages (DSLs) for GPU kernels such as TileLang and ThunderKittens remain challenging for LLMs due to scarce training data and limited documentation. Standard test-time techniques (repeated sampling, iterative refinement, evolutionary search) treat each kernel independently, missing a key insight: successful implementations encode reusable DSL usage patterns directly applicable to new tasks. We introduce DSL-Monkeys, a batched test-time approach that considers an entire collection of kernels jointly rather than individually. DSL-Monkeys maintains a growing archive of successfully-implemented kernels and retrieves them as in-context demonstrations for remaining tasks, resembling a form of test-time reinforcement learning—improving on future tasks by curating a retrieval database rather than updating weights. Starting from just 3 seed examples, DSL-Monkeys achieves 71% and 18% correctness on 200 KernelBench Level 1-2 tasks for TileLang and ThunderKittens respectively, compared to baseline pass@1 rates of $< 20\%$ and $< 1\%$. Secondly, with curriculum decomposition, DSL-Monkeys solves complex linear attention variants that approach or match human-expert-written Triton performance. Thirdly, we explore scaling DSL-Monkeys to 10K PyTorch programs from GitHub, producing thousands of DSL implementations that address DSL data scarcity. We show that fine-tuning improves Qwen's TileLang implementation ability via both distillation and self-generated data, a potential path towards improving LLMs' code generation ability with emerging DSLs.

1 INTRODUCTION

Many software domains and applications rely on specialized libraries and domain-specific languages (DSLs) that are poorly represented in LLM training corpora (Joel et al., 2025; Khandpur et al., 2024; Liu et al., 2023; Cassano et al., 2022). In these "low-resource" settings, LLMs lack knowledge about target DSLs' syntax and common programming idioms, leading to poor code generation performance.

DSLs for authoring high-performance GPU kernels for ML such as CuTe (NVIDIA, 2026), TileLang (Wang et al., 2025), ThunderKittens (Spector et al., 2024b), and Triton-TLX (Experimental, 2025) exemplify this challenge. They offer the promise of productivity and performance, but are only sparsely documented (<1 million public tokens) and introduce syntax and programming models that are rare in training corpora. For these reasons, although there is significant interest in harnessing the benefits of these frameworks in production settings (FlashAttention-4 and DeepSeek's Sparse Attention (Dao-AILab, 2024a; DeepSeek-AI, 2025) are successful recent examples), LLMs struggle to automatically generate correct kernels using these frameworks. For pass@1, frontier models correctly implement less than $< 20\%$ of KernelBench's Level 1-2 (Ouyang et al., 2025a) kernels, which test basic kernel generation and optimization from a PyTorch reference, when targeting TileLang, and less than 1% for ThunderKittens.

Since scarce DSL training data limits improvement via fine-tuning directly, scaling test-time compute is the natural approach given the presence of verifier. For instance, standard approaches include

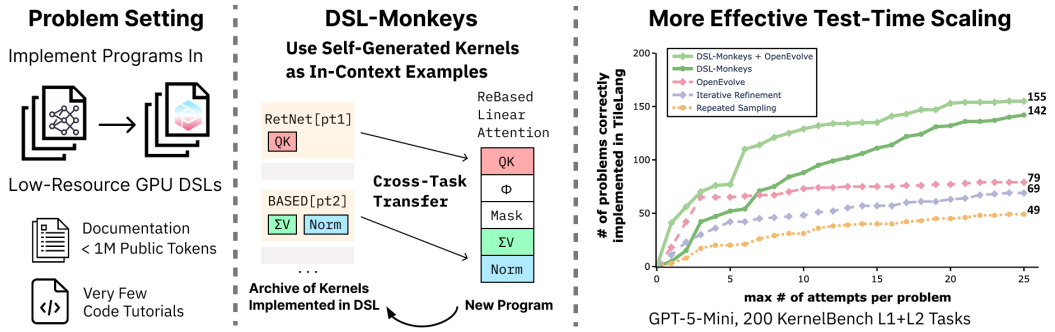

Figure 1: **DSL-Monkeys implements kernels in low-resource DSLs via cross-task transfer.**

repeated sampling Brown et al. (2024), iterative refinement based on feedback from compiler or profiler feedback, or evolutionary search (Sharma, 2025a), which boosts correct implementation rate on KernelBench's Level 1-2 to 60% (TileLang) and 17% (ThunderKittens). We argue that these prior test-time approaches are limited because they force the LLM to focus on implementing a *single kernel* at a time. However, deep learning kernels share related operators and implementation patterns: tiling strategies, memory layouts, and reductions. Therefore, when an LLM successfully implements a kernel in the target DSL, that success is likely to be a helpful example for future implementations. Given this observation, we ask: *when implementing kernels in a new DSL, can successful solutions from related kernels be leveraged at test-time to solve the remaining ones?*

Rather than developing advanced techniques for implementing a single kernel (and repeating this procedure individually for all kernels in a collection), our solution **DSL-Monkeys** adopts a simple approach that considers the entire collection as a whole: it repeatedly iterates over the collection, attempting to reimplement a batch of kernels in the target DSL. Successfully implemented kernels are logged in an archive. In future iterations, when attempting previously failed kernels, the system retrieves solutions for similar kernels from the archive and provides them as in-context examples. This formulation resembles label propagation: given unlabeled data (kernels to port) and initial labels (a small seed set from DSL documentation), the system propagates knowledge to similar kernels via in-context demonstrations, using a verifier (compilation + tests) for validation. We call this phenomenon *cross-task transfer*: using solved examples to enable solutions for new, related tasks.

Overall, we demonstrate that our test-time compute approach is a simple method that effectively implements ML kernels in low-resource DSLs. Specifically, we show:

1. **Self-Generated In-Context Examples Help Write Kernels in New DSLs.** Starting from just 3 seed examples, DSL-Monkeys using GPT-5-mini correctly implements 71% of the 200 KernelBench Level 1–2 tasks in TileLang and 18% in ThunderKittens, outperforming test-time methods that attempt each kernel independently (Figure 1). Combined with OpenEvolve (Sharma, 2025a), it achieves 77.5% in TileLang.
2. **Curriculum Decomposition Enables Solving Complex Kernels.** For challenging kernels like linear attention variants, we propose automated decomposition into simpler constituent parts. This allows DSL-Monkeys to bootstrap solutions for novel attention variants where other test-time methods fail, achieving parity with expert-written Triton implementations.
3. **Preliminary Findings from Scaling Up Data Generation for Rare DSLs.** We scale DSL-Monkeys to 10K PyTorch programs from GitHub, successfully implementing 4.7K in TileLang and 1.1K in ThunderKittens. Fine-tuning Qwen-30B-A3B on this dataset improves KernelBench L1+2 accuracy from 17% to 61% when used with DSL-Monkeys, pointing toward a data flywheel for emerging DSLs (Appendix 5).

2 Problem Statement

Implementing Programs in Novel DSLs. We consider implementing a collection of PyTorch programs in a target DSL, a common challenge when new DSLs emerge. Each DSL has only a few seed examples and a verifier (compiler + correctness tests). Unlike standard benchmarks that treat each program independently, our goal is to maximize successful implementations across programs.

The Low-Resource DSL Challenge. Emerging GPU DSLs exemplify this challenge. Table 1 shows frameworks like TileLang and ThunderKittens have fewer than 1M publicly available tokens. Figure 2

DSL	Language	Prog. Model	# Tokens
TileLang	Python	Block	733K
ThunderKittens	C++	Warp	395K
CuTe-DSL	Python	Thread	237K
Triton TLX	Python	Warp	182K
CUDA	C++	Thread	361M
Triton	Python	Block	3M

Table 1: **Estimates of Public Tokens for Emerging GPU DSLs**. We characterize GPU DSLs (Appendix B) by syntactic language, conceptual programming level, and number of high-quality tokens. Details in Appendix C.

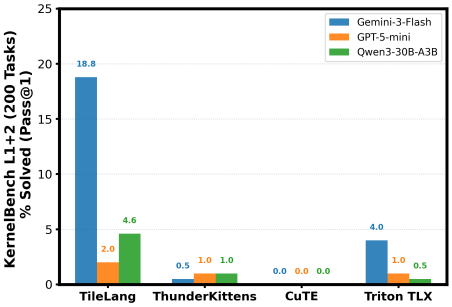

Figure 2: **Models struggle to generate kernels in emerging GPU DSLs.** We measure pass@1 (N=5) on KernelBench Level 1+2 tasks across DSLs. Even frontier models achieve <20% on TileLang and <2% on ThunderKittens.

demonstrates the impact: frontier models achieve below 20% pass@1 on KernelBench Level 1-2 for TileLang and below 2% for ThunderKittens.

Cross-Task Transfer Opportunity. Standard test-time strategies (Table 2) apply methods independently to each kernel. While execution feedback helps (OpenEvolve: 39.5% vs repeated sampling: 24.5% with GPT-5-Mini), these approaches discard successful solutions after verification rather than leveraging them for subsequent tasks. We hypothesize that successful generations contain reusable knowledge applicable to unsolved tasks. To test this, we analyze the 200 KernelBench tasks where repeated sampling (N=25) with Gemini-3-Flash solves 90 kernels. For each of the 110 failures, we check whether required DSL operators and patterns appear in solved kernels. **70% of failed kernels have all required components present in other solved kernels** (91% average coverage). While 2% require novel patterns, the knowledge needed to solve most failures already exists within the batch—it is simply inaccessible when treating tasks independently. This motivates maintaining an archive of successful solutions and retrieving relevant examples for unsolved tasks.

3 DSL-Monkeys: Self-Generated Examples for DSL Kernels

Standard test-time approaches treat each problem in isolation, failing to leverage knowledge discovered in related tasks. We introduce DSL-Monkeys, which maintains a persistent archive of verified solutions and retrieves relevant examples for new problems, enabling cross-task knowledge transfer.

DSL-Monkeys Algorithm. DSL-Monkeys follows an iterative bootstrapping process (Algorithm 1). We initialize an archive with 3 seed examples from DSL documentation and maintain a set of unsolved problems. At each iteration, we process unsolved problems in mini-batches. For each problem, we retrieve 3 examples from the archive, alternating between cosine similarity and random selection to balance exploitation and exploration (App. I.1). These serve as in-context demonstrations for kernel generation. Generated kernels are validated; passing kernels are added to the archive and marked as solved. Critically, newly solved kernels immediately become retrieval candidates for subsequent mini-batches, enabling intra-iteration bootstrapping.

DSL-Monkeys Effectively Leverages Test-Time Compute. We evaluate DSL-Monkeys against all baselines under identical conditions: 3 seed examples, 25 iterations, temperature 1.0. Table 2 shows results on TileLang and ThunderKittens. Cross-task retrieval substantially outperforms repeated sampling: on TileLang, out of 200 KernelBench tasks, DSL-Monkeys correctly implements 140 tasks with Gemini-3-Flash versus 90 for repeated sampling, and 142 with GPT-5-Mini versus 49. On ThunderKittens, DSL-Monkeys achieves 32 versus 7 for repeated sampling. Cross-task retrieval also outperforms methods using execution feedback on individual problems—on TileLang with Gemini-3-Flash, DSL-Monkeys (140) surpasses Iterative Refinement (102) and OpenEvolve (120), suggesting that retrieving relevant examples from related tasks provides better guidance than compiler errors from the current task alone. Finally, the 155 tasks solved by combining DSL-Monkeys and OpenEvolve (G) show that cross-task retrieval and same-task execution feedback are complementary.

How Retrieval Enables Bootstrapping. We analyze how cross-task retrieval enables knowledge transfer by examining code borrowing patterns and novel example impact. Using LLM-based analysis

Algorithm 1 DSL-Monkeys: In-Context Bootstrapping for Kernel Generation

$\mathcal{A} \leftarrow$ SEEDEXAMPLES(), $\mathcal{U} \leftarrow$ ALLPROBLEMS() ▷ Init archive, unsolved
for iteration $t = 1$ to T_{max} **do**
  for each mini-batch $B \subseteq \mathcal{U}$, problem $p \in B$ **do**
    if $t \bmod 2 = 1$ **then**
      $E \leftarrow$ RETRIEVESIMILAR($\mathcal{A}, p, k = 3$) ▷ Cosine sim. retrieval on odd iterations
    else
      $E \leftarrow$ RETRIEVERANDOM($\mathcal{A}, p, k = 3$) ▷ Random retrieval on even iterations
    kernel $\leftarrow$ GENERATEKERNEL(LLM, p, E)
    if TESTKERNEL(*kernel*) **then**
      $\mathcal{A} \leftarrow \mathcal{A} \cup \{(p, \textit{kernel})\}$ ▷ Check correctness, add to archive
      $\mathcal{U} \leftarrow \mathcal{U} \setminus \{p\}$ ▷ Mark solved
return $\mathcal{A}$

Test-Time Method	Seeds	TileLang		ThunderKittens	
		gemini-3-flash	gpt-5-mini	gemini-3-flash	gpt-5-mini
Repeated Sampling	3	90	49	7	27
Iterative Refinement	3	102	69	17	25
OpenEvolve	3	120	79	36	34
DSL-Monkeys	3	**140**	142	32	**36**
DSL-Monkeys	8	**140**	**152**	**41**	35
DSL-Monkeys+OpenEvolve	3	155	155	35	39

Table 2: **DSL-Monkeys improves DSL kernel synthesis. Results on implementing 200 KernelBench L1-2 tasks correctly (25 attempts).** DSL-Monkeys outperforms baselines (142/200 with GPT-5-Mini, 3 seeds) and combines well with OpenEvolve (155/200 on TileLang).

(Gemini-3-Flash, App. L.1), we identify conceptual code patterns borrowed from examples. On first attempts, successful kernels borrow 60% of code from examples vs. 51% for failures. By iteration 3, successes maintain 60% borrowing while failures show 49%. The fixed-seed baseline shows 49% for both, suggesting when examples lack relevant patterns, borrowing cannot increase even when helpful. Success rate increases from 2.0% when no retrieved examples change to 20.8% when all three are replaced, validating that failures persist due to missing relevant examples (App. K). Remaining failures are mostly synthetic compositions unrepresentative of real-world kernels (App. M).

4 CASE STUDY: SYNTHESIZING LINEAR ATTENTION KERNELS

Having demonstrated DSL-Monkeys's effectiveness on KernelBench, we evaluate its transfer to real-world kernels. We focus on 12 linear attention variants (Katharopoulos et al., 2020) from the Flash Linear Attention repository (Yang & Zhang, 2024a), including Based (Arora et al., 2025), Retention (Sun et al., 2023), and Chunked formulations—critical primitives in foundation models like Kimi Linear (Team et al., 2025) and Qwen-next (Qwen Team). These kernels are substantially more complex than KernelBench, requiring intricate recurrent formulations, stateful computations, and expert-level fusion patterns. TileLang has shown promise in this setting (DeepSeek-AI, 2025).

Standard approaches fail: Repeated sampling solves 1/12 tasks; OpenEvolve solves 0/12. Even using the 140 TileLang solutions from Section 3 as a starting archive, DSL-Monkeys solves only 2/12; these kernels are out-of-distribution from KernelBench's standalone primitives. While 85% of attempts compile, they fail due to errors in recurrent state propagation and complex fusion logic.

Curriculum learning via task decomposition. We automatically decompose each kernel into 3–9 interpretable subproblems (Appendix N), such as 'compute recurrent state update,' or 'fuse causal masking with projection,' to bridge the distribution gap. DSL-Monkeys solves each subproblem using the evolving archive as stepping stones, enabling solving 4/12 variants (Based, Retention, Chunked, ReBased). These implementations exceed 200 lines and coordinate 7+ optimization techniques.

Generated kernels achieve expert-level performance. The solved kernels demonstrate sophisticated optimizations including tensor core utilization and pipelining. Figure 3 shows our Chunked Linear

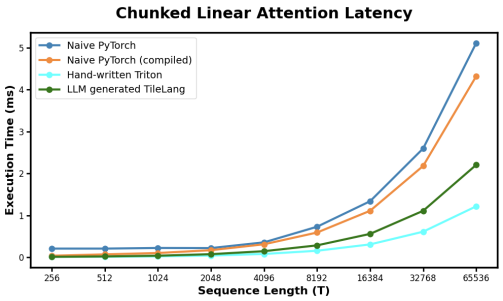

Figure 3: **DSL-Monkeys successful implements Linear Attention Variants that surpass `torch.compile`.** Chunkwise Linear Attention (Yang & Zhang, 2024a) in TileLang approaches competitive performance with hand-written Triton.

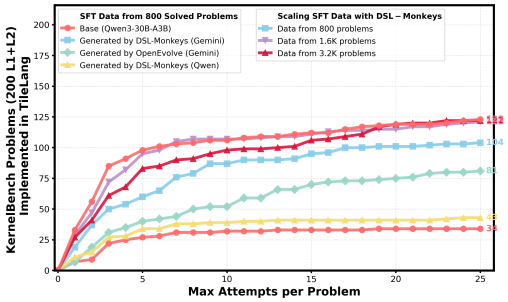

Figure 4: **DSL-Monkeys-generated data from KernelBook teaches Qwen to leverage cross-task transfer.** We show that fine-tuning `Qwen3-30B-A3B` improves its downstream performance on KernelBench, via both distillation and self-generated data (Appendix 5).

Attention achieves 2.1× speedup over Naive PyTorch, outperforms compiled PyTorch, and approaches hand-tuned Triton. DSL-Monkeys outperforms human code by 15% on some formulations (Figure 6), discovering optimizations beyond the training corpus. While direct transfer is limited for OOD problems, DSL-Monkeys with targeted curriculum construction scales to production kernels that approach or match expert programmers (Appendix N).

5 Scaling Data Unlocks Cross-Task Learning

Generating DSL Programs at Scale. Previous sections demonstrated the effectiveness of DSL-Monkeysin implementing collections of programs in novel DSLs—200 KernelBench tasks (Section 3) and 10 linear attention variants (Section 4), showing the value of cross-task transfer paired with curriculum. Can we scale to more tasks, capturing greater variety, and generate corresponding DSL code? We turn to the massively available PyTorch programs on the internet, such as KernelBook (Paliskara & Saroufim, 2025), a collection of PyTorch programs from The Stack (Kocetkov et al., 2022) that are executable through `torch.compile`. We run DSL-Monkeys on this collection of 10,082 valid KernelBook programs (after decontamination against KernelBench) with up to 5 attempts per task. For TileLang, we solve 4.7K programs with `Gemini-3-Flash` and 1.1K with `Qwen3-30B-A3B`. For the more challenging ThunderKittens, we solve 500 with `Gemini-3-Flash`.

This process creates a growing **dataset** of high-quality, annotated code samples for these low-resource DSLs: each successfully implemented program provides a trajectory containing (PyTorch Specification, Implementation Plan, verified DSL kernel). As emerging DSLs each have $< 1M$ public tokens (Section 2), these trajectories represent a significant addition to publicly available code data, totaling 14M tokens of verified code paired with reasoning traces.

Enable Weaker Model to Leverage Cross-Task Learning. Leveraging cross-task transfer is highly effective in this low-resource regime, as demonstrated with frontier models in Section 3. However, weaker models struggles: `Qwen3-30B-A3B` solves only 34 KernelBench Level 1+2 problems and plateaus quickly, even after 25 test-time iterations with DSL-Monkeys(Figure 4). How could we enable weaker models to leverage cross-task transfer and unlock test-time compute effectively?

The data generated at scale from KernelBook can bridge this gap. Fine-tuning Qwen (see Appendix J) on expert (Gemini) trajectories from just 800 solved problems improves test-time performance from 34 → 104. Fine-tuning on the same number of OpenEvolve-generated solutions also helps (34→ 81), but DSL-Monkeystrajectories are more effective, as they explicitly teach model to take advantage of cross-task transfer beyond improving its abilty in implementing in DSL. Further data scaling reaches 120+, approaching the teacher model's 140. Training Qwen on its own DSL-Monkeys-generated trajectories (1.6K samples) also yields gains (34 → 43). While smaller, this shows models can bootstrap DSL capability without a stronger teacher, pointing toward a data flywheel as open-source models continue to improve.

c

6 ACKNOWLEDGEMENTS

We gratefully acknowledge support from the Stanford HAI–GCP Cloud Credits for Research program, the Thinking Machines Lab Tinker Research Grant, OpenAI, Cognition AI, and Modal Labs for providing compute and credits that supported this project.

We thank Etash Guha, John Yang, Allen Nie, Alex Zhang, Neil Band, Sahan Paliskara, Stuart Sul, and Fredrik Kjolstad for their constructive feedback during the composition of the paper. We would also like to thank our collaborators at the Stanford Artificial Intelligence Laboratory (SAIL) and TogetherAI.

REFERENCES

Yaroslav Aksenov, Nikita Balagansky, Sofia Maria Lo Cicero Vaina, Boris Shaposhnikov, Alexey Gorbatovski, and Daniil Gavrilov. Linear transformers with learnable kernel functions are better in-context models, 2024. URL https://arxiv.org/abs/2402.10644.

Ekin Akyürek, Dale Schuurmans, Jacob Andreas, Tengyu Ma, and Denny Zhou. What learning algorithm is in-context learning? investigations with linear models. *arXiv preprint arXiv:2211.15661*, 2022.

Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, pp. 929–947, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703850. doi: 10.1145/3620665.3640366. URL https://doi.org/10.1145/3620665.3640366.

Simran Arora, Sabri Eyuboglu, Michael Zhang, Aman Timalsina, Silas Alberti, Dylan Zinsley, James Zou, Atri Rudra, and Christopher Ré. Simple linear attention language models balance the recall-throughput tradeoff, 2025. URL https://arxiv.org/abs/2402.18668.

Ian Barber. Cute-dsl. https://ianbarber.blog/2025/07/04/cute-dsl/, 2025. URL https://ianbarber.blog/2025/07/04/cute-dsl/. Ian's Blog, accessed January 2026.

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn rl for generating cuda kernels, 2025. URL https://arxiv.org/abs/2507.11948.

Dan Biderman, Jacob Portes, Jose Javier Gonzalez Ortiz, Mansheej Paul, Philip Greengard, Connor Jennings, Daniel King, Sam Havens, Vitaliy Chiley, Jonathan Frankle, Cody Blakeney, and John P. Cunningham. Lora learns less and forgets less, 2024. URL https://arxiv.org/abs/2405.09673.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling, 2024. URL https://arxiv.org/abs/2407.21787.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. Multipl-e: A scalable and extensible approach to benchmarking neural code generation, 2022. URL https://arxiv.org/abs/2208.08227.

Audrey Cheng, Shu Liu, Melissa Pan, Zhifei Li, Bowen Wang, Alex Krentsel, Tian Xia, Mert Cemri, Jongseok Park, Shuo Yang, Jeff Chen, Lakshya Agrawal, Aditya Desai, Jiarong Xing, Koushik Sen, Matei Zaharia, and Ion Stoica. Barbarians at the gate: How ai is upending systems research, 2025. URL https://arxiv.org/abs/2510.06189.

Christopher Choy. Cutedsl basics. https://chrischoy.org/posts/cutedsl-basics/, 2025. URL https://chrischoy.org/posts/cutedsl-basics/. Online technical note, accessed January 2026.

Colfax International Research. A user's guide to flexattention in flashattention & cutedsl. https://research.colfax-intl.com/a-users-guide-to-flexattention-in-flash-attention-cute-dsl/, 2025. URL https://research.colfax-intl.com/a-users-guide-to-flexattention-in-flash-attention-cute-dsl/. Colfax International Research blog, accessed January 2026.

Dao-AILab. Flash-attention. https://github.com/Dao-AILab/flash-attention, 2024a. URL https://github.com/Dao-AILab/flash-attention.

Dao-AILab. Quack. https://github.com/Dao-AILab/quack, 2024b. URL https://github.com/Dao-AILab/quack. GitHub repository.

DeepSeek-AI. Deepseek-v3.2-exp: Boosting long-context efficiency with deepseek sparse attention, 2025.

Ryan Ehrlich, Bradley Brown, Jordan Juravsky, Ronald Clark, Christopher Ré, and Azalia Mirhoseini. Codemonkeys: Scaling test-time compute for software engineering, 2025. URL https://arxiv.org/abs/2501.14723.

Facebook Experimental. Triton tlx: Tensor language extensions, 09 2025. URL https://github.com/facebookexperimental/triton/tree/tlxa.

Zacharias V. Fisches, Sahan Paliskara, Simon Guo, Alex Zhang, Joe Spisak, Chris Cummins, Hugh Leather, Gabriel Synnaeve, Joe Isaacson, Aram Markosyan, and Mark Saroufim. Kernelllm: Making kernel development more accessible, 6 2025. URL https://huggingface.co/facebook/KernelLLM.

Etash Guha, Ryan Marten, Sedrick Keh, Negin Raoof, Georgios Smyrnis, Hritik Bansal, Marianna Nezhurina, Jean Mercat, Trung Vu, Zayne Sprague, Ashima Suvarna, Benjamin Feuer, Liangyu Chen, Zaid Khan, Eric Frankel, Sachin Grover, Caroline Choi, Niklas Muennighoff, Shiye Su, Wanjia Zhao, John Yang, Shreyas Pimpalgaonkar, Kartik Sharma, Charlie Cheng-Jie Ji, Yichuan Deng, Sarah Pratt, Vivek Ramanujan, Jon Saad-Falcon, Jeffrey Li, Achal Dave, Alon Albalak, Kushal Arora, Blake Wulfe, Chinmay Hegde, Greg Durrett, Sewoong Oh, Mohit Bansal, Saadia Gabriel, Aditya Grover, Kai-Wei Chang, Vaishaal Shankar, Aaron Gokaslan, Mike A. Merrill, Tatsunori Hashimoto, Yejin Choi, Jenia Jitsev, Reinhard Heckel, Maheswaran Sathiamoorthy, Alexandros G. Dimakis, and Ludwig Schmidt. Openthoughts: Data recipes for reasoning models, 2025. URL https://arxiv.org/abs/2506.04178.

HazyResearch. Thunderkittens. https://github.com/HazyResearch/ThunderKittens, 2024. URL https://github.com/HazyResearch/ThunderKittens. GitHub repository.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021. URL https://arxiv.org/abs/2106.09685.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems, 2025. URL https://arxiv.org/abs/2408.08435.

Sathvik Joel, Jie JW Wu, and Fatemeh H. Fard. A survey on llm-based code generation for low-resource and domain-specific programming languages, 2025. URL https://arxiv.org/abs/2410.03981.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention, 2020. URL https://arxiv.org/abs/2006.16236.

Kabir Khandpur, Kilian Lieret, Carlos E. Jimenez, Ofir Press, and John Yang. Swe-bench multilingual, 2024. URL https://huggingface.co/datasets/SWE-bench/SWE-bench_Multilingual.

kiui.moe. Cute — cuda dsl documentation. https://note.kiui.moe/cuda/cute/#print, 2025. URL https://note.kiui.moe/cuda/cute/#print. Online technical note, accessed January 2026.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The stack: 3 tb of permissively licensed source code. *Preprint*, 2022.

Itay Levy, Ben Bogin, and Jonathan Berant. Diverse demonstrations improve in-context compositional generalization. *arXiv preprint arXiv:2212.06800*, 2022.

Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, Haojie Wang, Jianrong Wang, Xu Han, Zhiyuan Liu, and Maosong Sun. Tritonbench: Benchmarking large language model capabilities for generating triton operators, 2025. URL https://arxiv.org/abs/2502.14752.

Gang Liao, Hongsen Qin, Ying Wang, Alicia Golden, Michael Kuchnik, Yavuz Yetim, Jia Jiunn Ang, Chunli Fu, Yihan He, Samuel Hsia, Zewei Jiang, Dianshi Li, Uladzimir Pashkevich, Varna Puvvada, Feng Shi, Matt Steiner, Ruichao Xiao, Nathan Yan, Xiayu Yu, Zhou Fang, Abdul Zainul-Abedin, Ketan Singh, Hongtao Yu, Wenyuan Chi, Barney Huang, Sean Zhang, Noah Weller, Zach Marine, Wyatt Cook, Carole-Jean Wu, and Gaoxiang Liu. Kernelevolve: Scaling agentic kernel coding for heterogeneous ai accelerators at meta, 2025. URL https://arxiv.org/abs/2512.23236.

Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for gpt-3? *arXiv preprint arXiv:2101.06804*, 2021.

Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. Verilogeval: Evaluating large language models for verilog code generation, 2023. URL https://arxiv.org/abs/2309.07544.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder 2 and the stack v2: The next generation, 2024.

Thinking Machines. Announcing tinker: An llm-fine-tuning ecosystem. https://thinkingmachines.ai/blog/announcing-tinker/, 2025. Accessed: 2026-01-22.

Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites, 2015. URL https://arxiv.org/abs/1504.04909.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

NVIDIA. Cutlass python dsl documentation. https://docs.nvidia.com/cutlass/latest/media/docs/pythonDSL/cute_dsl.html, 2025. URL https://docs.nvidia.com/cutlass/latest/media/docs/pythonDSL/cute_dsl.html. NVIDIA official documentation.

NVIDIA. Cutlass: Cuda templates and python dsls for high-performance linear algebra, 2026. URL https://github.com/NVIDIA/cutlass. includes CuTe DSL — a Python domain-specific language for GPU code generation.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels?, 2025a. URL https://arxiv.org/abs/2502.10517.

Anne Ouyang, Azalia Mirhoseini, and Percy Liang. Surprisingly fast ai-generated kernels we didn't mean to publish (yet). https://crfm.stanford.edu/2025/05/28/fast-kernels.html, May 2025b.

Sahan Paliskara and Mark Saroufim. Kernelbook, 5 2025. URL https://huggingface.co/datasets/GPUMODE/KernelBook.

Chengwei Qin, Aston Zhang, Chen Chen, Anirudh Dagar, and Wenming Ye. In-context learning with iterative demonstration selection. *arXiv preprint arXiv:2310.09881*, 2023.

Qwen Team. Qwen3-coder-next technical report. Technical report. URL https://github.com/QwenLM/Qwen3-Coder/blob/main/qwen3_coder_next_tech_report.pdf. Accessed: 2026-02-03.

Vishnu Sarukkai, Zhiqiang Xie, and Kayvon Fatahalian. Self-generated in-context examples improve LLM agents for sequential decision-making tasks. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=WdL3O58gde.

John Schulman and Thinking Machines Lab. Lora without regret. *Thinking Machines Lab: Connectionism*, 2025. doi: 10.64434/tml.20250929. https://thinkingmachines.ai/blog/lora/.

Asankhaya Sharma. Openevolve: an open-source evolutionary coding agent, 2025a. URL https://github.com/algorithmicsuperintelligence/openevolve.

Asankhaya Sharma. Automated discovery of high-performance gpu kernels with openevolve. https://huggingface.co/blog/codelion/openevolve-gpu-kernel-discovery, June 27 2025b. Hugging Face blog post.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. URL https://arxiv.org/abs/2303.11366.

Benjamin Spector, Aaryan Singhal, Simran Arora, and Christopher Ré. Gpus go brrr. https://hazyresearch.stanford.edu/blog/2024-05-12-tk, 2024a. URL https://hazyresearch.stanford.edu/blog/2024-05-12-tk. Hazy Research blog post, May 12, 2024; introduces the ThunderKittens embedded DSL.

Benjamin F. Spector, Simran Arora, Aaryan Singhal, Daniel Y. Fu, and Christopher Ré. Thunderkittens: Simple, fast, and adorable ai kernels, 2024b. URL https://arxiv.org/abs/2410.20399.

Stuart Sul. 1.5x faster moe training with custom mxfp8 kernels. https://cursor.com/blog/kernels, August 2025. URL https://cursor.com/blog/kernels. Achieving a 3.5x MoE layer speedup with a complete rebuild for Blackwell GPUs.

Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and Furu Wei. Retentive network: A successor to transformer for large language models, 2023. URL https://arxiv.org/abs/2307.08621.

Garrett Tanzer, Mirac Suzgun, Eline Visser, Dan Jurafsky, and Luke Melas-Kyriazi. A benchmark for learning to translate a new language from one grammar book, 2024. URL https://arxiv.org/abs/2309.16575.

Gemini Team. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context, 2024. URL https://arxiv.org/abs/2403.05530.

Kimi Team, Yu Zhang, Zongyu Lin, Xingcheng Yao, Jiaxi Hu, Fanqing Meng, Chengyin Liu, Xin Men, Songlin Yang, Zhiyuan Li, Wentao Li, Enzhe Lu, Weizhou Liu, Yanru Chen, Weixin Xu, Longhui Yu, Yejie Wang, Yu Fan, Longguang Zhong, Enming Yuan, Dehao Zhang, Yizhi Zhang, T. Y. Liu, Haiming Wang, Shengjun Fang, Weiran He, Shaowei Liu, Yiwei Li, Jianlin Su, Jiezhong Qiu, Bo Pang, Junjie Yan, Zhejun Jiang, Weixiao Huang, Bohong Yin, Jiacheng You, Chu Wei, Zhengtao Wang, Chao Hong, Yutian Chen, Guanduo Chen, Yucheng Wang, Huabin Zheng, Feng

Wang, Yibo Liu, Mengnan Dong, Zheng Zhang, Siyuan Pan, Wenhao Wu, Yuhao Wu, Longyu Guan, Jiawen Tao, Guohong Fu, Xinran Xu, Yuzhi Wang, Guokun Lai, Yuxin Wu, Xinyu Zhou, Zhilin Yang, and Yulun Du. Kimi linear: An expressive, efficient attention architecture, 2025. URL https://arxiv.org/abs/2510.26692.

Qwen Team. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

TileLang Developers. Tilelang. https://tilelang.com/, 2025. URL https://tilelang.com/. Official project website.

Philippe Tillet, H. T. Kung, and David Cox. 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*, MAPL 2019, pp. 10–19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367196. doi: 10.1145/3315508.3329973. URL https://doi.org/10.1145/3315508.3329973.

Lukas Veitner. An applied introduction to cutedsl. https://veitner.bearblog.dev/an-applied-introduction-to-cutedsl/, 2024. URL https://veitner.bearblog.dev/an-applied-introduction-to-cutedsl/. Bear Blog, accessed January 2026.

Johannes Von Oswald, Eyvind Niklasson, Ettore Randazzo, João Sacramento, Alexander Mordvintsev, Andrey Zhmoginov, and Max Vladymyrov. Transformers learn in-context by gradient descent. In *International Conference on Machine Learning*, pp. 35151–35174. PMLR, 2023.

Lei Wang, Yu Cheng, Yining Shi, Zhengju Tang, Zhiwen Mo, Wenhao Xie, Lingxiao Ma, Yuqing Xia, Jilong Xue, Fan Yang, and Zhi Yang. Tilelang: A composable tiled programming model for ai systems, 2025. URL https://arxiv.org/abs/2504.17577.

Jiin Woo, Shaowei Zhu, Allen Nie, Zhen Jia, Yida Wang, and Youngsuk Park. Tritonrl: Training llms to think and code triton without cheating, 2025. URL https://arxiv.org/abs/2510.17891.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents, 2025. URL https://arxiv.org/abs/2504.21798.

Songlin Yang and Yu Zhang. Fla: A triton-based library for hardware-efficient implementations of linear attention mechanism, January 2024a. URL https://github.com/fla-org/flash-linear-attention.

Songlin Yang and Yu Zhang. Fla: A triton-based library for hardware-efficient implementations of linear attention mechanism, January 2024b. URL https://github.com/fla-org/flash-linear-attention.

Mert Yuksekgonul, Daniel Koceja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, and Yu Sun. Learning to discover at test time. *arXiv preprint arXiv:2601.16175*, 2026. URL https://arxiv.org/abs/2601.16175.

Alex L. Zhang, Tim Kraska, and Omar Khattab. Recursive language models, 2025a. URL https://arxiv.org/abs/2512.24601.

Genghan Zhang, Weixin Liang, Olivia Hsu, and Kunle Olukotun. Adaptive self-improvement llm agentic system for ml library development. *arXiv preprint arXiv:2502.02534*, 2025b.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin godel machine: Open-ended evolution of self-improving agents, 2025c. URL https://arxiv.org/abs/2505.22954.

A Related Work and Limitations

A.1 AI for GPU Kernel Generation

GPU kernel optimization is central to deep learning performance, yet it is challenging for experts to write, motivating recent interest in AI-assisted kernel generation Ouyang et al. (2025a); Li et al. (2025). Prior approaches have shown promise on more established framework such as CUDA and Trition, with evolutionary search Novikov et al. (2025); Liao et al. (2025), reinforcement learning Baronio et al. (2025); Woo et al. (2025), and test-time training Yuksekgonul et al. (2026). In particular, they require strong enough prior in the target language and face significant challenges with data scarcity in newer DSLs as shown in our evaluation in 3.

As GPU kernel data is rare, synthetic data generation has been explored, notably through leveraging compiler systems as Paliskara & Saroufim (2025); Fisches et al. (2025) generate (PyTorch, Triton) pairs via Torch Inductor Ansel et al. (2024); however such transpilers only exist for limited amount of DSLs. Our work differs by focusing on DSL kernel generation from minimal examples and leveraging the model's inherent in-context learning capability, making it applicable to more DSLs.

A.2 Bootstrapping Capability in Data-Scarce Domains

Our approach addresses capability bootstrapping in low-resource settings through two mechanisms: in-context learning and synthetic data generation leveraging verifiers.

In-context learning from self-generated examples. Prior work has established that the choice of in-context examples significantly impacts model performance Liu et al. (2021); Levy et al. (2022); Qin et al. (2023), with theoretical characterizations of in-context learning Akyürek et al. (2022); Von Oswald et al. (2023). This is particularly valuable in low-resource domains such as rare languages Tanzer et al. (2024); Team (2024). Recent work has explored using self-generated examples to build growing libraries: Sarukkai et al. (2025) show self-generated examples improve LLM agent performance, and Zhang et al. (2025b) leverage in-context learning from an expanding library of functions for ML library development. We apply these principles to test-time kernel generation, where each attempt at any problem can produce verified solutions that serve as examples for future attempts at different problems, enabling cross-task knowledge transfer in data-scarce DSL settings.

Synthetic data leveraging verifiers Solutions generated through test-time problem-solving can also serve as training data for domain specialization. Recent work has shown synthetic data improves model capabilities in various domains, including repository-level bug fixing Yang et al. (2025), and self-generated examples have been used for fine-tuning in agent tasks Sarukkai et al. (2025). In Section 5, We demonstrate this pathway is particularly effective for emerging DSLs where human-written training data is scarce, as DSL-Monkeys enables generation via in-context learning and execution-based verifier as way to ensure data quality.

A.3 Limitations

We note here that our system prioritizes simplicity over specialized method engineering. For instance, we intentionally omit building in profiler feedback to iterative refinement, which could improve kernel quality but adds more complexity. Also, while our recipe (self-generated examples + verification) should generalize to other DSLs, it requires a reliable execution-based verifier and a task distribution where solved examples serve as useful demonstrations. Finally, our approach requires API credit compute (approx. $40 USD per run with Gemini-3-Flash / $100 with GPT-5-Mini with the settings shown in Figure 1) from iterative generation and evaluation, which we consider justified for GPU kernels where small gains save substantial developer time and deployment costs.

B GPU DSL FUNDAMENTALS

Modern GPUs have a hierarchical programming model. On NVIDIA GPUs, a *thread* is the fundamental unit of execution, 32 threads form a *warp*, and up to 32 warps (1024 threads) form a *block*. Blocks execute on Streaming Multiprocessors (SMs) with dedicated computational and memory resources. Recent DSLs for GPU kernel development differ in which level of this hierarchy they expose, their host language (Python or C++), and the amount of publicly available code.

We focus on recently-introduced DSLs with minimal public code representation. TileLang Wang et al. (2025) is a Python-based DSL that operates at higher abstraction and relies on compiler optimizations, enabling DeepSeek's Sparse Attention DeepSeek-AI (2025). ThunderKittens Spector et al. (2024b) is a C++ embedded DSL with opinionated primitives operating over 16×16 tiles, enabling practitioners to author high-performance kernels Sul (2025). Both were released within the last year. Table 1 characterizes these DSLs by their syntax, programming model, and volume of publicly available documentation and code samples. Each provides minimal documentation with 3-8 code examples, which we use as seed examples for our experiments.

C COUNTING TOKENS

We determine the number of high-quality, publicly-available tokens for each DSL by running the Qwen (Team, 2025) tokenizer through various sources of data:

- **TileLang.** We scrape the TileLang documentation (TileLang Developers, 2025) and paper (Wang et al., 2025).
- **ThunderKittens.** We scrape the TK Github repository (HazyResearch, 2024), paper (Spector et al., 2024b), blog (Spector et al., 2024a), and a publicly available onboarding document linked from the ThunkerKittens repo.
- **CuTe DSL.** We aggregate pages from Nvidia's CuTe DSL documentation (NVIDIA, 2025), Github repositories with CuTe DSL kernels (Dao-AILab, 2024b), and various online blog posts (kiui.moe, 2025; Barber, 2025; Veitner, 2024; Colfax International Research, 2025; Choy, 2025).
- **Triton TLX.** We scrape the Triton TLX Github repository (Experimental, 2025).
- **Triton.** We count the number of tokens from GPUMODE's Triton KernelBook (Paliskara & Saroufim, 2025).
- **CUDA.** We analyze the *the-stack-v2-dedup* dataset on HuggingFace (Lozhkov et al., 2024) and count the total number of CUDA bytes. Based on previous byte to token ratios, we divide the number of bytes by 4 to arrive at our token count.

D EVALUATION SETUP

D.1 KERNELBENCH TASK FORMAT

We evaluate on KernelBench tasks Ouyang et al. (2025a) from the most recent v0.2 version. KernelBench is a standardized benchmark where each task consists of: (1) a PyTorch reference implementation that defines the operator's functionality, (2) input-generation routines that specify tensor dimensions and batch sizes, and (3) correctness and performance testing infrastructure. The benchmark is organized into four difficulty levels, from basic single-kernel operators (Level 1) to complex fused patterns (Level 2) and full model architectures (Levels 3-4). To ensure reliable benchmarking, the v0.2 version scales tensor dimensions and batch sizes for Level 1–2 tasks so each test runs in roughly 1–15ms on an H100, avoiding cases where kernel launch overhead dominates the timing.

D.2 EVALUATION SETTING

We run evaluation on NVIDIA H100 GPUs on Modal. The evaluation container uses CUDA 12.8.0 and PyTorch 2.9.0. For each task, we compile and execute the generated kernel and require agreement with the PyTorch reference across 5 correctness trials, using the benchmark's dtype-specific tolerances. We measure runtime using Triton's `do_bench` Tillet et al. (2019) with a warmup budget of 25 ms and a repetition budget of 100 ms.

D.3 EVALUATION RESULTS

Table 3: Model Performance Across Different Tasks (N=5 for both effort levels). Note: Only genuine successes are counted; reward hacking attempts are excluded.

Task	**Model**	**Level 1**			**Level 2**		
		Acc.	**Cov.**	**Pass@1**	**Acc.**	**Cov.**	**Pass@1**
TileLang	Gemini-3-flash-preview	38%	58%	35.6%	2%	7%	2%
	GPT-5-mini	6%	18%	4%	0.4%	2%	0%
	Qwen3-30B-A3B	9%	13%	9.2%	0%	0%	0%
ThunkerKittens	Gemini-3-flash-preview	0.8%	3%	1%	0%	0%	0%
	GPT-5-mini	1.4%	5%	2%	0%	0%	0%
	Qwen3-30B-A3B	0.6%	2%	2%	0%	0%	0%
CuTe-DSL	Gemini-3-flash-preview	0.4%	2%	0%	0%	0%	0%
	GPT-5-mini	0.2%	1%	0%	0%	0%	0%
	Qwen3-30B-A3B	0%	0%	0%	0%	0%	0%
TLX	Gemini-3-flash-preview	6%	16%	8%	0.2%	1%	0%
	GPT-5-mini	1.4%	7%	2%	0%	0%	0%
	Qwen3-30B-A3B	0.6%	3%	1%	0%	0%	0%

E CONTEXT CONSTRUCTION

Models without DSL-specific information perform poorly, so we provide strong baseline context for all methods. For each DSL, we provide: (1) a DSL-specific guide synthesized from documentation using recursive language models Zhang et al. (2025a), and (2) 3 representative in-context examples (see Table 4) drawn from each DSL's documentation. These 3 examples serve as our seed set. Following prior work Zhang et al. (2025c); Hu et al. (2025); Ouyang et al. (2025b), we use a 2-stage generation setup where the model first generates an implementation plan, then produces code conditioned on the plan.

E.1 DSL LANGUAGE GUIDE

Because frontier models are unlikely to have been trained on newly released DSLs, we supply them with the relevant language knowledge at inference time. To compress the large volume of scraped documentation into a single prompt-friendly reference, we synthesize domain-specific language guides using recursive language models (RLMs) to handle the long context Zhang et al. (2025a). Specifically, we run an RLM instance of GPT-5.1 with max recursive depth 7. The prompt to the RLM is shown below.

DSL Language Guide RLM Prompt

```
You are given extensive documentation, papers, blogposts, and code
for a GPU **<DSL>**.  Your task is to synthesize this material into a
structured guide that teaches a performance engineer how to *think*
in this DSL; not just use it, but learn its abstraction so they can
translate any PyTorch operation into an efficient kernel plan, which
targets the DSL features.

Do not use bullet points; explain in clear natural language.  The
goal is to convey the DSL's philosophy as accurately as possible, not
necessarily it's syntax.

Start the prompt with:  "You are an expert GPU kernel engineer
specializing in translating PyTorch operations into <DSL>
implementation plans.  Your task is to analyze PyTorch code and
produce detailed, structured natural language plans that describe how
to implement the operations efficiently in <DSL>.  You do not write
<DSL> code.  You produce implementation plans that a <DSL> programmer
could follow to produce the kernel that leverages the <DSL> feature.
Your plans must demonstrate a deep understanding of <DSL>
abstractions, philosophy, and the hardware realities of modern GPUs."

The rest of your prompt should address the following:

### 1.  What is <DSL> and what problem does it solve?

What gap does it fill between PyTorch and raw CUDA? What is the main
abstraction that the <DSL> provides?

### 2.  What is the fundamental unit of thought?  What does it mean
to write <DSL>?

Every DSL has a primary abstraction that defines how programmers
should decompose problems.  Identify it precisely.

### 3.  What does the compiler handle vs.  what must the programmer
specify?

State and explain what the <DSL> handles automatically, exposes for
optional control, or requires explicit specification.  The goal
should be that a programmer should know what to specify and what to
leave alone.

### 4.  How does <DSL> model the memory hierarchy?

GPU DSLs typically address global -> shared -> register data movement.
Describe:  - What primitives or types represent each memory tier
```

```
(actually use the <DSL> documentation to find the names of the
primitives) - The pattern for staging data through the hierarchy -
How reuse is expressed - Any DSL-specific features (swizzling, TMA
integration, typed tiles, etc.)

### 5.  What are the core compute primitives?

List the key operations the DSL provides and what hardware they map
to:  - Matrix multiply / MMA primitives (and tensor core
requirements) - Copy/load primitives (and async capabilities) -
Elementwise / parallel iteration constructs - Reduction primitives -
Synchronization primitives

For each, note any constraints (tile size alignment, layout
requirements, precision).

### 6.  How should programmers think when translating PyTorch ops?

Describe the mental model as a series of steps or questions.  Tailor
these steps to the DSL's abstraction.  If the DSL handles something
automatically, say so.

## Output Format

Structure your output as follows:

# <DSL>:  Abstraction and Programming Model

## Core Philosophy [The fundamental unit of thought]

## Thinking in <DSL> [Memory hierarchy and how to use it] [Compute
primitives and their constraints]

## Recognizing Patterns [Pattern-by-pattern breakdown with
DSL-idiomatic approaches]

Be thorough and draw explicitly from the material provided.
```

E.2 In-Context Examples

We select three representative kernel examples per DSL that cover common programming patterns, drawn from each framework's official repository. These examples serve as the seed set for bootstrapping.

DSL	Seed Examples
TileLang	FlashAttention; GEMM; Conv2D
TLX	FlashAttention; GEMM; LayerNorm
ThunderKittens	FlashAttention; GEMM; LayerNorm
CuTe-DSL	Multi-Head Latent Attention; GEMM; Softmax

Table 4: Seed in-context examples for each DSL.

E.3 DSL-Monkeys Planning Stage Context Construction

In the first stage, the model analyzes the PyTorch reference code and creates a detailed implementation plan. The prompt provides the model with: (1) an overview of the target DSL, (2) three in-context examples showing PyTorch code paired with corresponding implementation plans, and (3) the new problem to solve. The model then generates a natural-language plan that describes how to implement the operation in the DSL, including details about memory layout, tiling strategies, optimization techniques, and computation order.

Planning Prompt

You are an expert GPU kernel engineer specializing in turning reference PyTorch programs into DSL programs with custom kernels. Your task is to analyze the given PyTorch code and produce detailed, structured natural language plans that describe how to implement the operations correctly and efficiently in the DSL.

[[## dsl_overview ##]]

[DSL Language Guide]

[[## context ##]]

[ICL Example 1]

[Problem Code]:

```
import torch
import torch.nn as nn

class Model(nn.Module):
    """
    Simple model that performs a Swish activation.
    """
    def __init__(self):
        super(Model, self).__init__()

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        Applies Swish activation to the input tensor.

        Args:
            x (torch.Tensor): Input tensor of any shape.

        Returns:
            torch.Tensor: Output tensor with Swish applied.
        """
        return x * torch.sigmoid(x)

batch_size = 4096
dim = 393216
...
```

[Implementation Plan]:

The goal is to implement a single TileLang kernel that computes the Swish activation, y = x · sigmoid(x), for a tensor x of shape (B, D) in bf16, returning y in the same shape and dtype. The kernel fuses load, fp32 conversion, sigmoid (via exp and reciprocal), multiply, and store into a single tiled kernel, using cooperative global-to-shared copies (T.copy) inside T.Pipelined to enable cp.async/TMA lowering where available and double-buffering to hide global memory latency...

...

[ICL Example 2]

[Problem Code]:

...

[Implementation Plan]:

...

[ICL Example 3]

[Problem Code]:

```
...
[Implementation Plan]:
...
[[ ## task_to_be_optimized ## ]]

import torch
import torch.nn as nn

class Model(nn.Module):
    """
    Model that performs a convolution, applies HardSwish,
    and then ReLU.
    """
    def __init__(self, in_channels, out_channels, kernel_size):
        super(Model, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size)

    def forward(self, x):
        """
        Args:
            x (torch.Tensor): Input tensor of shape
            (batch_size, in_channels, height, width).

        Returns:
            torch.Tensor: Output tensor of shape
            (batch_size, out_channels, height, width).
        """
        x = self.conv(x)
        x = torch.nn.functional.hardswish(x)
        x = torch.relu(x)
        return x

batch_size = 128
in_channels = 8
out_channels = 64
height, width = 128, 128
kernel_size = 3

...

[[ ## precision ## ]]
bf16
```

E.4 DSL-MONKEYS IMPLEMENTATION STAGE CONTEXT CONSTRUCTION

In the second stage, the model translates the implementation plan into actual DSL code. The prompt includes: (1) the DSL overview, (2) three in-context examples showing PyTorch code, implementation plans, and their corresponding DSL solutions, (3) the new problem's PyTorch code, and (4) the implementation plan generated in stage one. Using both the plan and the PyTorch reference, the model writes complete DSL kernel code.

Coding Prompt

```
You are an expert GPU kernel engineer and Domain-Specific Language
(DSL) programmer.  Your task is to translate the provided
natural-language kernel design plan into complete, valid DSL code.
Follow the plan exactly, preserving its intended computation, shapes,
and performance-critical decisions (tiling, memory movement,
pipelining, synchronization, fusion, etc).
```

[[## dsl_overview ##]]

[DSL Language Guide]

[[## context ##]]

[ICL Example 1]

[Problem Code]:

```
import torch
import torch.nn as nn

class Model(nn.Module):
    """
    Simple model that performs a Swish activation.
    """
    def __init__(self):
        super(Model, self).__init__()

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        Applies Swish activation to the input tensor.

        Args:
            x (torch.Tensor): Input tensor of any shape.

        Returns:
            torch.Tensor: Output tensor with Swish applied.
        """
        return x * torch.sigmoid(x)

batch_size = 4096
dim = 393216
...
```

[Implementation Plan]:

```
The goal is to implement a single TileLang kernel that computes the
Swish activation, y = x · sigmoid(x), for a tensor x of shape (B, D)
in bf16, returning y in the same shape and dtype.  The kernel fuses
load, fp32 conversion, sigmoid (via exp and reciprocal), multiply,
and store into a single tiled kernel, using cooperative
global-to-shared copies (T.copy) inside T.Pipelined to enable
cp.async/TMA lowering where available and double-buffering to hide
global memory latency...

...
```

[Solution Code]:

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

def _build_swish_kernel(N, block_size=1024, dtype="bfloat16"):
    @T.prim_func
    def swish_kernel(x: T.Tensor((N,), dtype),
                     y: T.Tensor((N,), dtype)):
        grid_size = T.ceildiv(N, block_size)
        with T.Kernel(grid_size, threads=128) as bx:
            x_frag = T.alloc_fragment((block_size,), dtype)
            for i in T.Parallel(block_size):
                idx = bx * block_size + i
                if idx < N:
                    # Load
```

```
                    val = x[idx]
                    # Compute in float32 for precision
                    v_f32 = T.cast(val, "float32")
                    sigmoid_v = 1.0 / (1.0 + T.exp(-v_f32))
                    res_f32 = v_f32 * sigmoid_v
                    # Store back
                    y[idx] = T.cast(res_f32, dtype)

    return tilelang.compile(swish_kernel, out_idx=[1], target="cuda")
...
[ICL Example 2]
[Problem Code]:
...
[Implementation Plan]:
...
[Solution Code]:
...
[ICL Example 3]
[Problem Code]:
...
[Implementation Plan]:
...
[Solution Code]:
...
[[ ## task_to_be_optimized ## ]]

import torch
import torch.nn as nn

class Model(nn.Module):
    """
    Model that performs a convolution, applies HardSwish,
    and then ReLU.
    """
    def __init__(self, in_channels, out_channels, kernel_size):
        super(Model, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size)

    def forward(self, x):
        """
        Args:
            x (torch.Tensor): Input tensor of shape
            (batch_size, in_channels, height, width).

        Returns:
            torch.Tensor: Output tensor of shape
            (batch_size, out_channels, height, width).
        """
        x = self.conv(x)
        x = torch.nn.functional.hardswish(x)
        x = torch.relu(x)
        return x

batch_size = 128
in_channels = 8
out_channels = 64
```

```
height, width = 128, 128
kernel_size = 3

...
```

[[## implementation_plan ##]]

```
The operation is a composition of a 2D convolution followed by two
element-wise activations:  HardSwish and ReLU. For high performance,
we should fuse these into a single kernel to avoid multiple global
memory round-trips for the activations.  The 2D convolution is
mathematically mapped to a GEMM where M = batch * H_out * W_out, N =
out_channels, and K = in_channels * kernel_size * kernel_size.  Since
the output height and width for a standard convolution with stride=1,
padding=0, kernel=3 are (H-2) and (W-2), we will calculate these and
map our thread blocks over this M-dimension.  HardSwish is defined as
x * clip(x + 3, 0, 6) / 6, and ReLU is max(0, x).  Since
ReLU(HardSwish(x)) is the final op, we can optimize the calculation
in the register fragment before writing back...

...
```

[[## precision ##]]

```
bf16
```

F TEST-TIME BASELINE SETUP

Our baselines evaluate other test-time compute frameworks for GPU kernel generation. We select these baselines because they have proven effective on kernel generation tasks Sharma (2025b). Importantly, these methods generate each kernel independently without leveraging attempts from related problems, which demonstrates the value of DSL-Monkeys's approach to learning from related kernels. Each baseline uses the same prompting setup and test-time budget as DSL-Monkeys (detailed in E). We maintain three seed examples as in-context demonstrations for every iteration and run each method for 25 attempts per problem.

F.1 ITERATIVE REFINEMENT

We adopt the iterative refinement framework used in KernelBench Ouyang et al. (2025a) from their repository, which leverages the immediately preceding kernel attempt and its execution feedback to guide the next iteration. This is also similar to other works like Shinn et al. (2023); Ehrlich et al. (2025).

We adapt the iterative refinement setup to match the two-stage inference pipeline of DSL Monkeys, described in E. We append the immediately preceeding iteration's code and execution feedback to both the planning stage and the code generation stage.

F.2 EVOLUTIONARY SEARCH

OpenEvolve Sharma (2025a) is a commonly used open-source variant of AlphaEvolve Novikov et al. (2025), which has demonstrated previous success in writing and optimizing GPU kernels Sharma (2025b) and other systems engineering tasks Cheng et al. (2025). OpenEvolve employs an evolutionary search procedure that starts from one or more seed implementations and uses an island-based population structure to promote diversity. Candidate solutions are stored in a shared archive that is periodically pruned using MAP-Elites Mouret & Clune (2015), enabling a balance between exploration of diverse program behaviors and exploitation of high-performing solutions. At each iteration, OpenEvolve samples from this archive to generate new variants, gradually improving performance across different regions of the search space.

We adapt the OpenEvolve pipeline to fit our setting. Because OpenEvolve has its own prompt sampling setup, we do not use our two-stage planning setup, so it generates the kernel code in a single stage. Our OpenEvolve hyperparameters match the default recommended configuration, with the exception of the number of the previous attempts we include in our context. We cap this at 3 to match our existing test-time budget. We pass all DSL information and in-context examples to the system prompt, and maintain existing attempts' execution feedback within the database so that the model can improve.

G COMBINING DSL-MONKEYS AND OPENEVOLVE

We combine DSL-Monkeys with OpenEvolve to leverage both cross-task retrieval and same-task execution feedback. The combined approach proceeds in three phases. First, we run OpenEvolve for 5 iterations using its standard protocol with 3 fixed seed examples. Second, we replace the 3 examples in the system prompt by retrieving similar kernels from the archive of previously-implemented solutions using DSL-Monkeys's retrieval mechanism, then run OpenEvolve for an additional 10 iterations with these retrieved examples. Finally, we re-retrieve examples from the updated archive (which now includes any kernels solved in the previous phase) and run OpenEvolve for 10 more iterations. This staged approach allows OpenEvolve's evolutionary search to benefit from relevant cross-task examples while still applying execution feedback to refine individual implementations. The results in Table 2 show this combination achieves the highest performance, reaching 155/200 on TileLang with both Gemini-3-Flash and GPT-5-Mini, demonstrating that cross-task retrieval and same-task execution feedback are complementary strategies.

H REWARD HACKING PREVENTION

We outline our system for preventing reward hacking, a common pitfall of LLM kernel generation frameworks where the model cheats on the evaluation suite to inflate performance. We use two types of checkers: regex and LLM-as-a-judge. The regex checker runs in-loop, ensuring that each generated kernel includes certain DSL-specific features and excludes common reward hacking patterns. Since we are generating DSL code, we consider it reward hacking if a kernel fails to leverage the DSL's core features—thus, we define mandatory patterns that valid kernels must use (listed in Table 5). Additionally, the regex checker detects common reward hacking patterns outlined in Table 6. For example, we disallow the use of any PyTorch computation ops (i.e., no torch.nn, F.functional, etc.) similar to Baronio et al. (2025). Finally, we use an LLM-as-a-judge system to validate the generated kernels as a secondary check.

DSL Backend	Required Features
Triton	• Decorator: `@triton.jit` or `@triton.autotune` • Operations: `tl.*` operations required • Memory operations: `tl.load()` or `tl.store()` • Kernel patterns: `tl.program_id()`, `tl.num_programs()`, `tl.constexpr`, `tl.arange()`, or `tl.cdiv()`
TLX	• Async primitives: `tlx.async` patterns (e.g., `tlx.async_tasks`, `tlx.async_load`, `tlx.async_dot`) • Restriction: MUST NOT contain `@triton.autotune`
ThunderKittens	• Warp/warpgroup patterns: `kittens::warp`, `kittens::warpgroup`, `::warpgroup::`, `::warp::`, `warpgroup::`, or `warp::` • Tile declarations: `st_*<...>` or `rt_*<...>` patterns (e.g., `st_bf`, `rt_fl`)
CuTe	• Namespace: `cute::`, `cutlass::`, or `from cutlass` • Must contain one of: ◦ Tensor operations: `make_tensor()`, `make_layout()`, `make_shape()`, or `make_stride()` ◦ Copy operations: `copy()`, `copy_if()`, `cute::copy`, or `Copy_Atom` ◦ GEMM patterns: `cutlass::gemm`, `cutlass::epilogue`, `Gemm*<`, `GemmConfiguration`, or `ThreadblockSwizzle` ◦ Kernel patterns: `cutlass::arch`, `cutlass::layout`, `RowMajor/ColumnMajor`, or `TensorRef<`
TileLang	• Decorator: `@T.prim_func` • OR kernel: `T.Kernel`

Table 5: DSL Backend Requirements. Each backend must satisfy all listed requirements to be considered a valid kernel implementation.

We also use an LLM-as-a-Judge as a secondary layer of validation to complement our static checkers.

Reward Hack LM Judge Prompt

```
Analyze if this kernel correctly implements the reference PyTorch
code with the backend DSL. They are all marked as correct by our
evaluation system but we might have missed some reward hacking
patterns (they could be tricky)!

{reference_section}

Do note the kernel just have to work well on this particular set of
input shapes and initialization configs.  It does not have to be a
general solution (aka it does not have to be able to handle any input
shapes and initialization configs as different shapes require
different optimizations).

## Generated Kernel in DSL ({backend}):  '''python kernel_code '''
```

Category	Pattern	Description
Bypass Checks	Try-Except Fallback (Strict)	Models wrap incomplete CUDA/kernel code in exception handlers that fall back to PyTorch when custom code fails. Allows passing tests without implementing the kernel.
	Pass Statement (Strict)	Models inherit from reference class and use `pass` to do nothing, effectively calling the parent implementation.
	PyTorch NN Wrapping (Strict)	Uses PyTorch compute layers (`nn.Linear`, `nn.Conv2d`, `nn.ReLU`, etc.). Allows containers (`nn.Module`, `nn.Parameter`, `nn.init`) needed for model structure.
	Torch Computation Ops (Strict)	Uses high-level PyTorch operations (`torch.matmul`, `torch.softmax`, `F.relu`, `F.conv2d`, etc.) instead of custom kernel implementations. Includes matrix ops, convolutions, pooling, activations, normalization, reductions, and loss functions.
Timing	Stream Injection (Warning)	Uses CUDA streams (`torch.cuda.Stream()`, `with torch.cuda.stream(...)`, `.wait_stream()`, `.record_stream()`) to defer computation or manipulate execution order, potentially affecting benchmark timing measurements.
	Thread Injection (Strict)	Uses threading (`threading.Thread()`) or multiprocessing (`multiprocessing.Process`, `ThreadPoolExecutor`, `ProcessPoolExecutor`) to parallelize work or manipulate execution order.
	Lazy Evaluation (Strict)	Creates fake/lazy tensors using `_make_subclass`, custom tensor subclasses (`class X(torch.Tensor)`), or direct tensor construction (`torch.Tensor.__new__`) that don't actually compute anything, passing correctness checks without real implementation.
	Timing Event Patch (Strict)	Monkey-patches timing functions (`torch.cuda.Event.record = ...`, `torch.cuda.synchronize = ...`, `time.perf_counter = ...`) to fake benchmark results, making slow code appear fast.
	Precision Downgrade (Warning)	Downgrades precision internally (FP32 $\rightarrow$ FP16) to make kernels faster while still returning FP32 output. Detects patterns like `.half()`, `.to(torch.float16)`, `__float2half()`, `tl.astype(..., tl.float16)`, `NumericConverter<half_t, float>`, etc. Can be a valid optimization but often a reward hack.

Table 6: Reward Hacking Patterns

```
CRITICAL CHECKS: 1. **Op Coverage**: Are ALL operations from the
reference (matmul, softmax, etc.) implemented in the kernel's
computation path? Not just present as dead code, but actually
executed during the active code path with the correct input and
output. 2. **DSL Usage**: Is backend actually being USED for
computation? The DSL primitives must be CALLED in the forward path,
not just imported/defined but never invoked. 3. **No Torch
Fallback**: Does it avoid using torch.* or F.* for core computation?
(torch.Tensor for I/O is OK) We suppose to write kernels in DSL to do
the computation. 4. **No Empty Impl**: No pass statements, no
try-except that falls back to torch. Computation should be done with
the DSL. 5. **Other forms of potential reward hacking**: Are there
other signs of reward hacking that you think that might be there?
They could be subtle so be be careful and check them.
```

```
Computation should be done in the given precision.

Respond with ONLY JSON: {{"score": 0.85, "summary": "one line",
"reasons": ["reason1", "reason2"]}}

Scoring: - 1.0: All ref ops implemented with backend, DSL is
actually called - 0.7-0.9: Minor issues, most ops covered - 0.3-0.6:
Some ops missing OR DSL defined but not called - 0.0-0.2: Clear hack
and malicious attempts - empty impl, torch fallback, or DSL never
executed
```

I MORE ON DSL-MONKEYS

I.1 ABLATIONS ON RETRIEVAL METHOD

We ablate the retrieval method of DSL-Monkeys and compare three strategies for selecting in-context examples from the archive: **similarity** (cosine similarity over problem embeddings), **random** (uniform sampling), and **alternating** (switching between similarity and random each iteration). Alternating retrieval balances exploitation of relevant demonstrations with exploration of diverse patterns, and achieves the highest solve rate. We use the alternating strategy for all main experiments.

Table 7: Retrieval strategy ablation on TileLang (Gemini-3-Flash, 3 seeds, 25 max attempts per problem).

Retrieval Strategy	**Correct Kernels in Level 1+2**
Random	115
Similarity	133
Alternating	**140**

I.2 PERFORMANCE ANALYSIS ON KERNELBENCH LEVELS 1 AND 2

We evaluate the performance characteristics of our DSL-Monkeys approach using the fast_1 metric from KernelBench. The fast_1 score measures the speedup of generated kernels relative to a baseline implementation, where a score of 1.0 indicates performance parity with the reference kernel.

Figure 5 shows the progression of fast_1 scores across 25 iterations of the DSL-Monkeys pipeline using GPT-5-Mini with 3 seeds and 3 in-context examples per generation. We note that DSL-Monkeys prioritizes correctness over optimization: once a kernel passes correctness, we do not perform additional performance tuning and leave that as future work.

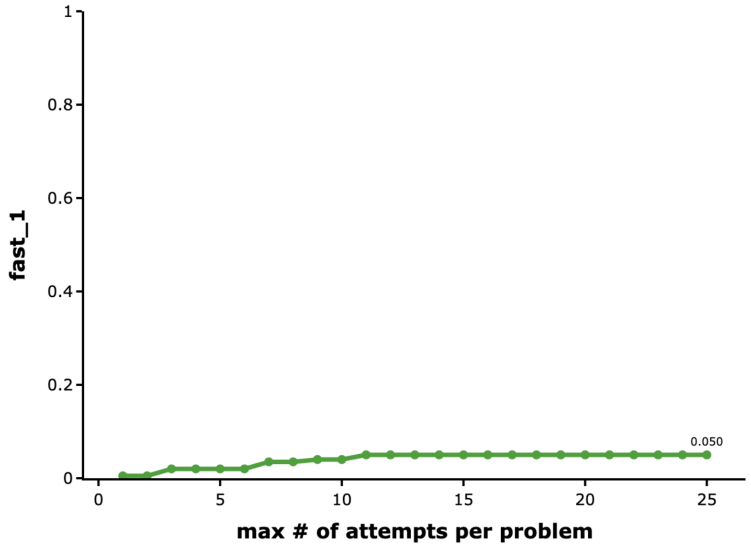

Figure 5: Evolution of fast_1 scores per iteration during the DSL-Monkeys run shown in Figure 1 (GPT-5-Mini, 3 seeds, 3 in-context examples, 25 iterations).

J TRAINING DETAILS

J.1 SCALING TASK CURATION

We use the KernelBook (Paliskara & Saroufim, 2025) dataset and format the problems to match KernelBench's format. To decontaminate, we follow (Guha et al., 2025) and apply fuzzy string matching (similarity threshold of 75.0) and n-gram matching (n=13): if 13 consecutive tokens from a training sample match any 13 consecutive tokens from any KernelBench problem, the sample is excluded. After decontamination, we end up with 10,082 PyTorch programs.

J.2 DATA GENERATION

We run DSL-Monkeys on decontaminated KernelBook for 5 iterations. Results are shown in Table 8.

	Gemini 3 Flash + TileLang		**Qwen3-30B-A3B + TileLang**	
Iteration	Correct	Compiled	Correct	Compiled
1	2,476	9,342	414	6,736
2	3,510	9,809	759	7,489
3	4,104	9,843	1,001	7,704
4	4,454	9,854	1,140	7,793
5	4,731	9,856	1,228	7,976

Table 8: **DSL-Monkeys enables synthetic data generation** from different models (teacher from Gemini and self-generated data from Qwen). We leverage these trajectories as future seed examples or SFT training data.

J.3 TRAINING CONFIGURATION

After having generated valid trajectories, we can now fine-tune on them. We run supervised fine-tuning with LoRA Training (Hu et al., 2021) through Tinker (Machines, 2025). To make trajectories fit into context, we split each successful trajectory into a planning and code generation phase, and fine-tune on them independently.

We chose LoRA because it has been demonstrated to preserve reasoning (Biderman et al., 2024). Our learning rate is computed according to the recommended value with (Schulman & Lab, 2025).

We tune the remaining main hyparameters (batch, rank, and number of epochs) by following the method in (Yang et al., 2025; Guha et al., 2025). Specifically, we gridsearch these values on a logarithmic scale for an SFT setup with 5000 training samples (2,500 successful kernels). This ensures we do not bias either end of the data spectrum.

We provide our training hyperparameter for all SFT training runs:

```
Constant learning rate of 4.99e-4
Max length 24576
Batch Size 8
Rank 64
Number of Epochs 4
```

J.4 SFT DATA SCALING

We design experiments to understand the effect of data quality and data scaling in the distillation setting. We evaluate all fine-tuned models on KernelBench Level 1+2 with TileLang, using DSL-Monkeys with 25 iterations.

Data Quality. We compare three sources of training data, each with 800 solved problems (1600 training samples). Gemini DSL-Monkeys trajectories are most effective (34 → 104), followed by Gemini OpenEvolve (34 → 81). The gap suggests that DSL-Monkeys trajectories, which explicitly demonstrate cross-task retrieval and transfer, teach the model a more useful skill than standalone solutions. Training on Qwen's own DSL-Monkeys-generated trajectories still yields gains (34 → 43), showing that self-improvement is possible without a stronger teacher.

Model	Source of Data	Problem Solved	Training Samples	KernelBench Problems Solved
Gemini-3-Flash (Teacher)		N/A	N/A	**140**
Qwen3-30B-A3B (Base)		0	0	34
Qwen SFT	Gemini OpenEvolve	800	1600	81
Qwen SFT	Gemini DSL-Monkeys	800	1600	**104**
Qwen SFT	Qwen DSL-Monkeys	800	1600	43
Qwen SFT	Gemini DSL-Monkeys	800	1600	104
Qwen SFT	Gemini DSL-Monkeys	1600	3200	121
Qwen SFT	Gemini DSL-Monkeys	3200	6400	**122**

Table 9: Effect of data source and scale on KernelBench test-time performance (25 test-time iterations using DSL-Monkeys).

Data Scaling in Distillation Setting. Scaling Gemini DSL-Monkeys data from 800 to 3200 solved problems steadily improves correctness, reaching 122 and approaching the teacher model's 140. While the number of correct kernels plateaus between 1600 and 3200 problems (121 → 122), the number of kernels that also match or exceed PyTorch Eager or Compile jumps from 7 to 24 (5.8% → 19.7%). This suggests that additional data helps the model make better use of DSL-specific features, producing not just correct but also more performant kernels that come with better usage the DSL.

K RETRIEVAL ANALYSIS: NOVEL EXAMPLES AND SUCCESS RATES

Table 10 shows the relationship between retrieval novelty and success on previously-failed tasks. When no examples change between attempts, the success rate is only 2.0%. As more examples are replaced with newly-solved kernels from the archive, success rates increase substantially: 3.4% with one changed example, 5.3% with two, and 20.8% when all three examples are replaced.

Examples Changed	Total Attempts	Successful	Success Rate
0	448	9	2.0%
1	706	24	3.4%
2	283	15	5.3%
3	125	26	20.8%

Table 10: Success rate on previously-failed tasks increases with novel retrieved examples.

Case studies. Examining specific cases reveals how novel examples enable solutions. Problem 35 (`Conv2d_Subtract_HardSwish_MaxPool_Mish`) succeeded after example `2_89` demonstrated incremental aggregation for MaxPool. Problem 6 (`Conv3d_Softmax_MaxPool_MaxPool`) succeeded after `2_96` showed proper nested loop structure for multi-dimensional pooling.

L Additional In-Context Analysis Details

L.1 Borrowing Analysis Prompt

We use the following prompt with Gemini-3-Flash to analyze code borrowing between in-context examples and generated kernels:

Listing 1: Prompt for analyzing code borrowing between generated kernels and in-context examples.

```
=== System ===
You are an expert kernel code reviewer. Given a target kernel and
    parent kernels (in-context examples) that were used to generate
    it, identify borrowed patterns. Return strict JSON with key
    'borrowed' as a list; each item must include parent_id, concept,
    parent_lines (list of [start,end]), target_lines (list of
    [start,end]), rationale. Prefer 3-8 findings, concise rationale.

=== User (JSON) ===
{"task": "Find borrowed code patterns from each parent example into
    the target kernel.", "target": {"problem_name": "...", "level":
    ..., "problem_id": ..., "code": ["0001: ..."]}, "parents":
    [{"id": "...", "problem_name": "...", "code": ["0001: ..."]}],
    "required_output": {"borrowed": [{"parent_id": "str", "concept":
    "short name of borrowed idea", "parent_lines": [[1, 5]],
    "target_lines": [[10, 14]], "rationale": "why these lines are
    borrowed/reused"}]}}
```

M CHARACTERIZATION OF UNSOLVED KERNELBENCH TASKS

The majority of kernels that remain unsolved after multiple evolution passes are Level 2 (L2) fusion benchmarks. Inspection of their reference implementations shows that these L2 unsolved problems tend to combine operations in long, highly synthetic chains—e.g., transposed convolution followed by LayerNorm or GroupNorm, then LogSumExp, then Mish or other activations—that do not correspond to common patterns in production models. At Level 1 (L1), four of the five unsolved problems are cumulative-sum variants (inclusive, exclusive, reverse, or masked scan). These remain unsolved largely because the model is not conditioned on the existence of TileLang's T.cumsum primitive; without that, the search space is effectively limited to hand-rolled sequential scans that either violate the DSL's parallel-loop rules or fail correctness. The fifth L1 unsolved problem is cross-entropy loss, which involves both reduction and irregular indexing. Thus, the unsolved set is largely made up of (i) L2 fusions that stress the compiler with nonstandard op sequences and conflicting layout/scheduling constraints, and (ii) L1 problems where the main bottleneck is awareness of a single primitive (T.cumsum) rather than fundamental expressiveness of the DSL.

N CASE STUDY

A core motivation of DSLs and abstractions is to easily allow developers to prototype kernels for novel attention algorithms: for instance, TileLang was used to develop DeepSeek's Sparse Attention DeepSeek-AI (2025). One important application concerns **linear attention**, which has a large, diverse set of variants Yang & Zhang (2024b).

N.1 EXPERIMENTAL DESIGN

To evaluate the efficacy of our approach on complex, real-world kernels, we compare its performance across two distinct problem formulations. We compare evolutionary search with our bootstrapping approach:

1. **Monolithic (Baseline)**: In this setting, we test whether `OpenEvolve` and `Repeated Sampling` can generate complete fused kernels from the FLA repository Yang & Zhang (2024b) in one go. This baseline shows how well standard test-time compute methods handle generating complex TileLang kernels (with optimizations like swizzling and pipelining) directly from PyTorch code.
2. **Decomposed (DSL-Monkeys Curriculum)**: In contrast to the monolithic approach, we use a setting where DSL-Monkeys decomposes the problem into smaller pieces: we split the PyTorch reference into simpler sub-problems and convert each into a separate KernelBench task. The model first solves these easier sub-kernels and stores them in its database. It can then reuse these building blocks when tackling similar problems and eventually combine them to generate the full kernel.

This decomposition is automated: we feed the PyTorch implementation to `Gemini-3-pro`, which uses a decomposition prompt to break it into a set of standalone KernelBench sub-problems. Applying this procedure to our suite of 12 linear attention variants yields a curated curriculum of 24, 39, or 104 constituent sub-problems, depending on the level of decomposition.

N.2 CORRECTNESS RESULTS

Testing DSL-Monkeys on complex linear attention kernels shows a large performance gap between repeated sampling and bootstrapping. We compare DSL-Monkeys against OpenEvolve and Repeated Sampling as baseline methods for solving difficult kernels.

Evolutionary search methods like OpenEvolve Sharma (2025a) have proven effective for generating state-of-the-art kernels Sharma (2025b); Novikov et al. (2025). However, Table 11 shows these methods fail to synthesize correct linear attention kernels in low-resource DSLs like TileLang. Repeated Sampling only succeeds on the Naive Parallel Based kernel, showing that the base model (Gemini 3 Pro) has some prior knowledge for this operator, but the approach doesn't generalize to other variants.

Instead of attempting monolithic synthesis, to utilize DSL-Monkeys we decompose the 12 linear attention variants into a set of foundational sub-problems. In this "Decomposed" setting, the system successfully utilized these sub-problems via RAG to synthesize the final fused kernels. The approach allowed us to solve up to 4/12 of the most challenging linear attention problems, a result that was impossible with prior methods. Specifically, we were able to solve the parallel BASED Arora et al. (2025), parallel ReBased Aksenov et al. (2024), parallel RetNet Sun et al. (2023), and chunkwise Linear Attention Yang & Zhang (2024b) kernels in the FLA repository.

We performed an ablation study on the granularity of our decomposition, varying the number of sub-problems from 24 to 104. Our findings indicate that a lower level of decomposition for these particular problems provides a robust enough curriculum that enables the model to solve complex parallel linear attention operators like Retention and ReBased.

These results prove that DSL-Monkeys is uniquely applicable to complicated, real-world systems tasks where the "complexity gap" between a reference implementation and an optimized kernel is too wide for standard synthesis techniques. By transforming the synthesis process into a more continuous journey through a curated curriculum of relevant sub-problems, we unlock the potential of test-time compute.

Method	Setting	Sub Tasks		FLA Tasks Solved				Others
		Total	Solved	Par. BASED	Par. ReBased	Par. RetNet	Chunk LinAttn.	
OpenEvolve	Monolithic	–	–					
Repeated Sampling	Monolithic	–	–	✓				
DSLMonkeys	Monolithic	–	–	✓	✓			
DSLMonkeys + 140 Seeds	Monolithic	–	–	✓		✓		
DSLMonkeys	Decomposed	24	12	✓	✓	✓	✓	
DSLMonkeys	Decomposed	39	23	✓	✓			
DSLMonkeys	Decomposed	104	80	✓	✓	✓		

Table 11: **Synthesis success rates across linear attention variants in TileLang.** We compare monolithic synthesis against curriculum-based decomposition. In the decomposed setting, each linear attention operator is sliced into 24 - 104 standalone KernelBench-style sub-tasks; we report the number of sub-tasks generated and solved. Columns under **FLA Tasks Solved** denote which fused FLA kernels were successfully synthesized. **Others** includes Chunked Based, Parallel NSA, Chunked RetNet, Recurrent Gated DeltaNet, Parallel DeltaNet, Parallel GLA, Recurrent GLA, and Recurrent GSA.

N.3 Performance Results

We integrated our most performant kernels into the Flash Linear Attention benchmark suite Yang & Zhang (2024b) to evaluate the efficacy of the DSL-Monkeys bootstrapping process. These experiments demonstrate that our curriculum-based bootstrapping method successfully navigates the low-resource TileLang DSL to produce highly efficient operators that consistently outperform optimized `torch.compile` references. Notably, the synthesized kernels were able to match the performance of expert, human-written Triton baselines provided in FLA, even outperforming hand-written baselines for architectures like Parallel ReBased at low sequence-lengths.

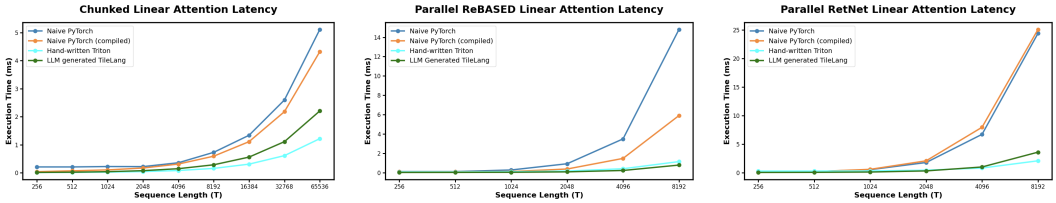

Figure 6: **DSL-Monkeysbootstrapped TileLang kernels demonstrate lower latency compared to `torch.compile` max-autotune baselines.** Latency benchmarks across three distinct architectures—`Chunked Linear Attention` (left), `Rebased Parallel` (center), and `Retention Parallel` (right)—show that kernels synthesized via our decomposition method consistently outperform `torch.compile` and approach the hand-written Triton baselines. Benchmarks were executed with a fixed batch size $B = 4$, head count $H = 8$, and head dimension $D = 64$. The above kernels were not successfully synthesized by OpenEvolve or Repeated Sampling.

N.4 Kernel Features

Across all three kernels, the model effectively leverages TileLang's memory hierarchy management through `T.alloc_shared` and `T.alloc_fragment`. By explicitly mapping data to shared memory and register tiles, the implementation achieves high data reuse and avoids the latency penalties associated with frequent global memory access.

The kernels further optimize throughput by utilizing:

- **Software Pipelining:** `T.Pipelined(num_stages)` enables asynchronous data movement, allowing the system to pre-fetch the next data tile into shared memory while simultaneously performing Tensor Core operations on the current tile.
- **Vectorized Parallelism:** The use of `T.Parallel` hints to the compiler that loop iterations are independent, enabling the generation of wide vectorized load/store instructions (e.g., `LDG.128`).

- **Tensor Core Acceleration:** Arithmetic-heavy operations are offloaded to hardware-accelerated units via `T.gemm`, significantly boosting FLOPS compared to standard SIMD evaluations.

In **Parallel Rebased**, TileLang utilizes `T.annotate_layout` to apply a swizzled memory layout to shared buffers. This is a critical optimization that permutes the mapping of indices to memory banks, effectively eliminating shared memory bank conflicts and maximizing effective bandwidth during tile loading.

In **Parallel Retention**, the observed performance degradation can be attributed to the frequent use of `T.sync_threads()`. Unlike the pipelined Rebased kernel, the Retention implementation requires explicit barriers to coordinate complex element-wise math (such as the `T.exp2` decay) and precision casting.

In **Chunkwise Linear Attention**, the implementation employs a recurrent strategy. It maintains a persistent state (`S_state`) in shared memory, allowing the kernel to decompose the computation into an intra-chunk causal path and an inter-chunk linear recurrence. This effectively reduces the global memory footprint by transforming the complexity from quadratic to linear relative to sequence length.

N.5 DSL-Monkeys Generated TileLang Kernels for Linear Attention Variants

Based Parallel Kernel (Arora et al., 2025)

```
def _build_based_flash_attn_kernel(
    batch: int,
    heads: int,
    seq_len: int,
    dim: int,
    block_m: int = 64,
    block_n: int = 64
):
    # Precompute scaling factor d**-0.5
    scale = 1.0 / math.sqrt(dim)

    @T.prim_func
    def based_kernel(
        Q: T.Tensor((batch, heads, seq_len, dim), "bfloat16"),
        K: T.Tensor((batch, heads, seq_len, dim), "bfloat16"),
        V: T.Tensor((batch, heads, seq_len, dim), "bfloat16"),
        Out: T.Tensor((batch, heads, seq_len, dim), "bfloat16")
    ):
        # Grid: (seq_len // block_m, heads, batch)
        with T.Kernel(T.ceildiv(seq_len, block_m), heads, batch,
    threads=128) as (bx, by, bz):
            # Shared Memory Allocations
            Q_s = T.alloc_shared((block_m, dim), "bfloat16")
            K_s = T.alloc_shared((block_n, dim), "bfloat16")
            V_s = T.alloc_shared((block_n, dim), "bfloat16")

            # Shared memory for scores to be used as input for the
    second GEMM
            Scores_s = T.alloc_shared((block_m, block_n), "bfloat16")

            # Fragment Accumulators
            # acc_o: accumulates the weighted sum of values
    (numerator)
            acc_o = T.alloc_fragment((block_m, dim), "float32")
            # acc_z: accumulates the sum of weights (denominator)
            acc_z = T.alloc_fragment((block_m,), "float32")
            # scores: intermediate storage for Q @ K.T
            scores = T.alloc_fragment((block_m, block_n), "float32")
            # temp_z: temporary storage for row-wise reduction of
    current block
```

```
            temp_z = T.alloc_fragment((block_m,), "float32")

            # Layout optimization
            T.annotate_layout({
                Q_s: tilelang.layout.make_swizzled_layout(Q_s),
                K_s: tilelang.layout.make_swizzled_layout(K_s),
                V_s: tilelang.layout.make_swizzled_layout(V_s),
                Scores_s:
    tilelang.layout.make_swizzled_layout(Scores_s),
            })

            # Initialize accumulators
            T.clear(acc_o)
            T.clear(acc_z)

            # Load Q block for this iteration
            # We scale Q immediately upon loading or during GEMM?
            # For simplicity, we load raw Q, then multiply scale
    during the polynomial computation
            # or just scale Q_s in place. Scaling Q_s in place is
    cleaner.
            T.copy(Q[bz, by, bx * block_m : (bx + 1) * block_m, :],
    Q_s)

            # Apply scaling to Q_s
            for i, d in T.Parallel(block_m, dim):
                Q_s[i, d] = Q_s[i, d] * scale

            # Pipelined loop over Key/Value blocks
            # We only iterate up to bx (inclusive) for causal
    attention
            for k_idx in T.Pipelined(bx + 1, num_stages=2):
                # Load K and V blocks
                T.copy(K[bz, by, k_idx * block_n : (k_idx + 1) *
    block_n, :], K_s)
                T.copy(V[bz, by, k_idx * block_n : (k_idx + 1) *
    block_n, :], V_s)

                # 1. Compute Scores = Q_s @ K_s.T
                T.clear(scores)
                T.gemm(Q_s, K_s, scores, transpose_B=True)

                # 2. Apply Polynomial Activation and Causal Masking
                # phi(x) = 1 + x + 0.5 * x^2
                for i, j in T.Parallel(block_m, block_n):
                    # Causal logic: valid if (k_idx < bx) OR (k_idx
    == bx AND j <= i)
                    # Note: Since block_m == block_n == 64, direct
    comparison j <= i works for diagonal blocks
                    is_causal = (k_idx < bx) or (j <= i)

                    val = scores[i, j]
                    poly = 1.0 + val + 0.5 * val * val

                    # Mask out non-causal entries
                    scores[i, j] = T.if_then_else(is_causal, poly,
    0.0)

                # 3. Accumulate Normalizer (Z)
                # Reduce row-wise sum of the activated scores
                T.reduce_sum(scores, temp_z, dim=1)
                for i in T.Parallel(block_m):
                    acc_z[i] += temp_z[i]

                # 4. Accumulate Output (O)
```

```
                # We need scores in shared memory to multiply with V_s (scores @ V_s)
                # Cast fp32 scores to bf16 for tensor core GEMM input
                T.copy(scores, Scores_s)

                # acc_o += Scores_s @ V_s
                T.gemm(Scores_s, V_s, acc_o)

            # Epilogue: Normalization and Write-back
            for i, d in T.Parallel(block_m, dim):
                # Add epsilon for numerical stability
                acc_o[i, d] = acc_o[i, d] / (acc_z[i] + 1e-6)

            # Store result to global memory
            T.copy(acc_o, Out[bz, by, bx * block_m : (bx + 1) * block_m, :])

    return tilelang.compile(based_kernel, out_idx=[3], target="cuda")
```

ReBased Parallel Kernel (Aksenov et al., 2024)

```
def _build_rebased_kernel(
    batch: int,
    heads: int,
    seq_len: int,
    dim: int,
    scale: float,
    block_M: int = 64,
    block_N: int = 64,
    dtype: str = "bfloat16",
    accum_dtype: str = "float32",
):
    @T.prim_func
    def rebased_attention(
        Q: T.Tensor((batch, heads, seq_len, dim), dtype),
        K: T.Tensor((batch, heads, seq_len, dim), dtype),
        V: T.Tensor((batch, heads, seq_len, dim), dtype),
        Output: T.Tensor((batch, heads, seq_len, dim), dtype),
    ):
        # Grid: (bx: query block, by: head, bz: batch)
        with T.Kernel(
            T.ceildiv(seq_len, block_M), heads, batch, threads=128
        ) as (bx, by, bz):
            # Shared memory allocations
            Q_shared = T.alloc_shared((block_M, dim), dtype)
            K_shared = T.alloc_shared((block_N, dim), dtype)
            V_shared = T.alloc_shared((block_N, dim), dtype)
            # Intermediate scores in shared memory for GEMM 2 operand
            S_shared = T.alloc_shared((block_M, block_N), dtype)

            # Register accumulators
            acc_s = T.alloc_fragment((block_M, block_N), accum_dtype)
            acc_o = T.alloc_fragment((block_M, dim), accum_dtype)
            acc_z = T.alloc_fragment((block_M,), accum_dtype)

            # Helper fragment for reductions
            row_sum = T.alloc_fragment((block_M,), accum_dtype)

            # Layout optimization
            T.annotate_layout({
                Q_shared: tilelang.layout.make_swizzled_layout(Q_shared),
                K_shared: tilelang.layout.make_swizzled_layout(K_shared),
```

```
                V_shared:
    tilelang.layout.make_swizzled_layout(V_shared),
                S_shared:
    tilelang.layout.make_swizzled_layout(S_shared),
            })

            # Initialize accumulators
            T.fill(acc_o, 0)
            T.fill(acc_z, 0)

            # Load Q once per block
            for i, j in T.Parallel(block_M, dim):
                row = bx * block_M + i
                if row < seq_len:
                    Q_shared[i, j] = Q[bz, by, row, j]
                else:
                    Q_shared[i, j] = 0

            # Loop over K/V blocks. Since it's causal, we only go up
    to the current query block.
            loop_range = bx + 1
            for k in T.Pipelined(loop_range, num_stages=2):
                # Load K and V
                for i, j in T.Parallel(block_N, dim):
                    row = k * block_N + i
                    if row < seq_len:
                        K_shared[i, j] = K[bz, by, row, j]
                        V_shared[i, j] = V[bz, by, row, j]
                    else:
                        K_shared[i, j] = 0
                        V_shared[i, j] = 0

                # Clear score accumulator for this chunk
                T.clear(acc_s)

                # GEMM 1: Q @ K.T
                T.gemm(Q_shared, K_shared, acc_s, transpose_B=True)

                # Element-wise operations: Scale, Square, Causal Mask
                # Fix: Use distinct variables to avoid 'already
    defined' error
                for i, j in T.Parallel(block_M, block_N):
                    global_q_idx = bx * block_M + i
                    global_k_idx = k * block_N + j

                    # Apply scale
                    s_raw = acc_s[i, j]
                    s_scaled = s_raw * scale

                    # Square
                    s_sq = s_scaled * s_scaled

                    # Causal Masking
                    if global_k_idx > global_q_idx or global_k_idx >=
    seq_len:
                        acc_s[i, j] = 0.0
                    else:
                        acc_s[i, j] = s_sq

                # Accumulate row sums for normalization (Z)
                T.reduce_sum(acc_s, row_sum, dim=1)
                for i in T.Parallel(block_M):
                    acc_z[i] += row_sum[i]

```

```
                # Prepare S for GEMM 2: Copy from fp32 fragment to bf16 shared
                T.copy(acc_s, S_shared)

                # GEMM 2: S @ V
                T.gemm(S_shared, V_shared, acc_o)

            # Final Normalization and Store
            epsilon = 1e-6
            for i, j in T.Parallel(block_M, dim):
                global_q_idx = bx * block_M + i
                if global_q_idx < seq_len:
                    # Normalize
                    out_val = acc_o[i, j] / (acc_z[i] + epsilon)
                    Output[bz, by, global_q_idx, j] = out_val

    return tilelang.compile(rebased_attention, out_idx=[3], target="cuda")
```

RetNet Parallel Kernel (Sun et al., 2023)

```
def _build_retention_kernel(
    batch_size: int,
    num_heads: int,
    seq_len: int,
    head_dim: int,
    block_M: int = 64,
    block_N: int = 64,
    dtype: str = "bfloat16",
    accum_dtype: str = "float32",
):
    # Calculate scale constant
    scale = 1.0 / math.sqrt(head_dim)

    @T.prim_func
    def retention_kernel(
        Q: T.Tensor((batch_size, num_heads, seq_len, head_dim), dtype),
        K: T.Tensor((batch_size, num_heads, seq_len, head_dim), dtype),
        V: T.Tensor((batch_size, num_heads, seq_len, head_dim), dtype),
        Output: T.Tensor((batch_size, num_heads, seq_len, head_dim), dtype),
    ):
        # Grid: (query_blocks, heads, batch)
        with T.Kernel(
            T.ceildiv(seq_len, block_M), num_heads, batch_size, threads=128
        ) as (bx, by, bz):
            # Shared memory allocations
            Q_shared = T.alloc_shared((block_M, head_dim), dtype)
            K_shared = T.alloc_shared((block_N, head_dim), dtype)
            # Use accum_dtype for V to maintain higher precision in the second GEMM
            V_shared = T.alloc_shared((block_N, head_dim), accum_dtype)

            # Fragment allocations
            acc_o = T.alloc_fragment((block_M, head_dim), accum_dtype)
            scores = T.alloc_fragment((block_M, block_N), accum_dtype)

            # Intermediate shared memory buffers
            # scores_shared must match fragment dtype (float32) for T.copy
```

```
        scores_shared = T.alloc_shared((block_M, block_N), accum_dtype)
        # Use accum_dtype for p matrix to avoid truncation errors before V multiplication
        p_shared = T.alloc_shared((block_M, block_N), accum_dtype)

        # Calculate head-specific decay slope s in float32
        # s = log2(1 - 2^(-5 - head_idx))
        head_idx_f32 = T.cast(by, "float32")
        exponent = -5.0 - head_idx_f32
        term = 1.0 - T.exp2(exponent)
        s_val = T.log2(term)

        # Initialize accumulator
        T.clear(acc_o)

        # Load Q block
        for i, j in T.Parallel(block_M, head_dim):
            row = bx * block_M + i
            if row < seq_len:
                Q_shared[i, j] = Q[bz, by, row, j]
            else:
                Q_shared[i, j] = 0.0
        T.sync_threads()

        # Loop limit for K/V blocks (Causal: only up to current block)
        loop_limit = bx + 1

        # Use standard range loop
        for k_block in range(loop_limit):
            # Load K and V blocks
            for i, j in T.Parallel(block_N, head_dim):
                row = k_block * block_N + i
                if row < seq_len:
                    K_shared[i, j] = K[bz, by, row, j]
                    # Cast V to accum_dtype
                    V_shared[i, j] = T.cast(V[bz, by, row, j], accum_dtype)
                else:
                    K_shared[i, j] = 0.0
                    V_shared[i, j] = 0.0

            T.sync_threads()

            # Compute Scores = Q @ K.T
            T.clear(scores)
            # Q (bf16) @ K (bf16) -> scores (fp32)
            T.gemm(Q_shared, K_shared, scores, transpose_B=True)

            # Move scores to shared for element-wise ops
            T.copy(scores, scores_shared)

            T.sync_threads()

            # Apply causal mask and decay pattern element-wise on shared memory
            base_i = bx * block_M
            base_j = k_block * block_N

            for i, j in T.Parallel(block_M, block_N):
                row_global = base_i + i
                col_global = base_j + j

                # Check causal mask
```

```
                    if row_global >= col_global:
                        # Computation in float32
                        diff = T.cast(row_global - col_global, "float32")
                        decay_exponent = diff * s_val
                        decay = T.exp2(decay_exponent)

                        val_f32 = scores_shared[i, j]
                        scaled_val = val_f32 * scale * decay

                        # Store as float32
                        p_shared[i, j] = scaled_val
                    else:
                        p_shared[i, j] = 0.0

                # Ensure modifications are visible before V multiplication
                T.sync_threads()

                # Accumulate into acc_o using modified scores
                # p (fp32) @ V (fp32) -> acc_o (fp32)
                T.gemm(p_shared, V_shared, acc_o)

                T.sync_threads()

            # Store Output
            for i, j in T.Parallel(block_M, head_dim):
                row = bx * block_M + i
                if row < seq_len:
                    Output[bz, by, row, j] = T.cast(acc_o[i, j], dtype)

    return tilelang.compile(retention_kernel, out_idx=[3], target="cuda")
```

Chunked Linear Attention Kernel (Yang & Zhang, 2024a)

```
def _build_linear_attn_kernel(
    batch_size,
    num_heads,
    seq_len,
    head_dim,
    chunk_size,
    n_chunks
):
    grid_size = batch_size * num_heads
    dtype = "bfloat16"
    accum_dtype = "float32"

    @T.prim_func
    def kernel(
        Q: T.Tensor((batch_size, seq_len, num_heads, head_dim), dtype),
        K: T.Tensor((batch_size, seq_len, num_heads, head_dim), dtype),
        V: T.Tensor((batch_size, seq_len, num_heads, head_dim), dtype),
        Out: T.Tensor((batch_size, seq_len, num_heads, head_dim), dtype),
        scale: T.float32
    ):
        with T.Kernel(grid_size, threads=128) as bz:
            batch_idx = bz // num_heads
            head_idx = bz % num_heads

```

```
            # Persistent State in Shared Memory
            # CHANGED: Use dtype (bf16) instead of accum_dtype (fp32)
    to match reference precision
            S_state = T.alloc_shared((head_dim, head_dim), dtype)

            # Temporary Buffers
            # CHANGED: Use dtype (bf16) for intermediate storage
            sOut_inter = T.alloc_shared((chunk_size, head_dim), dtype)
            sOut_intra = T.alloc_shared((chunk_size, head_dim), dtype)
            S_kv_update = T.alloc_shared((head_dim, head_dim), dtype)

            # Input Tiles
            sQ = T.alloc_shared((chunk_size, head_dim), dtype)
            sK = T.alloc_shared((chunk_size, head_dim), dtype)
            sV = T.alloc_shared((chunk_size, head_dim), dtype)

            # Scores Tile
            sScores = T.alloc_shared((chunk_size, chunk_size), dtype)

            # Fragments (Keep accumulators in FP32 for GEMM precision)
            acc_o = T.alloc_fragment((chunk_size, head_dim),
    accum_dtype)
            acc_scores = T.alloc_fragment((chunk_size, chunk_size),
    accum_dtype)
            acc_update = T.alloc_fragment((head_dim, head_dim),
    accum_dtype)

            T.annotate_layout({
                sQ: tilelang.layout.make_swizzled_layout(sQ),
                sK: tilelang.layout.make_swizzled_layout(sK),
                sV: tilelang.layout.make_swizzled_layout(sV),
                sScores:
    tilelang.layout.make_swizzled_layout(sScores),
                S_state:
    tilelang.layout.make_swizzled_layout(S_state),
                S_kv_update:
    tilelang.layout.make_swizzled_layout(S_kv_update),
                sOut_inter:
    tilelang.layout.make_swizzled_layout(sOut_inter),
                sOut_intra:
    tilelang.layout.make_swizzled_layout(sOut_intra),
            })

            # Initialize State to Zero
            for i, j in T.Parallel(head_dim, head_dim):
                S_state[i, j] = T.cast(0.0, dtype)
            T.copy(S_state, S_state) # Barrier

            for c in range(n_chunks):
                t_base = c * chunk_size

                # 1. Load Inputs
                for i, j in T.Parallel(chunk_size, head_dim):
                    t = t_base + i
                    if t < seq_len:
                        sQ[i, j] = Q[batch_idx, t, head_idx, j] *
    T.cast(scale, dtype)
                        sK[i, j] = K[batch_idx, t, head_idx, j]
                        sV[i, j] = V[batch_idx, t, head_idx, j]
                    else:
                        sQ[i, j] = T.cast(0.0, dtype)
                        sK[i, j] = T.cast(0.0, dtype)
                        sV[i, j] = T.cast(0.0, dtype)

                T.copy(sQ, sQ)
```

```
                T.copy(sK, sK)
                T.copy(sV, sV)

                # 2. Inter-Chunk Contribution: O_inter = Q @ S_prev
                # Use S_state directly as it is now in correct dtype
                T.clear(acc_o)
                T.gemm(sQ, S_state, acc_o)
                T.copy(acc_o, sOut_inter)

                # 3. Intra-Chunk Contribution: O_intra = Mask(Q @ K.T) @ V
                T.clear(acc_scores)
                T.gemm(sQ, sK, acc_scores, transpose_B=True)
                T.copy(acc_scores, sScores)
                T.copy(sScores, sScores)

                # Apply Causal Mask
                for i, j in T.Parallel(chunk_size, chunk_size):
                    if j > i:
                        sScores[i, j] = T.cast(0.0, dtype)
                T.copy(sScores, sScores)

                T.clear(acc_o)
                T.gemm(sScores, sV, acc_o)
                # Store to dedicated buffer sOut_intra
                T.copy(acc_o, sOut_intra)
                T.copy(sOut_intra, sOut_intra) # Barrier ensures visibility for step 4

                # 4. Final Accumulation and Output Store
                for i, j in T.Parallel(chunk_size, head_dim):
                    # Safe read: sOut_inter and sOut_intra are distinct and fully written
                    val = sOut_inter[i, j] + sOut_intra[i, j]
                    t = t_base + i
                    if t < seq_len:
                        Out[batch_idx, t, head_idx, j] = val

                # 5. Update State: S_state += K.T @ V
                T.clear(acc_update)
                T.gemm(sK, sV, acc_update, transpose_A=True)

                # Move to shared buffer S_kv_update (casts to bf16)
                T.copy(acc_update, S_kv_update)
                T.copy(S_kv_update, S_kv_update) # Barrier

                for i, j in T.Parallel(head_dim, head_dim):
                    S_state[i, j] += S_kv_update[i, j]
                T.copy(S_state, S_state) # Barrier for next chunk iteration

    return tilelang.compile(kernel, out_idx=3, target="cuda")
```