PROCESSKERNEL: CALIBRATED PROCESS SUPERVISION FOR GPU KERNEL GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Generating efficient GPU kernels requires searching a vast space of functionally equivalent programs whose performance depends on low-level hardware behavior. Effective search therefore relies on execution feedback to identify promising implementations. In practice, however, obtaining such feedback is cumbersome and expensive: candidate kernels must be compiled, validated, and evaluated under a reliable testing setup, making repeated execution a major bottleneck to scalable kernel search and generation. We thus introduce an execution-free process supervision framework that turns one-time outcome measurements into reusable guidance for unfinished kernels during autoregressive generation, enabling on-the-fly search without executing intermediate candidates. Extensive experiments show that our approach reduces process-supervision labeling cost by 76$\times$ (9.3 vs. an estimated 707 H100 GPU-hours), while improving generated-kernel correctness by up to 38% and achieving up to 21.64$\times$ speedup. These results highlight the promising potential of reusable process-level supervision to amortize costly execution feedback and enable more scalable search for correct and efficient GPU kernels.

1 INTRODUCTION

Efficient GPU kernels are fundamental to modern machine-learning systems, as they ultimately determine how high-level computations are executed on parallel hardware. A GPU kernel is a low-level program that specifies how computation is partitioned across threads, how data are moved through the memory hierarchy, and how hardware resources are utilized. Kernel design is therefore a specialized programming task: different implementations of the same operation can vary substantially in compute efficiency, memory traffic, and runtime. Carefully optimized kernels can fuse operations, improve data reuse, and avoid costly intermediate memory transfers, but achieving these gains requires expertise in parallel decomposition, memory access patterns, and hardware-specific optimization.

Large language models (LLMs) increasingly automate kernel development by synthesizing custom `Triton` (Tillet et al., 2019) or `CUDA` implementations from reference operators. Unlike conventional code generation, GPU kernel generation must optimize not only functional correctness but also hardware-dependent efficiency. Existing methods therefore place measured correctness and runtime in the optimization loop: training-based approaches use execution outcomes to update kernel generators (Fisches et al., 2025; Baronio et al., 2025; Liu et al., 2026), while agentic and evolutionary approaches repeatedly execute complete candidates to guide search (Du et al., 2026; Liao et al., 2026; Dong et al., 2026). In practice, however, obtaining such feedback is cumbersome and expensive: each candidate must be fully completed, compiled, and timed under a reliable testing setup, making repeated execution a major bottleneck to scalable kernel search and generation.

We seek to reduce the cost of GPU-kernel search by avoiding the need to execute every generated candidate. Doing so requires a way to estimate, during generation, which decisions are likely to lead to correct and efficient completed kernels before execution feedback becomes available. This motivates a *learned process-level scorer* that can evaluate intermediate generation states and guide search toward promising continuations without repeated compilation and execution during inference. The scorer should operate independently of the generator, requiring no retraining or modification of the generator itself and allowing it to transfer seamlessly across different generators, including those unseen during scorer training. The central challenge is how to obtain faithful supervision for such a scorer. Prior work derives intermediate signals from properties that can be locally verified through

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

execution, mutations, or refactorings (Li et al., 2025a; Ye et al., 2025), but GPU kernel generation differs since correctness and efficiency are properties of the completed program. Local correctness is thus an inadequate target: later code may repair an earlier mistake, while a seemingly sound prefix may still lead to an invalid kernel. We instead supervise the continuation value of an intermediate state, i.e., the expected quality of the completed kernels reachable from it.

A direct way to estimate continuation value is through Monte Carlo rollouts, but this is expensive at scale: each prefix must be completed multiple times, and every completion must then be executed. Since a single generated completion contains many prefixes, this approach repeatedly pays the execution cost to supervise intermediate states. Our key idea is thus to *amortize a much smaller set of measured executions into scalable process supervision*. Specifically, we first train an **outcome reward model (ORM)** on complete kernels using measured correctness-and-speed grades to learn within-problem rankings, avoiding direct regression of absolute values that can vary substantially across problems. Since ORM scores are ordinal rather than calibrated values, we map them to terminal values using a small set of executed anchor kernels. The calibrated ORM can then estimate the value of many sampled continuations without executing them, converting sparse terminal measurements into dense supervision for a **process reward model (PRM)** over unfinished generations. At inference time, the PRM guides segment-level beam search, and the ORM selects the final kernel from the candidate pool, with no execution required at either stage. See Figure 1 for an overview.

By amortizing a limited set of executed kernels into reusable process supervision, our framework substantially reduces the execution burden of GPU-kernel search without relying on language-specific search signals. Empirically, the imputed labels preserve most measured pairwise preferences while reducing continuation-labeling cost by $76\times$. This lower-cost supervision also yields stronger candidate pools, with process-guided search improving kernel correctness by up to 38% and speedup by up to $21.64\times$. These results highlight the potential of amortizing sparse execution feedback into reusable process-level guidance, enabling more scalable search for correct and efficient GPU kernels.

2 Related Work

LLM-Based GPU Kernel Generation. Recent work on LLM-based GPU-kernel synthesis can be broadly grouped into two paradigms: *generator specialization* and *search-time optimization*. Generator-specialization methods directly adapt an LLM for kernel synthesis through supervised fine-tuning (Paliskara & Saroufim, 2025; Fisches et al., 2025); reinforcement learning with execution feedback (Baronio et al., 2025; Liu et al., 2026); or execution-derived post-training (Du et al., 2026). These methods can internalize kernel-specific optimization behavior, but their gains are tied to the adapted generator and do not directly transfer to a different model. Search-time methods avoid this coupling by leaving the generator unchanged and instead allocating additional inference-time computation to explore and refine candidate kernels. Systems such as `KernelEvolve` (Liao et al., 2026) and `KernelBlaster` (Dong et al., 2026) repeatedly compile, validate, profile, and execute completed candidates to decide which ones to retain or refine. This improves portability across generators, but makes search dependent on a slow and operationally cumbersome execution loop. Our approach retains the same generator-agnostic advantage while replacing repeated candidate execution with a learned process scorer that guides generation directly, enabling efficient search without repeatedly invoking compilation and execution.

Process Reward Models for Output Search. Effective output search requires scoring intermediate generations before their final outcomes are known. In mathematical reasoning, such process supervision can come from human judgments (Lightman et al., 2023) or from Monte Carlo rollouts that estimate how often continuations reach the correct answer (Wang et al., 2024). Luo et al. (2024) then reduce this rollout cost by identifying likely error points with fewer evaluations. Extending this idea to code is more difficult because intermediate mistakes are not necessarily irreversible. Recent code-oriented methods therefore derive process labels from execution-based pass rates, as in `CodePRM` (Li et al., 2025a), or learn to predict execution-derived properties of complete programs without repeatedly running them, as in `RM-RF` (Bruches et al., 2026). `DeepConf` similarly guides intermediate generations without execution, but relies on the generator's token confidence rather than task-specific outcome supervision (Fu et al., 2025). Our method extends this line to GPU-kernel generation by defining the value of an unfinished kernel as the expected correctness-and-runtime quality of its completions. A calibrated outcome model transfers measured terminal feedback to

these continuation values, allowing many intermediate states to be scored without executing each continuation and providing task-specific guidance throughout search.

3 METHOD

Notation. A problem $\boldsymbol{x}$ specifies a computation to implement, typically through a high-level reference program (e.g., in `PyTorch`) together with example inputs. Let $f_{\boldsymbol{\theta}}$ denote the LLM generator, which maps $\boldsymbol{x}$ to a GPU kernel $\boldsymbol{y}$, i.e., a low-level implementation of the same computation (e.g., in `CUDA` or `Triton`). A complete kernel receives a measured terminal grade $v(\boldsymbol{x}, \boldsymbol{y})$ based on compilation, correctness, and runtime, which we treat as its ground-truth quality. We set $v(\boldsymbol{x}, \boldsymbol{y}) = 0$ for kernels that fail to compile or produce incorrect outputs, and $v(\boldsymbol{x}, \boldsymbol{y}) = (1+q)/2$ otherwise, where $q \in [0, 1]$ is a discretized, clamped log-scale transformation of speedup over the reference implementation.

3.1 OUTCOME REWARD MODELING FOR COMPLETE KERNELS

To guide generation without repeatedly executing candidates, we first need a learned surrogate for the quality of a complete kernel. The ORM $s(\boldsymbol{x}, \boldsymbol{y})$ takes a problem specification $\boldsymbol{x}$ and a complete generated kernel $\boldsymbol{y}$, and outputs a scalar quality score. Although execution provides a measured terminal grade $v(\boldsymbol{x}, \boldsymbol{y})$, its numerical scale is problem-dependent: runtime and speedup are meaningful relative to other implementations of the same computation, but are not naturally comparable across different problems. What is consistently available is the ordering induced by these measurements—for a given problem, we can determine which candidate is better.

We therefore train the ORM as a within-problem ranker rather than regressing directly to $v(\boldsymbol{x}, \boldsymbol{y})$. Each training list contains the compiling candidates for one problem, ordered by their measured terminal grades. Non-compiling kernels are excluded because compilation identifies them directly without kernel execution, allowing the ORM to focus on the harder distinctions among compiling candidates: which are correct and, among correct kernels, which are faster. The resulting scalar score captures relative kernel quality, but is not yet a calibrated estimate of the terminal grade. Specifically, we optimize the ORM with LambdaRank (Burges et al., 2006). For each problem, we form ordered pairs (i, j) such that $v_i > v_j$, where $v_i = v(\boldsymbol{x}, \boldsymbol{y}_i)$ and $s_i = s(\boldsymbol{x}, \boldsymbol{y}_i)$ (and likewise for j). LambdaRank applies a logistic loss to the score difference $s_i - s_j$, weighted by Δ_{ij}, the change in Normalized Discounted Cumulative Gain (NDCG) (Järvelin & Kekäläinen, 2002) induced by swapping the two candidates in the predicted ranking. Thus, misorderings that have a larger effect on ranking quality receive larger weights (see Appendix A.2).

Our terminal grade induces two types of pairwise comparisons. A *correctness pair* compares a correct kernel with an incorrect one, so $\mathcal{P}_{\text{corr}} = \{(i, j)\colon v_i{>}0 \text{ and } v_j{=}0\}$. A *speed pair* compares two correct kernels with different measured performance, so $\mathcal{P}_{\text{speed}} = \{(i, j)\colon v_i{>}v_j{>}0\}$. As correct kernels for the same problem often receive similar grades, speed pairs typically have smaller Δ_{ij} weights and would otherwise contribute much less to the ranking objective. We therefore normalize the two types of comparisons separately:

$$\mathcal{L}_{\text{corr}} = \frac{\sum_{(i,j)\in\mathcal{P}_{\text{corr}}} \Delta_{ij} \log\left(1 + e^{-(s_i - s_j)}\right)}{\sum_{(i,j)\in\mathcal{P}_{\text{corr}}} \Delta_{ij}}, \quad \mathcal{L}_{\text{speed}} = \frac{\sum_{(i,j)\in\mathcal{P}_{\text{speed}}} \Delta_{ij} \log\left(1 + e^{-(s_i - s_j)}\right)}{\sum_{(i,j)\in\mathcal{P}_{\text{speed}}} \Delta_{ij}} \tag{1}$$

The final loss is a convex combination, i.e., $\mathcal{L} = \alpha\mathcal{L}_{\text{corr}} + (1-\alpha)\mathcal{L}_{\text{speed}}$ where α controls the relative contribution of the two terms.

3.2 PROCESS REWARD MODELING FOR PARTIAL KERNELS

The ORM provides a signal only after a kernel has been completed, making it useful for final selection but unable to influence how generation compute is spent along the way. Under a fixed budget, this wastes compute on weak rollouts that may already look unpromising long before they terminate. We therefore introduce a PRM that scores partial kernels online, enabling a single generation run to redirect compute toward promising prefixes without repeated compilation or GPU execution.

Rollout-Based Prefix Value. Assigning supervision to a partial kernel is less straightforward than grading a complete one. A prefix that currently contains an error may still lead to a correct kernel

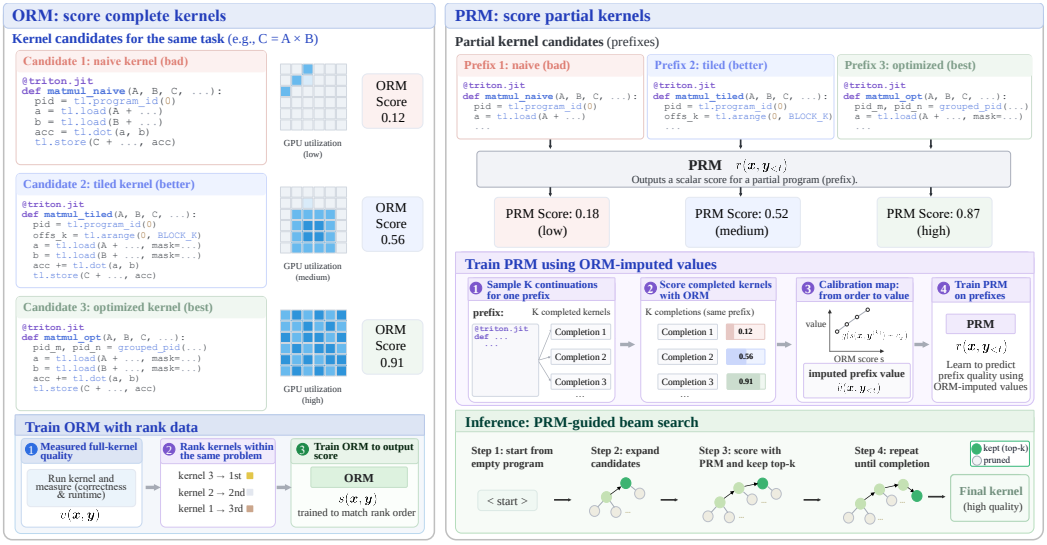


Figure 1. Overview of our PROCESSKERNEL framework.

if later code repairs or overwrites it, while an apparently valid prefix may nevertheless terminate incorrectly. We therefore do not assign a binary correctness label to a prefix or attempt to identify a first erroneous step. Instead, we define its value by the quality of the complete kernels that can be generated from it. Specifically, for a problem $\boldsymbol{x}$ and prefix $\boldsymbol{y}_{<t}$, we define the process value as the expected terminal grade under the continuation distribution, i.e., $\bar{v}(\boldsymbol{x}, \boldsymbol{y}_{<t}) = \mathbb{E}_{\boldsymbol{y}' \sim f_{\boldsymbol{\theta}}(\cdot|\boldsymbol{x}, \boldsymbol{y}_{<t})} [v(\boldsymbol{x}, \boldsymbol{y}')]$. Since this expectation cannot be computed exactly, we estimate it with Monte Carlo rollouts:

$$\boldsymbol{y}^{(k)} \sim f_{\boldsymbol{\theta}}(\cdot \mid \boldsymbol{x}, \boldsymbol{y}_{<t}), \quad \hat{v}(\boldsymbol{x}, \boldsymbol{y}_{<t}) = \frac{1}{K} \sum_{k=1}^{K} v(\boldsymbol{x}, \boldsymbol{y}^{(k)}). \tag{2}$$

using K fresh continuations sampled from $\boldsymbol{y}_{<t}$. Because the terminal grade v captures both correctness and execution performance, $\bar{v}$ reflects both the likelihood that a prefix leads to a correct kernel and the quality of the correct kernels it produces, while $\hat{v}$ is its finite-sample estimate. We exclude the original completion from which the prefix was obtained so that the label reflects the distribution of possible futures from that prefix rather than the particular outcome that happened to generate it. Rollouts use the same decoding configuration as the original generation run, ensuring that the estimated value is aligned with the continuation policy used at inference time.

Execution-Free Prefix Value Imputation. Directly estimating $\hat{v}$ at training scale is expensive: every prefix requires K continuations, each of which must be compiled and executed on a GPU. Since a single completion can yield multiple prefixes, rollout-based measurement substantially multiplies the cost of terminal evaluation. We therefore measure $\hat{v}$ only on a held-out subset and use the ORM as a surrogate grader for the training corpus. Ideally, we would replace each executed terminal grade $v(\boldsymbol{x}, \boldsymbol{y}^{(k)})$ in the rollout estimator with a cheap ORM prediction. However, the ORM score $s(\boldsymbol{x}, \boldsymbol{y})$ cannot be used directly: it is trained to order kernels within a problem rather than to reproduce their numerical terminal grades. This is intentional, since terminal-grade scales vary across problems whereas within-problem ordering remains meaningful. Prefix-value estimation, however, averages continuation outcomes and therefore requires surrogate predictions on a common value scale.

To map ORM scores onto the terminal-grade scale, we use a set of *anchors*: complete kernels for which both an ORM score $s(\boldsymbol{x}, \boldsymbol{y})$ and a measured terminal grade $v(\boldsymbol{x}, \boldsymbol{y})$ are available. Using these anchors, we fit a monotone mapping $\hat{g}$ from ORM score to terminal grade with isotonic regression (Ayer et al., 1955; Zadrozny & Elkan, 2002), which chooses a non-decreasing function whose predictions $g(s(\boldsymbol{x}, \boldsymbol{y}))$ are as close as possible to the corresponding measured grades $v(\boldsymbol{x}, \boldsymbol{y})$, without assuming a particular functional form. This matches the ORM objective: higher scores should correspond to better kernels, while the absolute score scale itself remains unconstrained.

A single global mapping is still insufficient because ORM scores are comparable only within a problem and may be shifted across problems. We therefore assign each problem $\boldsymbol{x}$ a scalar offset

$c_{\boldsymbol{x}}$ and calibrate a complete kernel as $g(s(\boldsymbol{x}, \boldsymbol{y}) + c_{\boldsymbol{x}})$. The offset aligns each problem's ORM scores with its measured anchors, while the shared monotone mapping $\hat{g}$ converts the aligned scores to the terminal-grade scale. This separates problem-specific score alignment from the common score-to-grade relationship, allowing one CALIBRATION MAP to be shared across problems despite their arbitrary ORM score scales. We apply the offset before g so that predictions remain within the range of the calibrated mapping, and fit the offsets and g alternately. We exclude problems whose anchors all have zero terminal grade, since they provide no information for aligning that problem to the value scale. With this calibration, we replace each executed terminal grade in the rollout estimator with its calibrated ORM prediction: $\tilde{v}(\boldsymbol{x}, \boldsymbol{y}_{<t}) = \frac{1}{K} \sum_{k=1}^{K} g\big(s(\boldsymbol{x}, \boldsymbol{y}^{(k)}) + c_{\boldsymbol{x}}\big)$. Thus, $\tilde{v}$ retains the same rollout-based form as the measured estimate $\hat{v}$, but replaces each kernel execution with an ORM forward pass. This removes repeated compilation and GPU execution from large-scale prefix labelling while keeping the imputed values tied to measured kernel outcomes through the ORM training data and calibration anchors. The calibration map is fitted on complete kernels sampled without conditioning on an intermediate prefix.

Training the PRM. The imputed process values $\tilde{v}$ provide scalable supervision for learning which partial kernels are more promising. The PRM $r(\boldsymbol{x}, \boldsymbol{y}_{<t})$ takes the problem specification and a prefix as input and outputs a scalar ranking score. We train it with the same ranking objective as the ORM, using $\tilde{v}$ only to determine the relative ordering of prefixes. Each training list contains prefixes from different completions of the same problem, truncated at the same index so that the model compares alternative states at a similar stage of generation. We discard lists with fewer than two prefixes or no variation in their imputed values. As with the ORM, the supervision provides an ordering signal rather than a calibrated prediction target: the PRM only needs to rank prefixes within a problem. This is sufficient for guided generation, where its role is to decide which partial kernels should receive further generation budget.

Steering Kernel Generation with Process Rewards. The PRM turns process supervision into an inference-time search policy by directing generation toward intermediate states that are more likely to yield high-quality kernels. We implement this with segment-level beam search. Starting from the current beam, each partial kernel is extended to a natural code boundary, such as the end of a logical line, and the PRM scores the resulting candidates. The highest-scoring candidates then form the next beam, so generation budget is progressively concentrated on more promising regions of the search space rather than spent uniformly on full rollouts. Completed kernels are collected throughout the search, while unfinished candidates continue to be expanded. The two reward models therefore play complementary roles: the PRM decides which partial generations to continue, while the ORM ranks the completed kernels for final selection. Both stages rely only on learned scores, avoiding compilation and execution of generated candidates during search.

Overall, our framework (See Figure 1) amortizes sparse execution outcomes into reusable process guidance, enabling search without further execution. The ORM and PRM are separate learned scorers designed to generalize across generator backbones. The training-time supervision flow is:

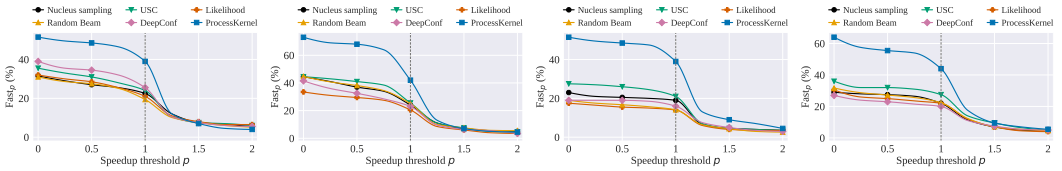
$$\underbrace{v(\boldsymbol{x}, \boldsymbol{y})}_{\text{terminal grade}} \xrightarrow{\text{rank}} \underbrace{s(\boldsymbol{x}, \boldsymbol{y})}_{\text{ORM}} \xrightarrow{\text{calibration}} \underbrace{\tilde{v}(\boldsymbol{x}, \boldsymbol{y}_{<t})}_{\text{imputed}} \approx \underbrace{\hat{v}(\boldsymbol{x}, \boldsymbol{y}_{<t})}_{\text{rollout}} \approx \underbrace{\bar{v}(\boldsymbol{x}, \boldsymbol{y}_{<t})}_{\text{true}} \xrightarrow{\text{rank}} \underbrace{r(\boldsymbol{x}, \boldsymbol{y}_{<t})}_{\text{PRM}}$$

where $\bar{v}$ is the true continuation value, $\hat{v}$ its execution-based rollout estimate, and $\tilde{v}$ the execution-free imputed value. During inference, the PRM uses $r(\boldsymbol{x}, \boldsymbol{y}_{<t})$ to decide which partial kernels to expand, while the ORM uses $s(\boldsymbol{x}, \boldsymbol{y})$ to rank completed candidates.

4 EXPERIMENTS

4.1 SETUP

Datasets, Models, and Hardware. We evaluate on the standard **KernelBench** benchmark (Ouyang et al., 2025), using level-1 problems with single primitive operations and level-2 problems with multi-operation sequences. We construct the training corpus from the `PyTorch` modules of `KernelBook` (Paliskara & Saroufim, 2025), which were mined from `GitHub`, and convert them to `KernelBench` format to obtain 13,309 kernel problems. Both reward models train on kernels from `DeepSeek-V4-Flash` (DeepSeek-AI, 2026), the ORM also on `GPT-OSS-120B` (OpenAI, 2025) and the PRM also on `Qwen3.5-122B` (Qwen Team, 2026) and `MiniMax-M2.7` (MiniMax,



(a) DR.Kernel-14B **(b)** Muse-Glimmer-30B **(c)** GPT-OSS-120B **(d)** DeepSeek-V4-Flash

Figure 2. Fast$_p$ profiles for four representative backbones, pooled across `KernelBench` Levels 1 and 2 (200 problems per backbone). At threshold p, Fast$_p$ is the expected percentage of problems for which the selected kernel is correct and achieves a speedup greater than p over PyTorch eager on the same GPU.

2026). Table 4 lists every generator we use, for training or evaluation, with its decoding settings. Both the ORM and PRM are fine-tuned from `Qwen3-Reranker-4B` (Zhang et al., 2025). Generation runs with `vLLM` (Kwon et al., 2023) on NVIDIA H100, A30 or GH200 GPUs, while kernel execution and timing are performed separately on a single idle H100. Full details are provided in Appendix B.1.

Metrics. We evaluate generated kernels using three primary metrics. **Correct** is the fraction of problems for which a correct kernel is produced. **Fast$_\mathbf{p}$** (Ouyang et al., 2025) is the fraction for which the selected kernel is correct and achieves more than a $p\times$ wall-clock speedup over the reference implementation. We report $\text{Fast}_1^{\text{eager}}$ and $\text{Fast}_1^{\text{compile}}$ using `PyTorch` eager and default `torch.compile` (Ansel et al., 2024) as the respective references. **Mean** is the average speedup across all problems, assigning zero to unsolved cases. Constant-output kernels and kernels that precompute sums or matrix products of the weights outside the timed call are counted as incorrect. Together, these metrics capture both solution coverage and execution efficiency.

Baselines and Implementation. We evaluate ProcessKernel in `Triton`, following the mainstream prior kernel-generation systems including `AutoTriton` (Li et al., 2025b), `KernelLLM` (Fisches et al., 2025), and `Dr. Kernel` (Liu et al., 2026). As existing inference-time kernel optimizers typically rely on online execution feedback or target different backends, we use controlled baselines under matched generation budgets: model Log-**Likelihood**, **Nucleus sampling**, **USC** (Chen et al., 2023), **Random Beam**, and **DeepConf** (Fu et al., 2025). In the execution-feedback setting, we additionally compare with **CodePRM** (Li et al., 2025a). See Appendix B.2 for details.

4.2 Comparison to Baselines

Process guidance consistently improves kernel generation across backbones. As shown in Table 1, ProcessKernel improves over nucleus sampling and alternative inference-time search baselines across both kernel-specialized and general-purpose LLMs. Relative to direct generation, it improves **Correct** and **Fast**$_1$ in all 16 generator–level settings, with gains of up to 38% and 34%, respectively; **Mean** speedup improves in 15 of 16 settings. These gains span kernel-specialized generators such as `AutoTriton` and `Dr. Kernel`, as well as general-purpose LLMs such as `GPT-OSS-120B` and `DeepSeek-V4-Flash`. ProcessKernel also generally outperforms Random Beam and DeepConf, indicating that the gains cannot be explained by branching or token-confidence-based guidance alone.

Process guidance primarily expands the range of problems solved with competitive kernels. Figure 2 shows how the advantage changes as the required speedup becomes more stringent. At low p, the gap mainly reflects the higher correctness coverage of ProcessKernel; as p approaches 1, it increasingly reflects kernels that are both correct and competitive with eager `PyTorch`. Across the representative backbones, ProcessKernel maintains the highest Fast$_p$ around the practically important $p = 1$ threshold, while the curves converge at larger p, where large speedups are rare for all methods. The gains are particularly pronounced on Level 2 for several backbones. For `DeepSeek-V4-Flash`, **Correct** increases from 33% to 71% and **Fast**$_1$ from 23% to 56%; for `GPT-OSS-120B`, the corresponding improvements are 23% to 58% and 18% to 49%. The same pattern extends to kernel-specialized generators: `DR.Kernel-8B` improves from 38% to 70% correct and from 23% to 57% at **Fast**$_1$. The relative advantage also remains under the stronger `torch.compile` reference, despite the reduced optimization headroom.

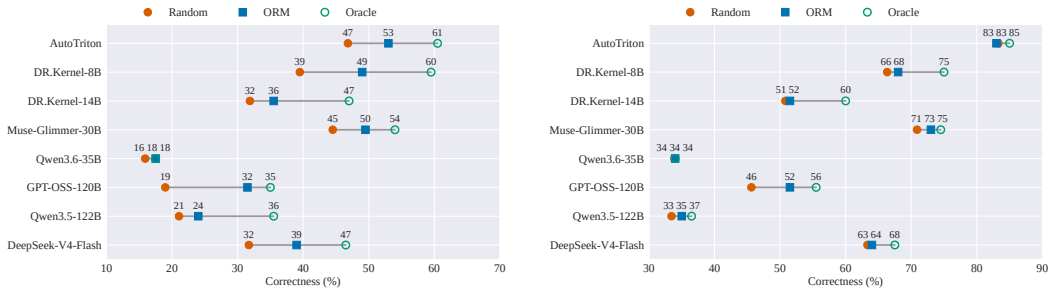
Table 1. Single-selected-kernel quality relative to PyTorch eager and `torch.compile`. The upper block reports kernel-specialized LLMs and the lower block general-purpose LLMs. Best results are shown in **bold**. Higher is better for all metrics.

Generator	Method	Level 1				Level 2			
		Correct ↑	$\text{Fast}_1^{\text{eager}}$ ↑	$\text{Fast}_1^{\text{compile}}$ ↑	Mean ↑	Correct ↑	$\text{Fast}_1^{\text{eager}}$ ↑	$\text{Fast}_1^{\text{compile}}$ ↑	Mean ↑
AutoTriton	Nucleus sampling	54.00	37.00	16.00	0.57	47.00	38.00	**9.00**	0.52
	USC	69.00	38.00	12.00	0.70	57.00	**43.00**	6.00	0.60
	Likelihood	48.00	26.00	9.00	0.44	47.00	32.00	**9.00**	0.46
	Random Beam	46.75	27.00	10.50	0.47	46.75	31.25	6.00	0.51
	DeepConf	48.00	20.00	8.00	0.46	57.00	33.00	7.00	0.60
	ProcessKernel (Ours)	**84.00**	**59.00**	**36.00**	**0.84**	**82.00**	41.00	7.00	**0.85**
DR.Kernel-8B	Nucleus sampling	39.00	26.00	5.00	0.48	38.00	23.00	7.00	0.40
	USC	41.00	27.00	5.00	0.47	54.00	39.00	**10.00**	0.59
	Likelihood	33.00	18.00	2.00	0.37	42.00	30.00	8.00	0.45
	Random Beam	36.00	21.25	7.00	0.40	44.50	23.50	5.75	0.45
	DeepConf	36.00	20.00	5.00	0.37	46.00	24.00	6.00	0.47
	ProcessKernel (Ours)	**66.00**	**43.00**	**17.00**	**0.70**	**70.00**	**57.00**	**10.00**	**0.78**
DR.Kernel-14B	Nucleus sampling	35.00	25.00	6.00	0.48	28.00	20.00	7.00	0.53
	USC	34.00	24.00	6.00	0.48	37.00	24.00	7.00	0.41
	Likelihood	31.00	20.00	5.00	0.46	33.00	22.00	4.00	0.61
	Random Beam	33.25	21.25	4.00	0.49	28.75	17.50	2.50	0.32
	DeepConf	38.00	24.00	5.00	0.49	40.00	27.00	7.00	0.59
	ProcessKernel (Ours)	**41.00**	**30.00**	**11.00**	**0.56**	**62.00**	**48.00**	**11.00**	**0.68**
Muse-Glimmer-30B	Nucleus sampling	40.00	17.00	6.00	0.43	49.00	31.00	9.00	0.72
	USC	39.00	15.00	7.00	0.55	50.00	36.00	10.00	0.72
	Likelihood	31.00	14.00	4.00	0.35	36.00	27.00	7.00	0.57
	Random Beam	45.75	19.50	9.25	0.63	43.75	30.50	10.25	0.75
	DeepConf	36.00	15.00	3.00	0.38	47.00	31.00	7.00	0.77
	ProcessKernel (Ours)	**72.00**	**33.00**	**16.00**	**0.77**	**74.00**	**51.00**	**14.00**	**0.83**
Qwen3.6-35B	Nucleus sampling	22.00	12.00	3.00	**0.52**	13.00	6.00	1.00	0.14
	USC	26.00	20.00	5.00	0.39	12.00	8.00	0.00	0.15
	Likelihood	17.00	10.00	0.00	0.21	13.00	7.00	0.00	0.14
	Random Beam	24.25	12.75	2.25	0.25	8.25	5.00	0.00	0.10
	DeepConf	25.00	14.00	5.00	0.29	7.00	6.00	0.00	0.10
	ProcessKernel (Ours)	**32.00**	**26.00**	**9.00**	0.48	**36.00**	**31.00**	**6.00**	**0.65**
GPT-OSS-120B	Nucleus sampling	23.00	20.00	3.00	0.30	23.00	18.00	3.00	0.27
	USC	28.00	22.00	6.00	0.36	27.00	20.00	1.00	0.33
	Likelihood	20.00	18.00	3.00	0.27	15.00	10.00	0.00	0.18
	Random Beam	21.52	15.15	3.56	0.28	16.42	13.09	1.86	0.21
	DeepConf	19.00	17.00	3.00	0.27	19.00	15.00	2.00	0.26
	ProcessKernel (Ours)	**45.00**	**29.00**	**12.00**	**0.60**	**58.00**	**49.00**	**11.00**	**0.70**
Qwen3.5-122B	Nucleus sampling	22.00	13.00	2.00	0.26	17.00	11.00	2.00	0.18
	USC	26.00	14.00	2.00	0.29	20.00	16.00	4.00	0.25
	Likelihood	18.00	8.00	1.00	0.20	15.00	12.00	3.00	0.18
	Random Beam	23.95	11.55	1.24	0.25	18.17	10.65	1.89	0.21
	DeepConf	28.00	15.00	2.00	0.33	14.00	8.00	3.00	0.14
	ProcessKernel (Ours)	**38.00**	**25.00**	**6.00**	**0.46**	**32.00**	**23.00**	**13.00**	**0.34**
DeepSeek-V4-Flash	Nucleus sampling	25.00	21.00	4.00	0.37	33.00	23.00	4.00	0.41
	USC	30.00	22.00	8.00	0.39	42.00	33.00	2.00	0.52
	Likelihood	31.00	20.00	3.00	0.39	30.00	24.00	4.00	0.41
	Random Beam	30.83	22.55	4.98	0.40	32.61	22.23	3.44	0.62
	DeepConf	30.00	20.00	4.00	0.39	24.00	20.00	2.00	0.34
	ProcessKernel (Ours)	**57.00**	**32.00**	**12.00**	**0.66**	**71.00**	**56.00**	**12.00**	**1.02**

4.3 Fidelity of the Reward Models

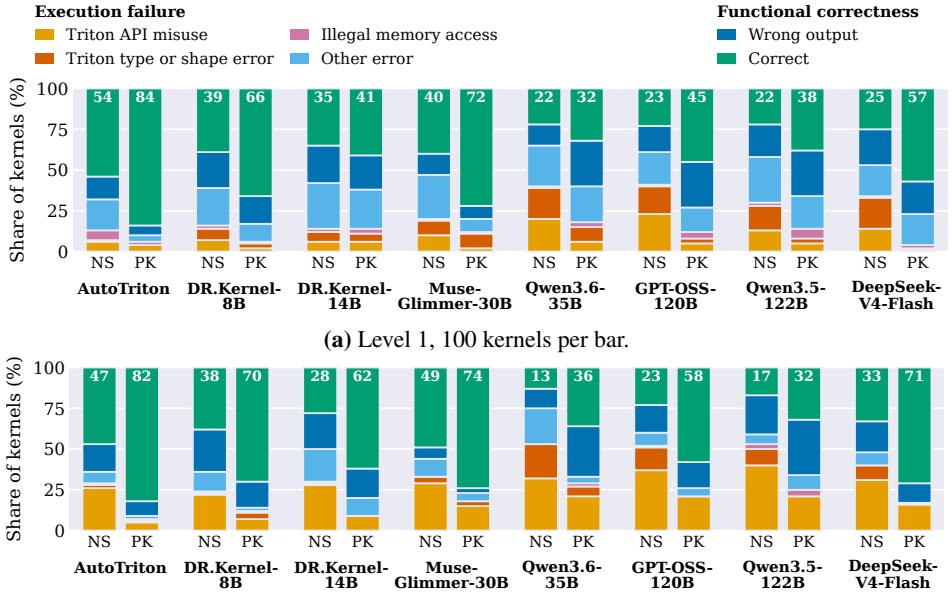
The PRM and ORM provide effective and complementary guidance across the search pipeline. On held-out `KernelBench` Level 1 examples, the PRM's top-ranked prefix leads to a correct kernel on 76.9% of lists, while the ORM ranks a correct kernel above an incorrect one on 92.3% of correctness pairs (See Table 12 in Appendix).These signals translate into stronger search outcomes in Figure 3, which separates the quality of the candidate pool from the quality of final selection. We define Oracle correctness as the fraction of problems for which a candidate pool contains at least one correct kernel, i.e., the best correctness achievable if the strongest candidate in each pool could be selected perfectly. Compared with random beam search, PRM-guided search improves this oracle correctness by up to 24.5% across generators, showing that process rewards produce substantially stronger candidate pools. The ORM then recovers most of this available quality without execution: its selected-kernel correctness remains within 8.5% of the corresponding Oracle, and within 4% for most generators. Together, these results show the complementary roles of the two models: the PRM improves which candidates are generated, while the ORM determines which completed candidate to return.

Execution-free supervision reduces labeling cost by 76×. We compare the cost of execution-based grading with ORM scoring on the same continuations. Executing one continuation takes 5.9 GPU-seconds on the measured subset, corresponding to roughly 707 GPU-hours for the 430,890



(a) RANDOM BEAM **(b)** PRM-GUIDED BEAM

Figure 3. Selector correctness on candidate pools from RANDOM BEAM and PRM-GUIDED BEAM. Each row covers the 200 problems of `KernelBench` levels 1 and 2 for one generator. Labels report the percentage of problems for which the selected kernel is correct. RANDOM denotes the expected correctness of uniform selection from the pool, ORM selects without execution, and ORACLE selects using measured outcomes and thus represents the pool upper bound. In (b), the ORM selection is the kernel returned by PRM-GUIDED BEAM.



(a) Level 1, 100 kernels per bar.

(b) Level 2, 100 kernels per bar.

Figure 4. Failure modes of generated kernels. Each bar partitions the single kernel returned per generator–problem pair into execution failures, wrong outputs, and correct kernels, following `KernelBench` error categories. *NS* (*Nucleus sampling*) is the generator's first completion and *PK* (*ProcessKernel*) the ORM-selected kernel from GUIDED SEARCH. *Other error* includes syntax errors, timeouts, and cases with no returned kernel.

continuations underlying our prefix corpus. In contrast, ORM scoring requires 9.3 GPU-hours in total, yielding an approximately 76× reduction in continuation-labeling cost. Since both approaches incur the same continuation-generation cost, the saving comes specifically from replacing execution-based grading with learned scoring.

4.4 ANALYSIS OF GENERATED KERNELS

Process guidance consistently shifts generation from failed kernels to correct implementations. Figure 4 compares the single kernel returned by nucleus sampling (NS) with the kernel selected from GUIDED SEARCH (PK). Across all eight generators and both `KernelBench` levels, guided search increases the fraction of correct kernels, with gains of 6%–32% on Level 1 and 15%–38% on Level 2. These improvements come largely from reducing execution failures—including `Triton` API misuse, type/shape errors, illegal memory accesses, and other runtime failures—as well as wrong-output

cases. This indicates that process guidance helps steer generation away from partial programs that are likely to lead to invalid launch configurations, incompatible tensor shapes or dtypes, unsafe memory accesses, or semantically incorrect implementations before those failures materialize in the completed kernel.

Further analyses (Appendix C.4 -C.6) suggest that the gains are not driven only by avoiding invalid code, but also by better high-level implementation choices among correct kernels. Guided search more often selects favorable operator placement, fusion boundaries, and parallel decomposition, while avoiding pathological launch granularity and unnecessary host-side memory movement. Specifically, guided search tends to favor qualitatively different execution strategies rather than merely local code edits. For instance, in the cumulative-sum case shown in Appendix Figure 20, direct generation uses a custom row-wise scan with serialized work, whereas search discovers a library-core plus custom-tail design that combines `torch.cumsum` with a lightweight Triton kernel, reducing runtime from 86.6 ms to 8.93 ms ($9.70\times$). More broadly, these cases suggest that the gains come from better operator placement, fusion, and parallelization choices among already-correct implementations, rather than merely the local syntax.

5 Discussions & Limitations

Execution-free guidance and scalability. Our method builds on the premise that much of kernel quality can be inferred from program structure before execution. Although exact runtime remains hardware-dependent, key determinants of efficiency, including operator choice, fusion strategy, parallel decomposition, launch granularity, memory movement, and API usage, are already reflected in the code. This makes learned reward models a practical surrogate for repeated execution during search: once trained on measured correctness and speed, they can score both partial and completed kernels using only model inference.

More broadly, this fits the growing view that strong LLMs already possess substantial latent capability that can be better exploited at inference time through search and learned feedback. Rather than changing the generator itself, a reward model can guide how inference compute is allocated, steering generation toward regions of the model's output space that are more likely to contain high-quality solutions. In our setting, this allows generation to concentrate compute on promising kernel trajectories without repeatedly compiling and benchmarking a great amount of possible candidates. Moreover, since the reward models are decoupled from the generator, the same guidance can also be reused across different generator backbones without modifying or retraining them, making reward-guided inference a practical and scalable complement to further generator training.

Limitations and future directions. The effectiveness of execution-free search naturally depends on how accurately the learned reward models capture kernel quality. Our results are particularly encouraging for correctness, while fine-grained runtime differences, especially among already efficient kernels, leave more room for improvement. Stronger speed-sensitive supervision, richer hardware-aware representations, and larger or more diverse calibration sets could further improve this signal. Future work could also study transfer across GPU architectures, kernel languages, and generator families, potentially reducing the amount of new execution data required when moving to a different deployment setting.

6 Conclusion

We introduced an execution-free process supervision framework for GPU-kernel generation that amortizes a limited set of measured executions into reusable guidance for both partial and completed kernels. By combining ranking-based outcome modeling, calibrated continuation-value imputation, and process-guided search, the framework reduces reliance on repeated compilation and execution while improving the quality of generated kernels. Extensive experiments show substantially lower supervision cost together with stronger candidate pools, higher correctness, and improved efficiency. More broadly, these results suggest that expensive terminal feedback can be distilled into reusable process-level signals, offering a promising path toward more scalable search in GPU-kernel generation and other domains where direct evaluation is costly and only available after completion.

REPRODUCIBILITV STATEMENT

The replication package can be found here: https://anonymous.4open.science/r/ProcessKernel-733C/. Detailed hyperparameters and implementation settings are provided in Appendix B.1.

AI USE STATEMENT

Generative AI tools were used to assist with language editing and polishing, as well as the retrieval and summarization of related work. All cited literature was independently read and verified by the authors before inclusion in the paper. We also used Codex and Claude Code to assist with implementing scripts for visualizing experimental results. AI-assisted code was reviewed and verified by the authors, and the core implementation of the research methodology was independently reviewed by the authors. The authors take full responsibility for the final manuscript, code, results, and conclusions.

REFERENCES

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, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, 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*, pp. 929–947. Association for Computing Machinery, 2024. doi: 10.1145/3620665.3640366. URL https://doi.org/10.1145/3620665.3640366.

Miriam Ayer, H. D. Brunk, G. M. Ewing, W. T. Reid, and Edward Silverman. An empirical distribution function for sampling with incomplete information. *The Annals of Mathematical Statistics*, 26(4): 641–647, 1955.

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.

Elena Bruches, Daniil Grebenkin, Mikhail Klementev, Vadim Alperovich, Roman Derunets, Dari Baturova, Georgy Mkrtchyan, Oleg Sedukhin, Ivan Bondarenko, Nikolay Bushkov, and Stanislav Moiseev. Rm-rf: Reward model for run-free unit test evaluation, 2026. URL https://arxiv.org/abs/2601.13097.

Christopher J. C. Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Gregory N. Hullender. Learning to Rank using Gradient Descent. In *Proceedings of the 22nd International Conference on Machine Learning*, pp. 89–96, 2005.

Christopher J. C. Burges, Robert Ragno, and Quoc V. Le. Learning to Rank with Nonsmooth Cost Functions. In *Advances in Neural Information Processing Systems 19*, pp. 193–200. MIT Press, 2006.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594. USENIX Association, 2018. URL https://www.usenix.org/conference/osdi18/presentation/chen.

Xinyun Chen, Renat Aksitov, Uri Alon, Jie Ren, Kefan Xiao, Pengcheng Yin, Sushant Prakash, Charles Sutton, Xuezhi Wang, and Denny Zhou. Universal Self-Consistency for Large Language Model Generation. arXiv:2311.17311, 2023. URL https://arxiv.org/abs/2311.17311.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026.

Kris Shengjun Dong, Sahil Modi, Dima Nikiforov, Sana Damani, Edward Lin, Siva Kumar Sastry Hari, and Christos Kozyrakis. Kernelblaster: Continual cross-task cuda optimization via memory-augmented in-context reinforcement learning, 2026. URL https://arxiv.org/abs/2602.14293.

He Du, Qiming Ge, Jiakai Hu, Aijun Yang, Zheng Cai, Zixian Huang, Sheng Yuan, Qinxiu Cheng, Xinchen Xie, Yicheng Chen, Yining Li, Jiaxing Xie, Huanan Dong, Yaguang Wu, Xiangjun Huang, Jian Yang, Hui Wang, Bowen Zhou, Bowen Li, Qipeng Guo, and Kai Chen. Kernel-smith: A unified recipe for evolutionary kernel optimization, 2026. URL https://arxiv.org/abs/2603.28342.

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

Yichao Fu, Xuewei Wang, Yuandong Tian, and Jiawei Zhao. Deep Think with Confidence. arXiv:2508.15260, 2025. URL https://arxiv.org/abs/2508.15260.

Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pp. 152–163. Association for Computing Machinery, 2009. doi: 10.1145/1555754.1555775. URL https://doi.org/10.1145/1555754.1555775.

Pin-Lun Hsu, Yun Dai, Vignesh Kothapalli, Qingquan Song, Shao Tang, Siyu Zhu, Steven Shimizu, Shivam Sahni, Haowen Ning, and Yanning Chen. Liger kernel: Efficient Triton kernels for LLM training, 2024. URL https://arxiv.org/abs/2410.10989.

Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Transactions on Information Systems (TOIS)*, 20(4):422–446, 2002.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626. Association for Computing Machinery, 2023. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Qingyao Li, Xinyi Dai, Xiangyang Li, Weinan Zhang, Yasheng Wang, Ruiming Tang, and Yong Yu. CodePRM: Execution Feedback-enhanced Process Reward Model for Code Generation. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 8169–8182, Vienna, Austria, July 2025a. Association for Computational Linguistics. doi: 10.18653/v1/2025.findings-acl.428. URL https://aclanthology.org/2025.findings-acl.428/.

Shangzhan Li, Zefan Wang, Ye He, Yuxuan Li, Qi Shi, Jianling Li, Yonggang Hu, Wanxiang Che, Xu Han, Zhiyuan Liu, and Maosong Sun. AutoTriton: Automatic Triton programming with reinforcement learning in LLMs, 2025b. URL https://arxiv.org/abs/2507.05687.

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, Liyuan Li, Nathan Yan, Xiayu Yu, Zhou Fang, Roman Levenstein, Kunming Ho, Haishan Zhu, Alec Hammond, Richard Li, Ajit Mathews, Kaustubh Gondkar, 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, 2026. URL https://arxiv.org/abs/2512.23236.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's Verify Step by Step. arXiv:2305.20050, 2023. URL https://arxiv.org/abs/2305.20050.

Wei Liu, Jiawei Xu, Yingru Li, Longtao Zheng, Tianjian Li, Qian Liu, and Junxian He. Dr. kernel: Reinforcement learning done right for triton kernel generations, 2026. URL https://arxiv.org/abs/2602.05885.

Liangchen Luo, Yinxiao Liu, Rosanne Liu, Samrat Phatale, Meiqi Guo, Harsh Lara, Yunxuan Li, Lei Shu, Yun Zhu, Lei Meng, Jiao Sun, and Abhinav Rastogi. Improve Mathematical Reasoning in Language Models by Automated Process Supervision. arXiv:2406.06592, 2024. URL https://arxiv.org/abs/2406.06592.

MiniMax. MiniMax-M2.7. https://huggingface.co/MiniMaxAI/MiniMax-M2.7, 2026. Hugging Face Model Repository.

OpenAI. gpt-oss-120b & gpt-oss-20b Model Card, 2025. URL https://arxiv.org/abs/2508.10925.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*, 2025.

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

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Oona Rainio, Jarmo Teuho, and Riku Klén. Evaluation metrics and statistical tests for machine learning. *Scientific reports*, 14(1):6086, 2024.

Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 73–82. Association for Computing Machinery, 2008. doi: 10.1145/1345206.1345220. URL https://doi.org/10.1145/1345206.1345220.

Ali Tehrani, Yahya Emara, Essam Wissam, Wojciech Paluch, Waleed Atallah, Łukasz Dudziak, and Mohamed S. Abdelfattah. Fine-tuning GPT-5 for GPU kernel generation, 2026. URL https://arxiv.org/abs/2602.11000.

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*, pp. 10–19. Association for Computing Machinery, 2019. doi: 10.1145/3315508.3329973. URL https://doi.org/10.1145/3315508.3329973.

Peiyi Wang, Lei Li, Zhihong Shao, R. X. Xu, Damai Dai, Yifei Li, Deli Chen, Y. Wu, and Zhifang Sui. Math-Shepherd: Verify and Reinforce LLMs Step-by-step without Human Annotations. arXiv:2312.08935, 2024. URL https://arxiv.org/abs/2312.08935.

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.

Yufan Ye, Ting Zhang, Wenbin Jiang, and Hua Huang. Process-supervised reinforcement learning for code generation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 14213–14226, Suzhou, China, November 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.emnlp-main.719. URL https://aclanthology.org/2025.emnlp-main.719/.

Bianca Zadrozny and Charles Elkan. Transforming classifier scores into accurate multiclass probability estimates. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 694–699, 2002.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*, 2025.

Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 863–879. USENIX Association, 2020. URL https://www.usenix.org/conference/osdi20/presentation/zheng.

Appendix

These supplementary materials include the additional details of the method (§A), the implementation details (§B), and additional results (§C). The implementation can be found here: https://anonymous.4open.science/r/ProcessKernel-733C/.

A Additional Details of the Method

A.1 Notation

Table 2 summarizes the notation used throughout the paper.

Table 2. Summary of Notation.

Symbol	Meaning
$\boldsymbol{x}$	problem: a high-level reference program and its evaluation inputs
$f_{\boldsymbol{\theta}}$	generator: an LLM that maps $\boldsymbol{x}$ to a complete GPU kernel
$\boldsymbol{y}$	generated candidate, consisting of a reasoning trace and a kernel implementation
q	speedup over the reference implementation, mapped to a discretized log scale in $[0, 1]$
t	truncation index within a generation
$\boldsymbol{y}_{<t}$	prefix obtained by truncating $\boldsymbol{y}$ at t before the kernel is complete
K	number of continuations sampled from a prefix
$\boldsymbol{y}^{(k)}$	k-th completion sampled from $f_{\boldsymbol{\theta}}(\cdot \mid \boldsymbol{x}, \boldsymbol{y}_{<t})$
(i, j)	within-problem ranking pair with measured grades $v_i > v_j$
Δ_{ij}	change in NDCG induced by swapping pair (i, j)
$\mathcal{L}$	ranking loss, with $\mathcal{L}_{\text{corr}}$ and $\mathcal{L}_{\text{speed}}$ for correctness and speed pairs
g	monotone calibration map from ORM score to measured terminal quality, fitted from anchors
$c_{\boldsymbol{x}}$	problem-specific calibration offset applied to the ORM score before $\hat{g}$
α	relative weight of the correctness term in $\mathcal{L}$
$v(\boldsymbol{x}, \boldsymbol{y})$	measured terminal quality of $\boldsymbol{y}$ on $\boldsymbol{x}$: 0 if the kernel fails to compile or is incorrect, otherwise $(1 + q)/2$
$s(\boldsymbol{x}, \boldsymbol{y})$	ORM score for a completed candidate, trained to preserve the within-problem ordering induced by v
$\bar{v}(\boldsymbol{x}, \boldsymbol{y}_{<t})$	true prefix value: expected terminal quality v over completions continued from $\boldsymbol{y}_{<t}$
$\hat{v}(\boldsymbol{x}, \boldsymbol{y}_{<t})$	measured prefix value: Monte Carlo estimate of $\bar{v}$ from K executed continuations
$\tilde{v}(\boldsymbol{x}, \boldsymbol{y}_{<t})$	imputed prefix value: mean calibrated ORM value over K sampled continuations, used as the PRM training target
$r(\boldsymbol{x}, \boldsymbol{y}_{<t})$	PRM score for a prefix, trained to preserve the ordering induced by $\tilde{v}$

We distinguish four levels of supervision. For a complete kernel $\boldsymbol{y}$ on problem $\boldsymbol{x}$, $v(\boldsymbol{x}, \boldsymbol{y})$ denotes its measured terminal quality. The ORM does not regress this value directly; instead, it outputs a score $s(\boldsymbol{x}, \boldsymbol{y})$ trained to preserve the within-problem ordering induced by v:

$$v(\boldsymbol{x}, \boldsymbol{y}) \xrightarrow{\text{ranking supervision}} s(\boldsymbol{x}, \boldsymbol{y}).$$

For a prefix $\boldsymbol{y}_{<t}$, the ideal target is its true continuation value,

$$\bar{v}(\boldsymbol{x}, \boldsymbol{y}_{<t}) = \mathbb{E}_{\boldsymbol{y}' \sim f_{\boldsymbol{\theta}}(\cdot \mid \boldsymbol{x}, \boldsymbol{y}_{<t})}\big[v(\boldsymbol{x}, \boldsymbol{y}')\big],$$

which can be estimated by executing K sampled continuations:

$$\hat{v}(\boldsymbol{x}, \boldsymbol{y}_{<t}) = \frac{1}{K} \sum_{k=1}^{K} v(\boldsymbol{x}, \boldsymbol{y}^{(k)}).$$

Because computing $\hat{v}$ at scale requires executing every continuation, we instead calibrate the ORM score with g and a problem-specific offset $c_{\boldsymbol{x}}$ to obtain the imputed prefix value

$$\tilde{v}(\boldsymbol{x}, \boldsymbol{y}_{<t}) = \frac{1}{K} \sum_{k=1}^{K} g\Big(s(\boldsymbol{x}, \boldsymbol{y}^{(k)}) + c_{\boldsymbol{x}}\Big).$$

The PRM then outputs $r(\boldsymbol{x}, \boldsymbol{y}_{<t})$ and is trained to preserve the ordering induced by $\tilde{v}$:

$$v \;\longrightarrow\; s \;\longrightarrow\; \tilde{v} \;\longrightarrow\; r, \qquad \bar{v} \approx \hat{v}, \qquad \tilde{v} \approx \bar{v}.$$

A.2 Objective Details

Normalized Discounted Cumulative Gain (NDCG). NDCG measures how closely a predicted ranking agrees with the ideal ordering induced by ground-truth grades. For a ranked list of n candidates with relevance values $\rho_1, \ldots, \rho_n$, the discounted cumulative gain is

$$\mathrm{DCG} = \sum_{k=1}^{n} \frac{\phi(\rho_k)}{\log_2(k+1)},$$

where $\phi(\rho) = 2^{\rho} - 1$ is the gain transformation. The logarithmic discount gives greater importance to candidates near the top of the ranking. Let $\rho_1^* \geq \cdots \geq \rho_n^*$ denote the same candidates sorted by their ground-truth relevance. The corresponding ideal discounted cumulative gain is

$$\mathrm{IDCG} = \sum_{k=1}^{n} \frac{\phi(\rho_k^*)}{\log_2(k+1)},$$

and

$$\mathrm{NDCG} = \frac{\mathrm{DCG}}{\mathrm{IDCG}}.$$

Higher NDCG therefore indicates closer agreement with the grade-induced ideal ranking. In LambdaRank, a pair (i, j) is weighted by Δ_{ij}, the change in NDCG that would result from swapping the predicted positions of candidates i and j. Misorderings that more strongly affect the top of the ranked list consequently receive greater training weight.

Isotonic Calibration. The ORM is trained only to preserve within-problem order, so its score $s(\boldsymbol{x}, \boldsymbol{y})$ need not lie on the same numerical scale as the measured terminal quality $v(\boldsymbol{x}, \boldsymbol{y})$. We recover an approximate value scale from executed anchors, for which both quantities are available. Given anchor pairs (s_i, v_i), we fit a monotone calibration map

$$g : s \mapsto v$$

using isotonic regression (Ayer et al., 1955; Zadrozny & Elkan, 2002). Isotonic regression is a nonparametric regression method that fits a piecewise-constant, nondecreasing function to the observations. In our setting, it solves

$$g = \arg \min_{h:\, \text{nondecreasing}} \sum_{i} \big(h(s_i) - v_i\big)^2,$$

so the calibrated values remain close to the measured anchor grades while preserving the assumption that a higher ORM score should not map to a lower quality value. Unlike a linear or logistic calibration, isotonic regression does not impose a particular functional relationship between score and value, which is desirable because the ORM is trained for ranking rather than numerical calibration. We first aggregate the anchors into 40 equal-count score bands and fit g over these bands subject to the monotonicity constraint.

Problem-Specific Offset. Because the ORM is identified only through within-problem rankings, its scores may contain a problem-specific additive shift. We account for this with an offset $c_{\boldsymbol{x}}$ before applying the shared calibration map. For the anchors $\{(s_i, v_i)\}_{i=1}^{m_{\boldsymbol{x}}}$ of problem $\boldsymbol{x}$, we first obtain the raw offset $\hat{c}_{\boldsymbol{x}}$ by matching the total calibrated value to the measured total:

$$\sum_{i=1}^{m_{\boldsymbol{x}}} g(s_i + \hat{c}_{\boldsymbol{x}}) = \sum_{i=1}^{m_{\boldsymbol{x}}} v_i.$$

We solve this one-dimensional equation by bisection over $[-4, 4]$. Since problems with few or noisy anchors yield less reliable offsets, we shrink the raw estimate toward zero using empirical Bayes:

$$c_{\boldsymbol{x}} = \frac{\tau^2}{\tau^2 + \sigma_{\boldsymbol{x}}^2} \hat{c}_{\boldsymbol{x}},$$

where $\sigma_{\boldsymbol{x}}^2$ estimates the uncertainty of the problem-specific offset and τ^2 is the across-problem variance. Thus, uncertain offsets receive stronger shrinkage, whereas well-supported offsets remain close to their raw estimates.

We fit the shared isotonic map and the problem-specific offsets alternately, using damping of 0.5 for up to 16 iterations; τ^2 is estimated after the first pass and then held fixed. Problems without anchors use $c_{\boldsymbol{x}} = 0$, while problems whose anchors all have zero measured quality are excluded from calibration.

Finally, for a prefix $\boldsymbol{y}_{<t}$ with K sampled continuations $\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(K)}$, the imputed prefix value used to supervise the PRM is

$$\tilde{v}(\boldsymbol{x}, \boldsymbol{y}_{<t}) = \frac{1}{K} \sum_{k=1}^{K} g\Big(s(\boldsymbol{x}, \boldsymbol{y}^{(k)}) + c_{\boldsymbol{x}}\Big).$$

Hence, g converts the ORM's relative score into an approximate terminal value, while $c_{\boldsymbol{x}}$ removes the problem-specific score shift before the values are averaged across continuations.

B Implementation Details

B.1 Prompt, Parameters, and Grading Protocol

Generation Prompt. Every generator receives the same two messages, `KernelBench`'s own one-shot `Triton` prompt at the commit we pin, preceded by a system message of ours that asks for a plan before the code. The system message is:

```
You write custom kernels to replace the pytorch operators in the given
architecture to get speedups.

You have complete freedom to choose the set of operators you want to
replace. You may replace some operators with custom kernels and leave
others unchanged.

Before writing any code, first think through and lay out a plan. Identify
which operators are the most promising to replace, explain why, and
describe the kernel strategy you intend to use. Keep this planning section
concise.

After you have written out the plan, implement it. You need to provide the
complete Python code wrapped in a Python code block that starts with
```python and ends with ```.
```

The user message is rendered from `KernelBench`'s prompt template. Square brackets mark the two code blocks we elide for space, both of which are files in the `KernelBench` repository:

```
You write custom Triton kernels to replace the pytorch operators in the
given architecture to get speedups.

You have complete freedom to choose the set of operators you want to
replace. You may make the decision to replace some operators with custom
Triton kernels and leave others unchanged. You may replace multiple
operators with custom implementations, consider operator fusion
opportunities (combining multiple operators into a single kernel, for
example, combining matmul+relu), or algorithmic changes (such as online
softmax). You are only limited by your imagination.

Here's an example to show you the syntax of inline embedding custom Triton
kernels in PyTorch:

Example:

Input architecture:
```
```

```
[prompts/model_ex_add.py]

Optimized with Triton kernels:

[prompts/model_new_ex_add_triton.py]

You are given the following architecture:

[the problem's reference PyTorch module]

Note: The kernels should be optimized for FP32 (32-bit floating point)
precision.

Optimize the architecture named Model with custom Triton kernels! Name your
optimized output architecture ModelNew. Output the new code in codeblocks.
Please generate real code, NOT pseudocode, make sure the code compiles and
is fully functional. Just output the new model code, no other text, and NO
testing code!
```

Every kernel we report is generated from this prompt unchanged, except for the execution-feedback methods of Section C.2, which use their own prompts. The completions the PRM trains on were generated earlier, with two blocks appended to the same template: an input contract stating each argument's dtype and shape, and a note on `fp32` tolerance that warns that `tl.dot`'s default `TF32` fails the 10^{-4} correctness tolerance and observes that `cuBLAS` is near-optimal for `fp32` matrix multiplication. Continuations sampled during label collection replay the stored prompt of the completion they extend, so a prefix and its continuations always share one prompt.

Grading Protocol. `KernelBench` grades every kernel over five correctness trials against the reference module and one hundred timing trials, both at `fp32`, with a 300 s timeout per kernel. A kernel counts as correct only if all five trials match the reference within a tolerance of 10^{-4}.

Hyperparameters. Table 3 summarizes the training configuration of the ORM and PRM. Table 4 summarizes the decoding settings used for each generator. Each generator produces at most 22,784 new tokens within a 32,768-token context and is allowed to reason before answering inside a single call.Table 5 summarizes the hardware and software environment. All stages use seed 42. Table 6 summarizes the search, label-generation, and grading settings.

Table 3. Reward-model training configuration. Batch sizes are effective batch sizes obtained by gradient accumulation over single-list steps.

Reward model	ORM	PRM
Backbone	`Qwen3-Reranker-4B`	`Qwen3-Reranker-4B`
Training unit	complete kernel	prefix list
Kernels written by	`DeepSeek-V4-Flash`, `GPT-OSS-120B`	`DeepSeek-V4-Flash`, `MiniMax-M2.7`, `Qwen3.5-122B`
Candidates per list	16 (8 positive, 8 negative)	2–8
Epochs	5	5
Learning rate	1×10^{-5}	2×10^{-5}
Batch size (lists)	32	128
Warmup ratio	0.03	0.03
Weight decay	0.05	0.01
Context length (tokens)	6,144	24,576
α in loss $\mathcal{L}$	0.37	0.5
Checkpoint selected on	`fast_1.0`	NDCG

B.2 Baseline Methods

Existing inference-time kernel optimization methods are difficult to compare directly in our setting: they all are designed for `CUDA`, whereas `Triton` is the backend most commonly used by recent LLM-based kernel-generation systems. We therefore evaluate ProcessKernel in `Triton` and compare against controlled baselines that isolate where its gains come from—additional sampling, post-hoc selection, beam-style exploration, confidence-based guidance, or learned process supervision itself.

Table 4. Generator decoding configuration.

Generator	Temperature	Top-p	Top-k
`AutoTriton`	0.6	0.95	0
`DR.Kernel-8B`	1.0	1.0	0
`DR.Kernel-14B`	1.0	1.0	0
`Muse-Glimmer-30B`	1.0	0.95	64
`Qwen3.6-35B-A3B`	0.6	0.95	20
`GPT-OSS-120B`	1.0	1.0	0
`Qwen3.5-122B-A10B-FP8`	0.6	0.95	20
`DeepSeek-V4-Flash`	1.0	1.0	0
`MiniMax-M2.7` (PRM training only)	1.0	0.95	40

Table 5. Hardware and software for training, generation and grading. GPU counts are per job.

Reward-model training	
GPUs	4 × NVIDIA H100 80GB
Python	3.12.13
`PyTorch`	2.11.0 (CUDA 12.9)
`transformers`	5.14.1
Generation	
GPUs	2–4 × NVIDIA H100 80GB, 2–4 × NVIDIA A30 24GB or 1 × NVIDIA GH200
`vLLM`	0.26.0 (nucleus sampling, training corpora), 0.25.1 (search, label continuations), 0.29.0 (`Muse-Glimmer-30B`)
Grading	
GPU	1 × NVIDIA H100 80GB per kernel
Python	3.10.21
`PyTorch`	2.10.0 (CUDA 12.8)
`Triton`	3.6.0
Host CPUs and operating systems	
H100 nodes	2 × Intel Xeon 8480C, 2.00 GHz, Ubuntu 24.04.4 LTS
A30 node	2 × Intel Xeon 8468V, 2.40 GHz, Ubuntu 24.04.4 LTS
GH200 nodes	72-core NVIDIA Grace (Arm), RHEL 9

Table 6. Search, label-generation, and grading configuration.

Search	
Beam width B	4
Children per candidate N	4
Maximum steps S	89
Segment budget	256 tokens
Distinct parents among survivors	≥ 2
Labels	
Cuts per completion group D	4
Relative cut depth	0.1–0.9
Maximum completions cut per depth M	8
Continuations per prefix K	5
Minimum surviving continuations	3
Calibration score bands	40
Calibration iterations	16
Grading	
Correctness trials	5
Timing trials	100
Timeout per kernel	300 s
Baseline precision	`fp32`

Specifically, we adopt the following baseline methods: "**Nucleus sampling**" directly samples one complete kernel for each problem from the generator, with no branching, intermediate scoring, or

post-hoc selection. It therefore represents the standard one-shot generation setting and provides the reference point for measuring the benefit of additional search or selection. "**USC**" (Chen et al., 2023) draws four independent complete kernels and then asks the generator itself to select the most self-consistent candidate among them. In contrast, "**Likelihood**" uses the same four direct generations but selects the candidate with the highest average token log-probability under the generator. These two baselines keep generation fully terminal and differ only in how the completed candidates are ranked, testing whether the gains of our PROCESSKERNEL can be explained by additional sampling followed by lightweight post-hoc selection. "**Random Beam**" preserves the full search structure of PROCESSKERNEL, including the same segmentation, branching factor, beam width, and overall candidate budget, but removes learned guidance. At every intermediate pruning step, surviving prefixes are chosen uniformly at random, and the final returned kernel is also selected randomly from the completed pool. This controls for the effect of structured branching itself: any improvement over "Nucleus sampling" comes from exploring more trajectories, while the gap to PROCESSKERNEL isolates the contribution of the learned PRM and ORM. "**DeepConf**" (Fu et al., 2025) uses the same search skeleton as PROCESSKERNEL but replaces learned reward models with confidence signals computed directly from the generator's token probabilities. For intermediate prefixes, we use DeepConf's tail-confidence score to decide which branches survive; for completed kernels, we use its bottom-10% token confidence to select the final candidate. Because these scores require neither compilation nor execution, DeepConf provides a controlled execution-free baseline for testing whether generic model-confidence signals can substitute for task-specific process and outcome reward models.

Finally, in the execution-feedback setting of Appendix C.2, we additionally compare against "**CodePRM**" (Li et al., 2025a). CodePRM is a learned process scorer trained from code-generation trajectories with execution-derived supervision, providing a complementary comparison in which process-level guidance is explicitly grounded in program execution feedback. This allows us to test whether the execution-free supervision used by PROCESSKERNEL remains effective relative to a process reward model trained with substantially stronger execution-based signals.

Together, these baselines form a controlled progression from one-shot generation, to terminal post-hoc selection, to unguided structured search, to execution-free confidence-guided search, and finally to execution-supervised process guidance. This design isolates whether improvements come from more samples, search structure, generic confidence estimates, or the process supervision introduced by PROCESSKERNEL.

B.3 PATTERN LINTER

The linter of Section C.5 reads a kernel file without running it. It parses the file into a syntax tree and builds a model of the module from it. The model holds the `Triton` kernels the file defines, the launch sites in its host code, the launches that `forward` reaches and the buffers each launch reads and writes. Each check is a rule over this model. A kernel file shows a pattern when its check matches at least once. For the check on delegated heavy operations (F1.4), we count only calls to computationally expensive operators; for fusible intermediates between launches (F2.1), only tensors that the linter can prove lie between two fusible launches; and for repeated launches (F2.2), only launches occurring inside a host-side loop. Table 7 lists all thirteen checks with representative examples.

The checks form three families. Family 1 records how much of the work a file leaves to `PyTorch` or does not do at all. Family 2 records memory passes and launches that host code adds around the kernels. Its checks rest on the fact that `Triton` never fuses separate launches. A tensor that one launch writes and another reads therefore makes a full round trip through device memory. Family 3 looks inside the kernel body. Its checks key on facts of the syntax tree that hold for every input shape and every GPU.

The linter sees neither the input shapes nor the arguments a problem passes to the module. It therefore also counts a `PyTorch` call in a branch the problem never takes, such as a fallback for options the problem leaves unset. A pattern describes the code, so we judge speed by timing alone.

Table 7. The thirteen checks of the pattern linter. Each example abridges a test case from the linter's test suite to the lines the check matches.

ID	Pattern	What the linter looks for	Example
Family 1, work left to `PyTorch` or not done			
F1.1	No `Triton` kernel	The file defines no function decorated with `@triton.jit` (Li et al., 2025b).	`return torch.relu(x)`
F1.2	Kernel never launched	A kernel is defined, but no launch of it is reachable from `forward` (Tehrani et al., 2026; Liu et al., 2026).	`return x + y`
F1.3	Kernel output discarded	A launched kernel writes a buffer that never reaches the result (Tehrani et al., 2026).	`add_kernel[(1,)](x, y, out, n)` `return torch.add(x, y)`
F1.4	Heavy `PyTorch` call	`forward` calls a heavy `PyTorch` operation such as a convolution, matrix product, normalization or pooling (Li et al., 2025b; Woo et al., 2025; Liu et al., 2026).	`h = torch.conv2d(x, w)`
F1.5	`PyTorch` module call	`forward` calls a `PyTorch` module instead of only reading its weights (Li et al., 2025b; Woo et al., 2025).	`self.fc = nn.Linear(a, b)` `return self.fc(x)`
F1.6	Copy kernel	A launched kernel stores exactly the values it loads (Tehrani et al., 2026).	`v = tl.load(x_ptr + offs, mask=m)` `tl.store(out_ptr + offs, v, mask=m)`
F1.7	Compiler call	`forward` hands the computation to `torch.compile` or `TorchScript`.	`f = torch.compile(lambda t:  t * 2)`
Family 2, memory passes and launches added by host code			
F2.1	Intermediate between launches	A buffer written by one launch and read by a fusible next launch is never returned (Hsu et al., 2024).	`exp_kernel[g](x, tmp, n)` `scale_kernel[g](tmp, out, n)`
F2.2	Launch in a host loop	A kernel launch sits inside a Python loop of the host code (Liao et al., 2026).	`for i in range(x.shape[0]):` `exp_kernel[(1,)](x[i], out[i], n)`
F2.3	Copy in host code	Host code copies a tensor, for instance with `.contiguous()` after a permute, `.clone()` or `.to(dtype)` (Liao et al., 2026).	`xt = x.permute(1, 0).contiguous()`
F2.4	Zeroed buffer overwritten	A kernel overwrites a buffer from `torch.zeros` completely without reading it.	`out = torch.zeros_like(x)` `exp_kernel[g](x, out, n)`
Family 3, structure of the kernel body			
F3.1	One element per program	A launched kernel loads or stores memory but never builds a range with `tl.arange` (Ryoo et al., 2008; Hong & Kim, 2009; Tillet et al., 2019).	`pid_n = tl.program_id(0)` `acc += x_val * w_val`
F3.2	Loop bound at run time	A kernel that builds a range loops over `range(P)`, with `P` an ordinary argument rather than a `tl.constexpr` (Chen et al., 2018; Zheng et al., 2020).	`for kd in range(K):`

C Additional Results

C.1 Harder Problems

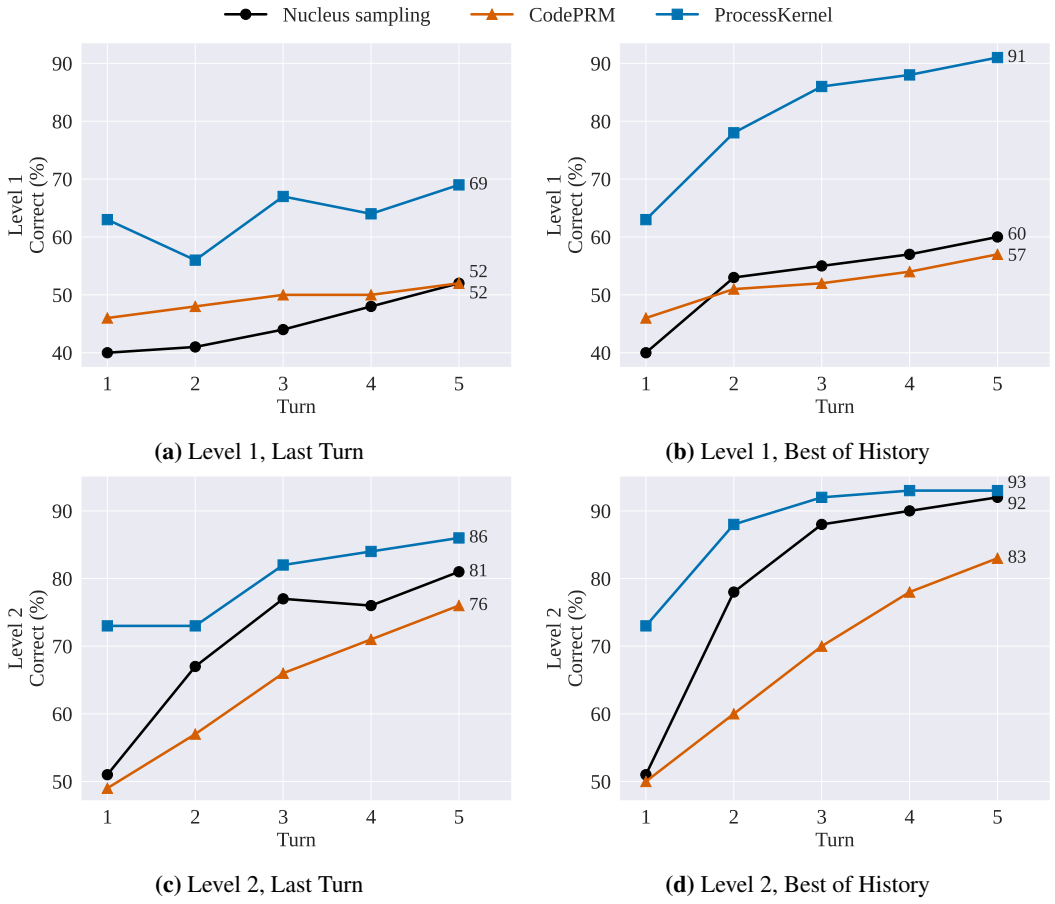
Table 8. Solved share by problem difficulty on `KernelBench` Levels 1 and 2. Results are aggregated over generator–problem pairs for the eight generators in Table 1. *Pairs* counts generator–problem pairs, *Problems* distinct problems, and Δ the percentage-point gain of *ProcessKernel* over *Nucleus sampling*.

Level	Difficulty	Pairs	Problems	Nucleus sampling	ProcessKernel	Δ	Random
1	hard	297	44	9.1	33.0	$+23.9$ $[18.8, 29.2]$	10.9
	medium	95	21	15.8	43.2	$+27.4$ $[16.5, 38.3]$	15.0
	easy	408	56	53.4	72.5	$+19.1$ $[13.2, 25.2]$	52.8
2	hard	187	29	7.5	40.6	$+33.2$ $[25.0, 41.8]$	8.3
	medium	97	22	19.6	59.8	$+40.2$ $[27.5, 53.6]$	24.7
	easy	516	71	41.7	68.0	$+26.4$ $[20.2, 32.5]$	38.7

To understand whether guided search helps mainly on problems that are already easy for a generator or also extends its reach to harder cases, we group generator–problem pairs by difficulty using the other seven generators as a reference. A pair is ***hard*** if at most two of the other generators produce a correct kernel in four completions each, ***medium*** if at most half of them do, and ***easy*** otherwise. Each method contributes one final kernel per pair: *Nucleus sampling* uses the generator's first completion, *ProcessKernel* the ORM-selected kernel from GUIDED SEARCH, and *Random* the expected solved share under uniform selection from the RANDOM BEAM pool. We report solved share and the improvement of *ProcessKernel* over *Nucleus sampling* with 95% bootstrap intervals over problems.

C.2 Search Under Execution Feedback

Execution-free search removes the need for online compilation and execution during generation, but practical kernel-optimization systems may still have access to execution feedback across refinement turns. We therefore ask whether ProcessKernel remains useful in this setting, or whether its benefits are subsumed once iterative feedback is available. To study this, we combine ProcessKernel with the multi-turn refinement loop of `Dr. Kernel` (Liu et al., 2026) and evaluate `DR.Kernel-8B` on `KernelBench` levels 1 and 2, comparing nucleus sampling with ProcessKernel-guided generation at every turn. We also compare with `CodePRM` (Li et al., 2025a) outside the harness, where we integrate `KernelGYM` into its search to provide feedback on every program. Figures 5–7 and Tables 9–10 report performance over five refinement turns.



(a) Level 1, Last Turn

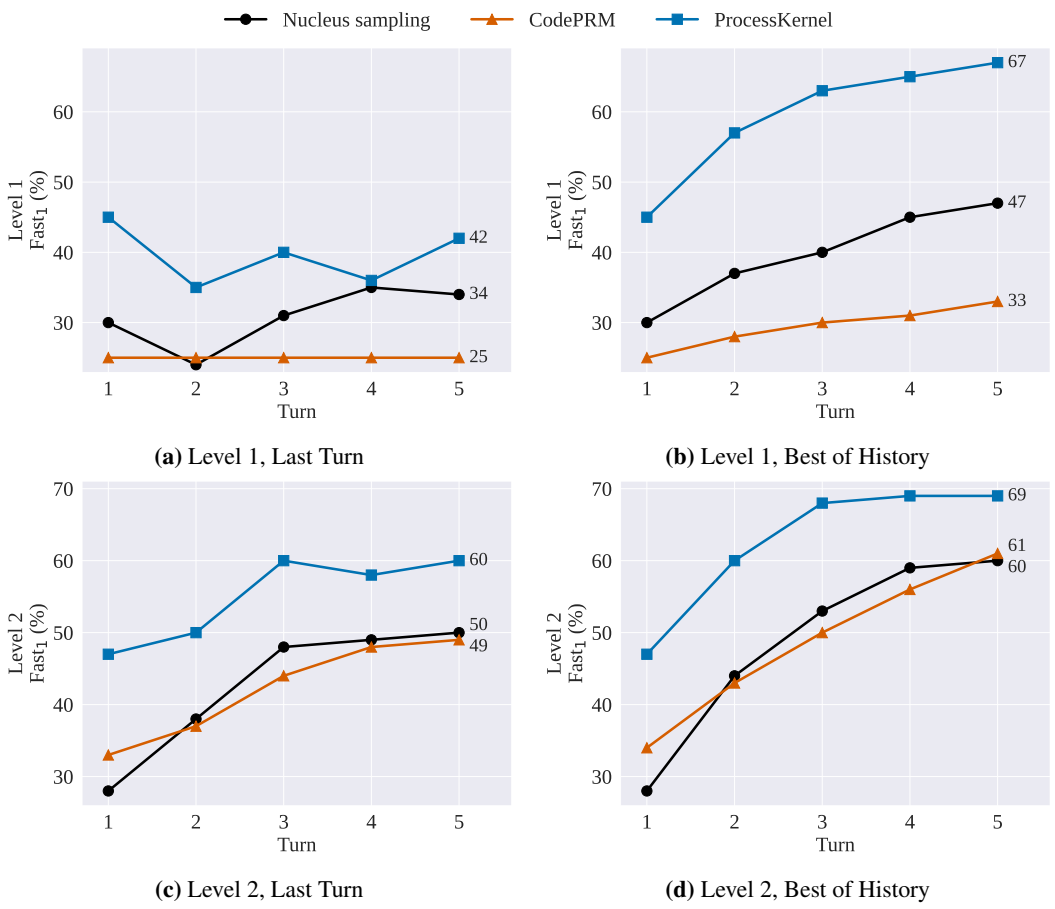
(b) Level 1, Best of History

(c) Level 2, Last Turn

(d) Level 2, Best of History

Figure 5. Correctness of `DR.Kernel-8B` over five turns of `Dr. Kernel`'s multi-turn harness, with nucleus sampling alone and with ProcessKernel. For `CodePRM`, which runs its own search framework with the same `DR.Kernel-8B` model and `KernelGYM` feedback, turn t is its rollout t. Rows are `KernelBench` levels 1 and 2, with 100 problems each. *Last Turn* scores the kernel a method returns after turn t. *Best of History* scores the best kernel of turns 1 to t. Numbers give the value at turn 5. Constant-output kernels and kernels that precompute weight sums or products outside the timed call count as incorrect (Section 4.1).

Setup. The harness keeps its prompts, feedback and turn loop unchanged. From turn 2 on, the model sees the whole conversation with `KernelGYM`'s verdict on its last kernel and is asked to improve it. Nucleus sampling is the model alone at the harness's settings, temperature 1.0 and top-p 0.95 with 8,192 new tokens per turn. ProcessKernel answers each turn with the ORM's pick from the Guided Search, run with $B = N = 4$ and at most $S = 32$ steps of 256 tokens. `CodePRM` runs its published tree search with the same model outside the harness, with five rollouts to match the five turns. It tests every program with `KernelGYM` and returns the one that passes the most correctness trials. Its process reward model scores the thought steps of every failing program to select those to refine and ranks the programs that pass all trials. We grade every turn's kernel with `KernelBench`



(a) Level 1, Last Turn

(b) Level 1, Best of History

(c) Level 2, Last Turn

(d) Level 2, Best of History

Figure 6. Fast$_1$ of `DR.Kernel-8B` over five turns, as in Figure 5.

on one H100 as in Table 1. Following Section 4.1, kernels that return a constant output without computing it and kernels that precompute weight sums or products outside the timed call count as incorrect.

A Lead at Every Turn. ProcessKernel achieves higher correctness than nucleus sampling at every turn on both levels (left column of Figures 5 to 7, Table 9). The lead is largest at turn 1, before any feedback, with 23.0 points on level 1 and 22.0 on level 2. Nucleus sampling needs three turns of feedback to reach ProcessKernel's first-turn correctness on level 2 and does not reach it within five turns on level 1 (52.0% against 63.0%). On level 2 the lead narrows to 5.0 points by turn 5, while on level 1 it stays at 15.0 to 23.0 points. `Fast`$_1$ leads most at turn 1, by 15.0 points on level 1 and 19.0 on level 2. No method reaches a mean speedup of 1.0 in the last turn. `CodePRM` stays below ProcessKernel at every turn on both levels in all three metrics. On level 1 its correctness lies between the two harness arms until turn 4 and ties with nucleus sampling at turn 5 (52.0%), while its `Fast`$_1$ stays at 25.0%. On level 2 it remains below nucleus sampling in correctness at every turn. ProcessKernel thus reaches in its first turn a correctness that feedback alone attains at the earliest after several turns.

Best of History. Keeping the best kernel of turns 1 to t widens the lead on level 1 (right column of Figures 5 to 7, Table 10). After five turns ProcessKernel reaches a correctness of 91.0%, a `Fast`$_1$ of 67.0% and a mean speedup of 1.24, against 60.0%, 47.0% and 0.94 for nucleus sampling. On level 2 both harness arms end near the ceiling in correctness (93.0% and 92.0%), with ProcessKernel ahead by 9.0 points in `Fast`$_1$. ProcessKernel's mean speedup passes 1.0 already at turn 3 on both levels, earlier than nucleus sampling on either level. `CodePRM`'s best program of five rollouts stays below both harness arms in correctness and mean speedup (57.0% and 0.74 on level 1, 83.0% and

Table 9. Correctness, Fast$_1$ and mean speedup of `DR.Kernel-8B` over five turns of `Dr. Kernel`'s multi-turn harness, with nucleus sampling alone and with ProcessKernel (*Last Turn* in Figures 5 to 7). Each row scores the kernel a method returns after turn t on the 100 problems of each `KernelBench` level, with the metrics of Table 1. For `CodePRM`, which runs its own search framework with the same `DR.Kernel-8B` model and `KernelGYM` feedback, turn t is its rollout t. Constant-output kernels and kernels that precompute weight sums or products outside the timed call count as incorrect (Section 4.1). Bold marks the best method.

		Level 1			Level 2		
Turn	Method	Correct ↑	Fast$_1$ ↑	Mean ↑	Correct ↑	Fast$_1$ ↑	Mean ↑
1	Nucleus sampling	40.00	30.00	0.48	51.00	28.00	0.55
	CodePRM	46.00	25.00	0.58	49.00	33.00	0.52
	ProcessKernel	**63.00**	**45.00**	**0.69**	**73.00**	**47.00**	**0.77**
2	Nucleus sampling	41.00	24.00	0.58	67.00	38.00	0.72
	CodePRM	48.00	25.00	0.58	57.00	37.00	0.59
	ProcessKernel	**56.00**	**35.00**	**0.68**	**73.00**	**50.00**	**0.83**
3	Nucleus sampling	44.00	31.00	0.67	77.00	48.00	0.87
	CodePRM	50.00	25.00	0.57	66.00	44.00	0.68
	ProcessKernel	**67.00**	**40.00**	**0.82**	**82.00**	**60.00**	**0.93**
4	Nucleus sampling	48.00	35.00	0.75	76.00	49.00	0.84
	CodePRM	50.00	25.00	0.59	71.00	48.00	0.72
	ProcessKernel	**64.00**	**36.00**	**0.86**	**84.00**	**58.00**	**0.95**
5	Nucleus sampling	52.00	34.00	0.77	81.00	50.00	0.89
	CodePRM	52.00	25.00	0.59	76.00	49.00	0.78
	ProcessKernel	**69.00**	**42.00**	**0.95**	**86.00**	**60.00**	**0.95**

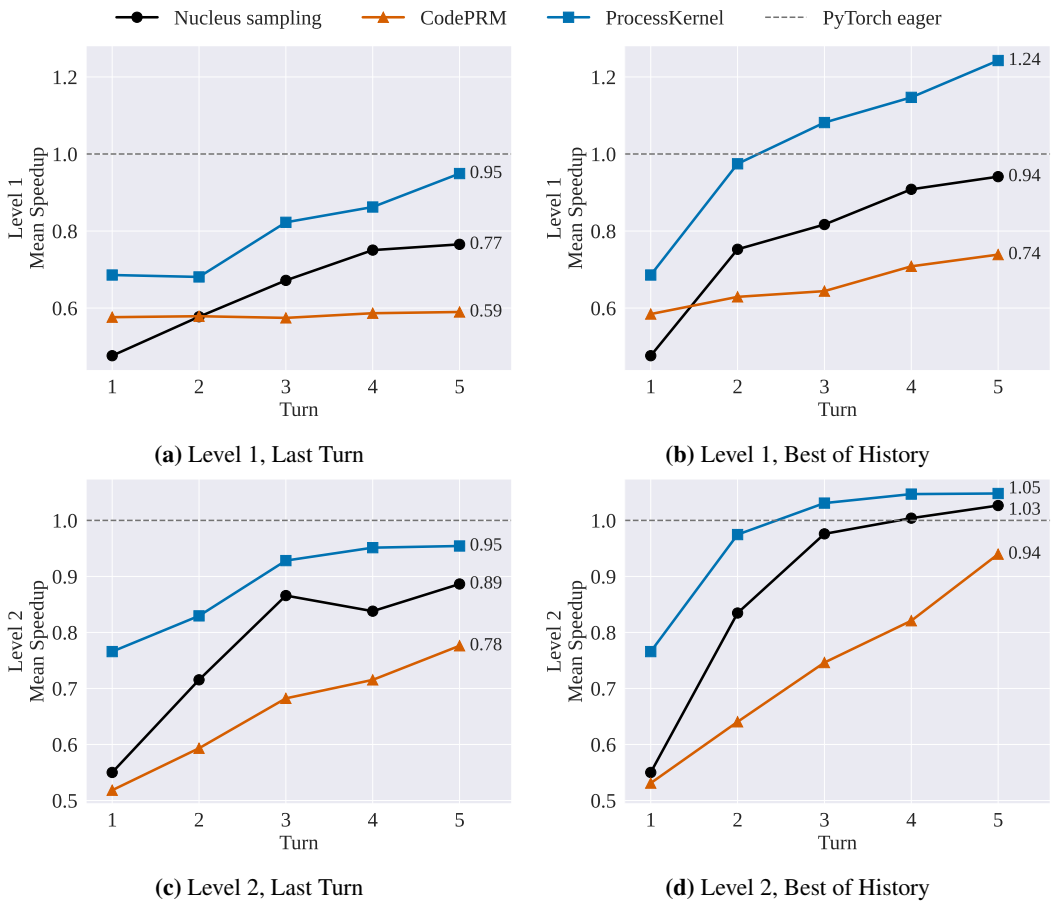
Table 10. The best kernel of turns 1 to t (*Best of History*) for the runs of Table 9.

		Level 1			Level 2		
Turn	Method	Correct ↑	Fast$_1$ ↑	Mean ↑	Correct ↑	Fast$_1$ ↑	Mean ↑
2	Nucleus sampling	53.00	37.00	0.75	78.00	44.00	0.83
	CodePRM	51.00	28.00	0.63	60.00	43.00	0.64
	ProcessKernel	**78.00**	**57.00**	**0.97**	**88.00**	**60.00**	**0.97**
3	Nucleus sampling	55.00	40.00	0.82	88.00	53.00	0.98
	CodePRM	52.00	30.00	0.64	70.00	50.00	0.75
	ProcessKernel	**86.00**	**63.00**	**1.08**	**92.00**	**68.00**	**1.03**
4	Nucleus sampling	57.00	45.00	0.91	90.00	59.00	1.00
	CodePRM	54.00	31.00	0.71	78.00	56.00	0.82
	ProcessKernel	**88.00**	**65.00**	**1.15**	**93.00**	**69.00**	**1.05**
5	Nucleus sampling	60.00	47.00	0.94	92.00	60.00	1.03
	CodePRM	57.00	33.00	0.74	83.00	61.00	0.94
	ProcessKernel	**91.00**	**67.00**	**1.24**	**93.00**	**69.00**	**1.05**

0.94 on level 2). Beyond correctness, ProcessKernel thus also makes its best kernels faster than eager `PyTorch` on average, and in fewer turns.

Cost and Scope. On one GH200, the five turns took ProcessKernel 3.6 and 3.4 hours on levels 1 and 2, against 0.8 and 0.5 hours for nucleus sampling. The extra time goes into generation and scoring alone. `CodePRM`'s five rollouts took 3.7 and 3.5 hours on three A30 GPUs, which are slower than a GH200, with `KernelGYM` on separate GPUs. It tested 7.5 and 8.1 programs per problem, against five kernels for each harness arm.

Although neither of our reward models was trained on conversations with execution feedback, ProcessKernel stays ahead of nucleus sampling and `CodePRM` in correctness, `Fast`$_1$ and mean speedup at every turn on both levels.



(a) Level 1, Last Turn

(b) Level 1, Best of History

(c) Level 2, Last Turn

(d) Level 2, Best of History

Figure 7. Mean speedup of `DR.Kernel-8B` over five turns, as in Figure 5. The dashed line marks parity with eager `PyTorch`.

C.3 Fidelity of the Imputed Labels

The PRM is trained only on imputed prefix values $\tilde{v}$, obtained by calibrating ORM scores in place of directly measured prefix values $\hat{v}$. Since the listwise objective depends on the relative ordering of prefixes within each list, we evaluate whether $\tilde{v}$ preserves the ordering induced by $\hat{v}$. We compare the two on a measured subset of 500 validation problems from our `KernelBook` conversion, for which all sampled continuations are graded with `KernelBench`. Because these problems are drawn from the same corpus as the training data, the resulting agreement measures the fidelity of the supervision used to train the PRM.

Setup. We evaluate the imputed values and both reward models against measured grades: v for completed kernels and $\hat{v}$ for prefixes, where zero denotes an incorrect kernel or a prefix with no correct continuation. Pairwise accuracy (Burges et al., 2005) is the fraction of unequal-grade pairs within a list for which the scorer assigns the higher score to the higher-graded item. We report pairwise accuracy over all pairs (**All**), correctness pairs with lower grade zero (**Corr**), and speed pairs with both grades positive (**Speed**). **Speed**$_{2\times}$ further restricts to speed pairs whose measured difference corresponds to at least a $2.1\times$ speedup. A pair is *resolved* if its measured gap exceeds the noise from five continuations at the 95% level. **NDCG** measures the quality of the full-list ordering. **Top-1** is the fraction of lists whose highest-ranked item has positive grade, while **Value@1** is that item's grade normalized by the best grade in the list. For imputed values, bias and mean absolute error (MAE; Rainio et al., 2024) measure deviation from the measured values. All reported intervals are 95% bootstrap intervals over problems.

For the reward-model evaluation, we use 13,942 finished kernels and 251 held-out prefix lists from the same rollout campaign. Prefix grades $\hat{v}$ are the mean measured grade over their continuations. As a training- and execution-free baseline, `DeepConf` ranks candidates using the generator's bottom-10% group confidence (Fu et al., 2025); we use 512-token windows because the median prefix is 879 tokens long. On finished kernels, the same score picks the final kernel of *DeepConf* in Table 1. Ranking metrics are computed only on lists containing items with different measured grades, retaining 54 of the 85 finished-kernel lists and all 251 prefix lists.

Table 11. Pairwise accuracy of the imputed values $\tilde{v}$ with the measured values $\hat{v}$ on the measured subset of the validation split. Each list is ordered by $\tilde{v}$ and graded by $\hat{v}$, the mean grade of five continuations. **Pairs** counts the pairs whose measured values differ. The last row orders each list at random. Columns are taken over the 1,811 of 2,038 lists whose prefixes differ in measured value.

	Pairs	All	Corr	Speed	Speed$_{2\times}$	NDCG	Top-1	Value@1
`DeepSeek-V4-Flash`	2,850	0.718	0.787	0.653	0.663	0.903	0.874	0.770
`MiniMax-M2.7`	1,611	0.691	0.752	0.642	0.655	0.907	0.864	0.769
`Qwen3.5-122B`	2,138	0.720	0.757	0.697	0.724	0.924	0.888	0.820
Pooled	6,599	0.712	0.770	0.666	0.679	0.911	0.876	0.786
Random order	–	0.500	0.500	0.500	0.500	0.822	0.718	0.585

Imputed Labels Preserve the Measured Order. We evaluate the agreement between the imputed prefix values $\tilde{v}$ and the measured values $\hat{v}$ using the same ranking metrics as for the reward models, treating $\hat{v}$ as ground truth. As Table 11 shows, $\tilde{v}$ agrees with $\hat{v}$ on 71.2% of all pairs, compared with 50.0% for a random ordering. Agreement is higher on correctness pairs at 77.0%, while speed pairs reach 66.6%. On the 2,379 resolved pairs, where the measured gap exceeds the noise of five continuations, agreement rises to 80.4% (77.8–83.0), with 81.1% on correctness pairs and 79.1% on speed pairs. This indicates that most disagreement on speed pairs comes from near-ties within measurement noise. The prefix ranked first by $\tilde{v}$ retains 78.6% of the best measured value in its list, compared with 58.5% for a random pick. Overall, the imputed labels recover the measured ordering on roughly four out of five resolvable pairs, which is the signal required by the listwise training objective.

Table 12. Ranking accuracy of the reward models on `KernelBench` Level 1. Models are evaluated against measured grades on 13,942 finished kernels and 251 held-out prefix lists. The upper block evaluates completed kernels and the lower block prefixes; random ordering provides a reference. **Bold** marks the better reward model on the finished kernels.

	Pairs	All	Corr	Speed	Speed$_{2\times}$	NDCG	Top-1	Value@1
Finished kernels								
ORM	271,911	**0.813**	**0.923**	0.512	**0.766**	**0.844**	**0.778**	**0.744**
PRM	271,911	0.712	0.766	**0.566**	0.539	0.777	0.593	0.573
`DeepConf`	271,911	0.466	0.461	0.479	0.515	0.564	0.333	0.317
Random order	–	0.500	0.500	0.500	0.500	0.584	0.308	0.292
Held-out prefixes								
PRM	1,024	**0.665**	**0.740**	**0.560**	**0.551**	**0.874**	**0.769**	**0.703**
`DeepConf`	1,024	0.462	0.419	0.522	0.540	0.753	0.534	0.485
Random order	–	0.500	0.500	0.500	0.500	0.780	0.581	0.519

Imputed Supervision Preserves Ordering More Than Scale. To characterize the signal seen by the PRM, we weight each pair exactly as in the loss objective, assuming the current ranking matches the imputed labels, and compare the resulting pair directions with measurement. Measured values agree with 59.8% of the total pair weight and disagree with 18.7%, while the remaining 21.5% falls on measured ties. Restricting to pairs whose measured gap exceeds measurement noise, 28.5% (26.3–30.8) of the weight supports the imputed ordering and only 5.0% (4.1–6.0) contradicts it; the rest lies within the noise range. Thus, the loss places roughly three times more weight on measurement-consistent than measurement-opposing directions.

The absolute scale is less faithful. Imputed values underestimate measured values by 0.09 on average, with an MAE of 0.21, and the gap grows to 0.28 on the highest-value quartile. The calibration map

saturates around 0.73–0.76 even though measured values reach 1.0, compressing the value gaps used to weight pairs. The imputed labels therefore preserve relative order more reliably than absolute scale, which is sufficient for the ranking-based search objective.

C.4 Per-Generator Breakdown of Kernel Patterns

Table 14 provides the per-generator breakdown underlying the aggregate code-pattern analysis in Table 13. For each generator, we compare its first completion with the ORM-selected kernel from GUIDED SEARCH, restricting the analysis to kernels graded correct by `KernelBench` on an H100. Each column contains at most one kernel per problem, and the final row reports the number of correct kernels included.

Run-time loop bounds are associated with slower kernels. To examine the F3.2 pattern more closely, we compare correct kernels from the same generator and problem that differ in whether a loop bound is supplied at run time or as a `tl.constexpr`. On Level 1, kernels with run-time bounds achieve a median of $0.68\times$ the speed of their compile-time counterparts and are slower in 8 of 10 matched pairs. Only one qualifying pair appears on Level 2. Because matched kernels may differ in other implementation details, this result indicates an association rather than a causal effect.

Table 13. Code patterns in the correct kernels the two methods ship for the eight generators of Table 1, as shares of those kernels (%). *Nucleus sampling* counts the generator's first completion where it is correct. *ProcessKernel* counts the candidate the ORM ranks first in the pool of the GUIDED SEARCH where it is correct. The Family 1 rows count kernels flagged by F1.4 or F1.5 of Table 7, for any heavy operation and for a convolution or matrix multiplication. The Family 2 row counts any of F2.1 to F2.4. The Family 3 rows are checks F3.1 and F3.2. Appendix C.4 gives each check per generator.

		Level 1		Level 2	
Family	Pattern	Nucleus sampling	ProcessKernel	Nucleus sampling	ProcessKernel
1	Leaves a heavy operation to `PyTorch`	24.2	40.0	98.0	99.0
	Leaves a convolution or matrix multiplication to `PyTorch`	15.0	29.9	97.6	99.0
2	Adds a memory pass or launch in host code	10.0	5.1	5.2	5.4
3	Computes one element per program	5.4	4.4	5.2	1.9
	Passes a loop bound at run time	1.9	6.2	2.4	2.5
Correct kernels		260	435	248	485

Table 14. Code-pattern frequencies among correct kernels on `KernelBench` Levels 1 and 2, reported as percentages. *NS* denotes *Nucleus sampling*, the generator's first completion, and *PK ProcessKernel*, the ORM-selected kernel from GUIDED SEARCH, restricted to cases where the selected kernel is correct. Generator names follow Table 1, and pattern definitions follow Table 7. Patterns F1.3 and F1.7 are omitted because they remain below 1% in every cell.

	AutoTriton		DR.Kernel-8B		DR.Kernel-14B		Muse-Glimmer-30B		Qwen3.6-35B		GPT-OSS-120B		Qwen3.5-122B		DeepSeek-V4-Flash	
Check	NS	PK	NS	PK	NS	PK	NS	PK	NS	PK	NS	PK	NS	PK	NS	PK
Level 1																
F1.1 no `Triton` kernel	0.0	14.3	2.6	13.6	0.0	0.0	2.5	19.4	0.0	3.1	0.0	0.0	0.0	0.0	0.0	1.8
F1.2 kernel never launched	0.0	0.0	15.4	18.2	2.9	4.9	2.5	5.6	0.0	0.0	0.0	0.0	27.3	7.9	0.0	1.8
F1.4 heavy `PyTorch` call	33.3	52.4	30.8	43.9	25.7	29.3	27.5	44.4	0.0	6.2	17.4	11.1	0.0	21.1	4.0	21.1
F1.5 `PyTorch` module call	35.2	41.7	23.1	43.9	11.4	24.4	25.0	31.9	0.0	0.0	4.3	8.9	0.0	21.1	0.0	19.3
F1.6 copy kernel	0.0	0.0	0.0	0.0	0.0	0.0	15.0	9.7	0.0	0.0	0.0	4.4	0.0	0.0	0.0	0.0
F2.1 intermediate between launches	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
F2.2 launch in a host loop	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	4.0	0.0
F2.3 copy in host code	1.9	2.4	10.3	6.1	45.7	17.1	7.5	6.9	0.0	0.0	0.0	4.4	4.5	0.0	0.0	3.5
F2.4 zeroed buffer overwritten	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
F3.1 one element per program	5.6	3.6	7.7	6.1	5.7	4.9	2.5	4.2	4.5	0.0	4.3	2.2	9.1	2.6	4.0	8.8
F3.2 loop bound at run time	0.0	0.0	0.0	0.0	0.0	0.0	2.5	1.4	0.0	12.5	4.3	20.0	4.5	7.9	8.0	17.5
Correct kernels	54	84	39	66	35	41	40	72	22	32	23	45	22	38	25	57
Level 2																
F1.1 no `Triton` kernel	0.0	1.2	0.0	4.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
F1.2 kernel never launched	2.1	2.4	18.4	7.1	0.0	3.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.1	3.0	0.0
F1.4 heavy `PyTorch` call	42.6	46.3	55.3	45.7	39.3	50.0	30.6	41.9	15.4	8.3	26.1	41.4	23.5	40.6	12.1	32.4
F1.5 `PyTorch` module call	100.0	100.0	100.0	100.0	89.3	93.5	98.0	100.0	92.3	91.7	95.7	98.3	100.0	100.0	97.0	97.2
F1.6 copy kernel	0.0	1.2	0.0	0.0	0.0	0.0	0.0	1.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
F2.1 intermediate between launches	0.0	0.0	2.6	1.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
F2.2 launch in a host loop	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
F2.3 copy in host code	2.1	3.7	10.5	5.7	17.9	19.4	4.1	1.4	0.0	5.6	4.3	1.7	0.0	0.0	0.0	2.8
F2.4 zeroed buffer overwritten	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
F3.1 one element per program	0.0	1.2	10.5	1.4	0.0	1.6	6.1	0.0	0.0	0.0	0.0	1.7	17.6	6.2	9.1	4.2
F3.2 loop bound at run time	0.0	0.0	2.6	0.0	0.0	0.0	0.0	1.4	7.7	2.8	13.0	3.4	5.9	0.0	0.0	11.3
Correct kernels	47	82	38	70	28	62	49	74	13	36	23	58	17	32	33	71

C.5 ANALYSIS OF GENERATED KERNELS

Search changes the computational strategy, not merely the local code. The paired examples in Figure 8 and Figure 9 show that GUIDED SEARCH can escape inefficient implementation choices made by direct generation. On Level 2 matrix multiplication, `Qwen3.5-122B` directly implements the dominant product in `Triton` and reaches at most $0.27\times$ eager `PyTorch`; search instead delegates the roughly 137-billion-FLOP matrix multiplication to `cuBLAS` and fuses only the lightweight post-processing, reaching $1.01\times$ eager and $1.02\times$ `torch.compile`. On Level 1 3D max pooling, search changes the launch granularity from one program per output to 1,024 outputs per program, reducing the launch count from roughly 122 million to 119,000 and improving performance from $0.02\times$ to $1.77\times$ eager and $2.10\times$ `torch.compile`. Additional examples appear in Appendix C.6.

The aggregate analyses support the same mechanism. Search ships substantially fewer invalid `Triton` programs, with API, type, and shape errors dropping sharply on both levels (Figure 4). Among correct kernels, the main differences are concentrated in a small number of consequential structural decisions rather than a wholesale change in coding style (Table 13). On Level 1, searched kernels delegate heavy operators to `PyTorch` in 40.0% of correct solutions versus 24.2% under direct generation, with convolution and matrix multiplication accounting for most of the gap. This behavior is especially concentrated on hard cases: 64 of the 75 hard Level 1 cases solved only by search use such delegation. Search also reduces extra host-side memory passes or launches from 10.0% to 5.1% and one-element-per-program implementations from 5.2% to 1.9%. On Level 2, however, both methods already delegate nearly all convolution and matrix-multiplication work (97.6% versus 99.0%), showing that delegation is a standard ingredient of correct solutions rather than a shortcut specific to search. Most other patterns differ only modestly; the main negative exception is run-time loop bounds, which become more frequent under search and are associated with lower performance (Appendix C.4). Overall, the gains appear to come from steering generation toward better operator placement, fusion, and parallel granularity while leaving most lower-level coding patterns largely unchanged.

(a) NUCLEUS SAMPLING, 0.27×.

```python
@triton.jit
def fused_linear_relu_div_kernel(X_ptr, W_ptr,
    B_ptr, out_ptr, batch_size, in_features,
    out_features, divisor, BLOCK_M: tl.constexpr,
    BLOCK_N: tl.constexpr, BLOCK_K: tl.constexpr):
  # ... row and column offsets and masks of the tile
  acc = tl.zeros((BLOCK_M, BLOCK_N), dtype=tl.float32)
  for k in range(0, in_features, BLOCK_K):
    # ... load a block of X and a block of W
    acc += tl.dot(x_block, tl.trans(w_block),
                  allow_tf32=False)          product in Triton
  b_block = tl.load(B_ptr + col_offsets,
                    mask=col_mask, other=0.0)
  acc = acc + b_block[None, :]
  acc = tl.maximum(acc, 0.0)
  acc = acc / divisor
  # ... output mask
  tl.store(out_ptr + row_offsets[:, None]
      * out_features + col_offsets[None, :],
      acc, mask=out_mask)
```

(b) GUIDED SEARCH, 1.01×.

```python
@triton.jit
def fused_relu_div_kernel(input_ptr, output_ptr,
    n_elements, divisor, BLOCK_SIZE: tl.constexpr):
  block_start = tl.program_id(0) * BLOCK_SIZE
  offsets = block_start + tl.arange(0, BLOCK_SIZE)
  mask = offsets < n_elements
  x = tl.load(input_ptr + offsets, mask=mask,
              other=0.0)
  x = tl.maximum(x, 0.0)
  x = x / divisor
  tl.store(output_ptr + offsets, x, mask=mask)

class ModelNew(nn.Module):
  def forward(self, x):
    x = self.linear(x)                  product in cuBLAS
    x = fused_relu_div(x, self.divisor)
    return x
```

Figure 8. Matrix multiplication, ReLU and division, level 2, problem 63. The left panel shows the faster correct kernel from `Qwen3.5-122B` alone, while the right panel shows the kernel shipped by its GUIDED SEARCH. Comments, type hints and bookkeeping lines are removed. Indentation is shortened. A line `# ...` marks an elision. Shading marks the lines that show the change, with labels identifying the key difference. Speedups are against eager `PyTorch` on one H100. Against `torch.compile`, the kernels run at 0.28× in (a) and 1.02× in (b).

(a) NUCLEUS SAMPLING, 0.02×.

```python
@triton.jit
def maxpool3d_kernel(input_ptr, output_ptr, N, C,
    D_in, H_in, W_in, D_out, H_out, W_out,
    kernel_size: tl.constexpr, stride: tl.constexpr,
    padding: tl.constexpr, dilation: tl.constexpr,
    BLOCK: tl.constexpr = 512):
  pid = tl.program_id(0)
  # ... return past the last output, decode pid
  max_val = -1e38                    one output per program
  for kd in range(kernel_size):      compile-time loop bound
    # ... d_in
    for kh in range(kernel_size):
      # ... h_in
      for kw in range(kernel_size):
        # ... w_in and in_bounds
        if in_bounds:
          # ... in_offset
          val = tl.load(input_ptr + in_offset)
          max_val = tl.maximum(max_val, val)
  # ... out_offset
  tl.store(output_ptr + out_offset, max_val)

def triton_maxpool3d(x, kernel_size, stride=None,
    padding=0, dilation=1):
  # ... output shape and total_elements
  grid = lambda meta: (total_elements,)
```

(b) GUIDED SEARCH, 1.77×.

```python
@triton.jit
def maxpool3d_kernel(x_ptr, out_ptr, N, C, D, H, W,
    out_D, out_H, out_W, kernel_size, stride,
    padding, dilation, BLOCK_SIZE: tl.constexpr):
  pid = tl.program_id(0)
  offsets = pid * BLOCK_SIZE + tl.arange(0,
      BLOCK_SIZE)                 1,024 outputs per program
  total_out = N * C * out_D * out_H * out_W
  mask = offsets < total_out
  # ... decode (b, c, d, h, w) of each offset
  max_val = tl.full((BLOCK_SIZE,), float('-inf'),
                    dtype=tl.float32)
  for kd in range(kernel_size):          run-time loop bound
    # ... in_d and d_valid
    for kh in range(kernel_size):
      # ... in_h and h_valid
      for kw in range(kernel_size):
        # ... in_w, valid and in_idx
        val = tl.load(x_ptr + in_idx, mask=valid,
                      other=float('-inf'))
        max_val = tl.where(val > max_val, val,
                           max_val)
  out_idx = offsets
  tl.store(out_ptr + out_idx, max_val, mask=mask)

def triton_maxpool3d(x, kernel_size, stride,
    padding, dilation):
  # ... output shape, total_out and the grid
  BLOCK_SIZE = 1024
```

Figure 9. 3D max pooling, level 1, problem 43. The left panel shows the faster correct kernel from `DeepSeek-V4-Flash` alone, while the right panel shows the kernel shipped by its GUIDED SEARCH. Comments, type hints and bookkeeping lines are removed. Indentation is shortened. A line `# ...` marks an elision. Shading marks the lines that show the change, with labels identifying the key difference. Speedups are against eager `PyTorch` on one H100. Against `torch.compile`, the kernels run at 0.03× in (a) and 2.10× in (b).

Process guidance expands the set of hard problems the generator can reach. We define a generator–problem pair as hard when at most two of the other seven generators produce any correct kernel among four direct completions. On these cases, GUIDED SEARCH raises the solved share from 9.1% to 33.0% on Level 1 and from 7.5% to 40.6% on Level 2 (Table 8), compared with only 10.9% and 8.3% for a same-budget RANDOM BEAM. This indicates that the gain is not due to branching alone: the PRM changes which partial programs survive, while the ORM selects among the resulting complete kernels. The newly reached solutions are also practically usable: on hard cases solved only by search, the selected kernels achieve median speedups of 1.00× and 1.03× over eager `PyTorch` on Levels 1 and 2, respectively. Thus, on difficult problems the main effect of process guidance is to cross the correctness barrier while retaining roughly reference-level efficiency, expanding the region of useful implementations that direct sampling rarely reaches.

C.6 REPRESENTATIVE KERNEL TRANSFORMATIONS

Figure 8 and Figure 9 pairs two nucleus-sampling kernels with the kernels PROCESSKERNEL selects for the same problems. This appendix extends the comparison to ten pairs. Table 15 lists them, Figures 10–19 show the abridged code of pairs 3–10, and Table 16 gives the reward-model scores of all ten. Each pair sets the fastest correct completion of nucleus sampling against the candidate the ORM ranks first in the pool of GUIDED SEARCH. We drew the pairs from problems on Levels 1 and 2 where the nucleus-sampling kernel is slower than eager `PyTorch` and the selected kernel is faster, and chose ten that cover both levels, five generators and different kinds of implementation change. They illustrate what search changes in a kernel rather than how often it does so; Table 13 and Appendix C.4 report the frequencies.

Three kinds of change recur across the ten pairs. Search delegates the heavy operator to a library and keeps only a fused pointwise tail in `Triton` (pairs 1, 3, 4 and 9). It widens the work per program or per load from one output or one value to blocks of 128 to 1,024 (pairs 2, 4, 5, 6, 7 and 8). And it removes a redundant pass over memory (pair 10). Pairs 7 and 8 also differ in whether the loop bound over the filter is fixed at compile time, but in opposite directions, which suggests that the launch granularity rather than the loop bound accounts for their speedups.

1. *Matrix multiplication, ReLU and division* (level 2, problem 63; Figure 10). The nucleus-sampling kernel computes the product in `Triton`. The selected kernel hands it to `cuBLAS` through `nn.Linear` and fuses only the ReLU and the division into one `Triton` kernel.
2. *3D max pooling* (level 1, problem 43; Figure 11). The nucleus-sampling kernel launches one program per output. The selected kernel runs the same loop over the pooling window on blocks of 1,024 outputs.
3. *Convolution, division and LeakyReLU* (level 2, problem 71; Figure 12). The nucleus-sampling kernel writes the whole convolution in `Triton`, with its loop bounds passed at run time. The selected kernel calls `cuDNN` through `nn.Conv2d` and fuses the division and the LeakyReLU into one `Triton` kernel.
4. *3D convolution, minimum and softmax* (level 2, problem 24; Figure 13). The nucleus-sampling kernel computes the minimum and the softmax in `Triton`, one output position per program and one value per load. The selected kernel leaves the softmax to `PyTorch` and takes the minimum over depth on blocks of 1,024 outputs. Its depth loop runs a fixed 32 steps, which covers this problem's depth of 22 but would not cover a depth above 32.
5. *Transposed 3D convolution, scaling, max pooling, global average and clamp* (level 2, problem 96; Figure 14). Both kernels leave the transposed convolution to `PyTorch` and fuse the remaining operations into one program per channel. The nucleus-sampling kernel loads one value at a time; the selected kernel loads blocks of 128 pooled positions.
6. *Transposed 3D convolution, batch normalization and two average poolings* (level 2, problem 72; Figure 15). Both kernels fuse the two poolings into one $4 \times 4 \times 4$ average. The nucleus-sampling kernel uses one program per output; the selected kernel uses blocks of 1,024 outputs.
7. *Depthwise convolution* (level 1, problem 84; Figure 16), `DeepSeek-V4-Flash`. Both kernels compute the convolution in `Triton`. The nucleus-sampling kernel passes the filter size at run time and covers 128 columns of one output row per program. The selected kernel passes it as a `tl.constexpr`, so that `Triton` can unroll the loop over the filter, and covers 512 output pixels per program.
8. *The same depthwise convolution* (Figure 17), `GPT-OSS-120B`. The nucleus-sampling kernel launches one program per output, with every size fixed at compile time. The selected kernel processes blocks of 1,024 outputs and passes the filter size at run time.
9. *Exclusive cumulative sum* (level 1, problem 92; Figure 18). The nucleus-sampling kernel scans each row in `Triton`, one element after another. The selected kernel computes the inclusive sum with `torch.cumsum` and turns it into the exclusive one with a single `Triton` subtraction.
10. *Group normalization* (level 1, problem 35; Figure 19). The nucleus-sampling kernel reads each group three times, for the mean, the variance and the normalization. The selected kernel reads it twice, collecting the sum and the sum of squares in one pass.

Table 15 adds two observations. The search pool is entirely correct in nine of the ten pairs, while nucleus sampling has at most two correct completions out of four on these problems. The gains also depend on the baseline: against `torch.compile`, the selected kernels of pairs 3, 4, 6, 9 and 10

remain slower than the reference, so on half of the problems search closes the gap to eager `PyTorch` but not to the compiled reference.

The reward models agree with the measurements. Both score the selected kernel above the fastest correct completion of nucleus sampling in all ten pairs (Table 16). Under the PRM, every correct candidate in the search pool scores above every correct completion of nucleus sampling in all ten pairs. Under the ORM this holds in nine pairs; in pair 9, 3 of the 35 correct candidates score below the best correct completion of nucleus sampling.

Table 15. Ten representative kernel pairs. Each pair sets the fastest correct completion of nucleus sampling (*NS*) against the kernel PROCESSKERNEL selects (*PK*), the candidate the ORM ranks first in the pool of GUIDED SEARCH. Speedups are against eager `PyTorch` and `torch.compile` on one H100. *Correct* counts the correct kernels among the generator's four completions and in the search pool.

				Eager		Compile		Correct	
Pair	Level	Problem	Generator	NS	PK	NS	PK	NS	PK
1	2	63	`Qwen3.5-122B`	0.27	1.01	0.28	1.02	2/4	28/28
2	1	43	`DeepSeek-V4-Flash`	0.02	1.77	0.03	2.10	2/4	16/16
3	2	71	`Qwen3.6-35B`	0.26	1.71	0.09	0.56	1/4	20/20
4	2	24	`Muse-Glimmer-30B`	0.20	1.02	0.18	0.89	2/4	12/20
5	2	96	`Qwen3.6-35B`	0.40	1.22	0.34	1.04	1/4	16/16
6	2	72	`DeepSeek-V4-Flash`	0.64	1.06	0.25	0.42	1/4	19/19
7	1	84	`DeepSeek-V4-Flash`	0.91	1.39	0.88	1.34	2/4	19/19
8	1	84	`GPT-OSS-120B`	0.02	1.11	0.02	1.07	1/4	16/16
9	1	92	`Muse-Glimmer-30B`	0.15	1.44	0.04	0.38	1/4	35/35
10	1	35	`Qwen3.5-122B`	0.84	1.28	0.62	0.95	1/4	23/23

Table 16. Reward-model scores of the kernels in Table 15. For each problem, the four completions of nucleus sampling and the whole search pool are scored in one run with the checkpoints the search used; *Scored* is their number. The ORM reads the reference and the kernel. The PRM reads the prompt and the kernel code, the only input available to both methods. Scores are ranking logits, so only their order within a problem is meaningful. In parentheses is the rank the ORM assigns the kernel among all scored kernels of the problem, correct or not; the PRM steers the search but does not choose from the pool, so it has no rank. In pair 6 the search's own scores tie three candidates at the top, and when recomputed another of them scores 0.19 above the selected kernel. In pair 10 an incorrect completion of nucleus sampling scores 0.06 above it.

		ORM		PRM	
Pair	Scored	NS	PK	NS	PK
1	32	−12.75 (31)	8.88 (1)	−1.20	5.13
2	20	−10.69 (19)	2.70 (1)	−0.51	2.84
3	24	−9.00 (22)	5.25 (1)	1.25	5.66
4	24	−4.50 (23)	2.95 (1)	−0.34	2.73
5	20	−9.06 (19)	−2.22 (1)	−0.01	2.36
6	23	0.54 (21)	5.28 (2)	1.66	3.34
7	23	−10.31 (21)	−2.47 (1)	−0.12	3.11
8	20	−11.88 (17)	−1.23 (1)	−1.00	3.36
9	39	−7.09 (34)	−4.41 (1)	0.16	2.58
10	27	−9.25 (26)	−3.47 (2)	1.38	3.86

(a) Matrix multiplication, ReLU and division, level 2, problem 63. `Qwen3.5-122B` alone, 0.27×.

```python
@triton.jit
def fused_linear_relu_div_kernel(X_ptr, W_ptr,
    B_ptr, out_ptr, batch_size, in_features,
    out_features, divisor, BLOCK_M: tl.constexpr,
    BLOCK_N: tl.constexpr, BLOCK_K: tl.constexpr):
  # ... row and column offsets and masks of the tile
  acc = tl.zeros((BLOCK_M, BLOCK_N), dtype=tl.float32)
  for k in range(0, in_features, BLOCK_K):
    # ... load a block of X and a block of W
    acc += tl.dot(x_block, tl.trans(w_block),
                  allow_tf32=False)          product in Triton
  b_block = tl.load(B_ptr + col_offsets,
                    mask=col_mask, other=0.0)
  acc = acc + b_block[None, :]
  acc = tl.maximum(acc, 0.0)
  acc = acc / divisor
  # ... output mask
  tl.store(out_ptr + row_offsets[:, None]
      * out_features + col_offsets[None, :],
      acc, mask=out_mask)
```

(b) The same problem. `Qwen3.5-122B` with the GUIDED SEARCH, 1.01×.

```python
@triton.jit
def fused_relu_div_kernel(input_ptr, output_ptr,
    n_elements, divisor, BLOCK_SIZE: tl.constexpr):
  block_start = tl.program_id(0) * BLOCK_SIZE
  offsets = block_start + tl.arange(0, BLOCK_SIZE)
  mask = offsets < n_elements
  x = tl.load(input_ptr + offsets, mask=mask,
              other=0.0)
  x = tl.maximum(x, 0.0)
  x = x / divisor
  tl.store(output_ptr + offsets, x, mask=mask)

class ModelNew(nn.Module):
  def forward(self, x):
    x = self.linear(x)                  product in cuBLAS
    x = fused_relu_div(x, self.divisor)
    return x
```

Figure 10. Pairs 1 of Table 15

(c) 3D max pooling, level 1, problem 43. `DeepSeek-V4-Flash` alone, 0.02×.

```python
@triton.jit
def maxpool3d_kernel(input_ptr, output_ptr, N, C,
    D_in, H_in, W_in, D_out, H_out, W_out,
    kernel_size: tl.constexpr, stride: tl.constexpr,
    padding: tl.constexpr, dilation: tl.constexpr,
    BLOCK: tl.constexpr = 512):
  pid = tl.program_id(0)
  # ... return past the last output, decode pid
  max_val = -1e38                       one output per program
  for kd in range(kernel_size):         compile-time loop bound
    # ... d_in
    for kh in range(kernel_size):
      # ... h_in
      for kw in range(kernel_size):
        # ... w_in and in_bounds
        if in_bounds:
          # ... in_offset
          val = tl.load(input_ptr + in_offset)
          max_val = tl.maximum(max_val, val)
  # ... out_offset
  tl.store(output_ptr + out_offset, max_val)

def triton_maxpool3d(x, kernel_size, stride=None,
    padding=0, dilation=1):
  # ... output shape and total_elements
  grid = lambda meta: (total_elements,)
```

(d) The same problem. `DeepSeek-V4-Flash` with the GUIDED SEARCH, 1.77×.

```python
@triton.jit
def maxpool3d_kernel(x_ptr, out_ptr, N, C, D, H, W,
    out_D, out_H, out_W, kernel_size, stride,
    padding, dilation, BLOCK_SIZE: tl.constexpr):
  pid = tl.program_id(0)
  offsets = pid * BLOCK_SIZE + tl.arange(0,
      BLOCK_SIZE)                   1,024 outputs per program
  total_out = N * C * out_D * out_H * out_W
  mask = offsets < total_out
  # ... decode (b, c, d, h, w) of each offset
  max_val = tl.full((BLOCK_SIZE,), float('-inf'),
                    dtype=tl.float32)
  for kd in range(kernel_size):         run-time loop bound
    # ... in_d and d_valid
    for kh in range(kernel_size):
      # ... in_h and h_valid
      for kw in range(kernel_size):
        # ... in_w, valid and in_idx
        val = tl.load(x_ptr + in_idx, mask=valid,
                      other=float('-inf'))
        max_val = tl.where(val > max_val, val,
                           max_val)
  out_idx = offsets
  tl.store(out_ptr + out_idx, max_val, mask=mask)

def triton_maxpool3d(x, kernel_size, stride,
    padding, dilation):
  # ... output shape, total_out and the grid
  BLOCK_SIZE = 1024
```

Figure 11. Pairs 2 of Table 15

(a) Convolution, division and LeakyReLU, level 2, problem 71. `Qwen3.6-35B` alone, 0.26×.

```python
@triton.jit
def fused_conv_div_leaky_relu_kernel(
    x_ptr, w_ptr, b_ptr, out_ptr, batch_size,
    in_channels, out_channels, in_height, in_width,
    out_height, out_width, kernel_size, divisor,
    negative_slope, BLOCK_SIZE: tl.constexpr):
  idx = tl.program_id(0) * BLOCK_SIZE + tl.arange(0,
      BLOCK_SIZE)
  # ... mask, decode (n, c_out, h_out, w_out), bias
  acc = tl.zeros([BLOCK_SIZE], dtype=tl.float32)
  for c_in in range(in_channels):          convolution in Triton
    for kh in range(kernel_size):
      for kw in range(kernel_size):
        # ... x_idx, w_idx and valid_mask
        x_val = tl.load(x_ptr + x_idx,
                        mask=valid_mask, other=0.0)
        w_val = tl.load(w_ptr + w_idx)
        acc += x_val * w_val
  acc += bias
  acc = acc / divisor
  out = tl.where(acc > 0, acc, acc * negative_slope)
  tl.store(out_ptr + idx, out, mask=mask)

class ModelNew(nn.Module):
  def forward(self, x):
    weight = self.conv.weight
    bias = self.conv.bias
    return fused_conv_div_leaky_relu(x, weight, bias,
        self.divisor, self.negative_slope)
```

(b) The same problem. `Qwen3.6-35B` with the GUIDED SEARCH, 1.71×.

```python
@triton.jit
def fused_div_leaky_relu_kernel(x_ptr, out_ptr,
    n_elements, divisor, negative_slope,
    BLOCK_SIZE: tl.constexpr):
  pid = tl.program_id(axis=0)
  block_start = pid * BLOCK_SIZE
  offsets = block_start + tl.arange(0, BLOCK_SIZE)
  mask = offsets < n_elements
  x = tl.load(x_ptr + offsets, mask=mask, other=0.0)
  x_div = x / divisor
  out = tl.where(x_div > 0, x_div,
                 x_div * negative_slope)
  tl.store(out_ptr + offsets, out, mask=mask)

def triton_fused_div_leaky_relu(x, divisor,
    negative_slope=0.01):
  # ... output and n_elements
  BLOCK_SIZE = 1024
  # ... grid and launch

class ModelNew(nn.Module):
  def forward(self, x):
    x = self.conv(x)                       convolution in cuDNN
    x = triton_fused_div_leaky_relu(x, self.divisor)
    return x
```

Figure 12. Pairs 3 of Table 15.

(c) 3D convolution, minimum and softmax, level 2, problem 24. `Muse-Glimmer-30B` alone, 0.20×.

```python
@triton.jit
def min_reduce_kernel(x_ptr, out_ptr, B, H, W,
    stride_b, stride_c, stride_d, stride_h, stride_w,
    ..., C: tl.constexpr, D: tl.constexpr):
  pid = tl.program_id(0)
  # ... decode (b, c, h, w) from pid
  min_val = 1e30                           one position per program
  for d in range(D):
    val = tl.load(
      x_ptr + b * stride_b + c * stride_c
      + d * stride_d + h * stride_h + w * stride_w)
    if val < min_val:
      min_val = val
  # ... store min_val at (b, c, h, w)

@triton.jit
def softmax_kernel(in_ptr, out_ptr, B, H, W, ...,
    C: tl.constexpr):
  pid = tl.program_id(0)
  # ... decode (b, h, w) from pid
  max_val = -1e30
  for c in range(C):
    v = tl.load(in_ptr + b * stride_b + c * stride_c
                + h * stride_h + w * stride_w)
    if v > max_val:
      max_val = v
  # ... two more loops over c: the sum of the
  # ... exponentials, then each normalized output

def triton_min(x):
  # ... shapes and output
  grid = (B * C * H * W,)

class ModelNew(nn.Module):
  def forward(self, x):
    x = self.conv(x)
    x = triton_min(x)
    x = triton_softmax(x)                  softmax in Triton
    return x
```

(d) The same problem. `Muse-Glimmer-30B` with the GUIDED SEARCH, 1.02×.

```python
@triton.jit
def min_reduce_kernel(x_ptr, out_ptr, B, C, D, H, W,
    stride_n, stride_c, stride_d, stride_h, stride_w,
    BLOCK_SIZE: tl.constexpr):
  pid = tl.program_id(0)
  start = pid * BLOCK_SIZE
  offsets = start + tl.arange(0, BLOCK_SIZE)
  # ... mask, decode (b, c, h, w) and base
  min_val = tl.full([BLOCK_SIZE], 1e30,
                    dtype=tl.float32)
  for d in range(32):                      fixed 32 steps
    if d < D:
      ptr = base + d * stride_d
      val = tl.load(x_ptr + ptr, mask=mask, other=0.0)
      min_val = tl.where(mask,
          tl.minimum(min_val, val), min_val)
  tl.store(out_ptr + offsets, min_val, mask=mask)

def triton_min(x):
  # ... shapes, output and strides
  BLOCK_SIZE = 1024                        1,024 outputs per program
  # ... grid and launch

class ModelNew(nn.Module):
  def forward(self, x):
    x = self.conv(x)
    if self.dim == 2:
      x = triton_min(x)
    else:
      x = torch.min(x, dim=self.dim)[0]
    x = torch.softmax(x, dim=1)            softmax in PyTorch
    return x
```

Figure 13. Pairs 4 of Table 15.

(a) Transposed 3D convolution, scaling, max pooling, global average and clamp, level 2, problem 96. `Qwen3.6-35B` alone, 0.40×.

```python
@triton.jit
def fused_pool_scale_clamp_kernel(x_ptr, out_ptr,
    B, C, D, H, W, scale, clamp_min, clamp_max):
  pid = tl.program_id(0)
  # ... return past B * C, pooled sizes, base_ptr
  sum_max = 0.0
  for di in range(D_pooled):
    for hi in range(H_pooled):
      for wi in range(W_pooled):
        cur_max = -float('inf')
        for di2 in range(2):
          for hi2 in range(2):
            for wi2 in range(2):
              # ... idx of this window element
              val = tl.load(base_ptr + idx)
              if val > cur_max:                    one value per load
                cur_max = val
        # ... track block_max, which is never used
        sum_max += cur_max
  avg_max = sum_max / num_pooled
  out_val = scale * avg_max
  out_val = tl.maximum(tl.minimum(out_val,
                                  clamp_max), clamp_min)
  tl.store(out_ptr + pid, out_val)

def triton_fused_pool_scale_clamp(x, scale,
    clamp_min, clamp_max):
  # ... shapes and output
  grid = lambda meta: (B * C,)

class ModelNew(nn.Module):
  def forward(self, x):
    x = self.conv_transpose(x)
    x = triton_fused_pool_scale_clamp(x, self.scale,
        self.clamp_min, self.clamp_max)
    return x
```

(b) The same problem. `Qwen3.6-35B` with the GUIDED SEARCH, 1.22×.

```python
@triton.jit
def fused_post_conv_kernel(conv_out_ptr, scale,
    out_ptr, B, C, D, H, W, D_prime, H_prime,
    W_prime, stride_b, stride_c, stride_d, stride_h,
    stride_w, BLOCK_SIZE: tl.constexpr):
  pid = tl.program_id(0)
  # ... decode (b, c) and base_ptr
  sum_max = 0.0
  num_elements = D_prime * H_prime * W_prime
  for start_idx in range(0, num_elements, BLOCK_SIZE):
    offsets = start_idx + tl.arange(0, BLOCK_SIZE)
    mask = offsets < num_elements
    # ... pooled coordinates and window corner
    max_val = tl.full((BLOCK_SIZE,), -float('inf'),
                      dtype=tl.float32)
    for k_d in range(2):
      for k_h in range(2):
        for k_w in range(2):
          # ... valid mask and conv_offset
          val = tl.load(base_ptr + conv_offset,
              mask=valid, other=-float('inf'))
          max_val = tl.maximum(max_val, val)
    sum_max += tl.sum(tl.where(mask, max_val, 0.0))
  avg_max = sum_max / num_elements
  scaled = avg_max * scale
  out_val = tl.maximum(0.0, tl.minimum(1.0, scaled))
  # ... store out_val for (b, c)

def triton_fused_post_conv(conv_out, scale):
  # ... shapes, output and strides
  BLOCK_SIZE = 128                          128 values per load
  grid = (B * C,)
```

Figure 14. Pairs 5 of Table 15.

(a) Transposed 3D convolution, batch normalization and two average poolings, level 2, problem 72. `DeepSeek-V4-Flash` alone, 0.64×.

```python
@triton.jit
def avg_pool3d_kernel(input_ptr, output_ptr,
    N, C, D, H, W, D_out, H_out, W_out,
    BLOCK_SIZE: tl.constexpr):
  pid = tl.program_id(0)
  # ... decode (n, c, d, h, w) from pid
  d_start = d * 4
  h_start = h * 4
  w_start = w * 4
  base = n * C * D * H * W + c * D * H * W
  sum_val = 0.0                          one output per program
  for kz in range(4):
    dz = d_start + kz
    for ky in range(4):
      dy = h_start + ky
      for kx in range(4):
        dx = w_start + kx
        idx = base + dz * H * W + dy * W + dx
        v = tl.load(input_ptr + idx)
        sum_val = sum_val + v
  avg = sum_val / 64.0
  # ... out_idx
  tl.store(output_ptr + out_idx, avg)

def triton_avg_pool3d(x):
  # ... shapes, output and total_elements
  grid = lambda meta: (total_elements,)

class ModelNew(nn.Module):
  def forward(self, x):
    x = self.conv_transpose(x)
    x = self.batch_norm(x)
    x = triton_avg_pool3d(x)
    return x
```

(b) The same problem. `DeepSeek-V4-Flash` with the GUIDED SEARCH, 1.06×.

```python
@triton.jit
def fused_avg_pool_kernel(in_ptr, out_ptr,
    B, C, D, H, W, D_out, H_out, W_out,
    kernel_size, stride, BLOCK_SIZE: tl.constexpr):
  pid = tl.program_id(0)
  block_start = pid * BLOCK_SIZE
  offsets = block_start + tl.arange(0, BLOCK_SIZE)
  total_out = B * C * D_out * H_out * W_out
  mask = offsets < total_out
  # ... strides, decode (b, c, d2, h2, w2), base
  acc = tl.zeros([BLOCK_SIZE], dtype=tl.float32)
  for kd in range(4):
    for kh in range(4):
      for kw in range(4):
        # ... offset of this window element
        val = tl.load(in_ptr + offset, mask=mask,
                      other=0.0)
        acc += val
  out_val = acc / 64.0
  # ... out_offset
  tl.store(out_ptr + out_offset, out_val, mask=mask)

def triton_fused_avg_pool(x):
  # ... shapes, output and total_out
  BLOCK_SIZE = 1024                    1,024 outputs per program
  grid = (triton.cdiv(total_out, BLOCK_SIZE),)

class ModelNew(nn.Module):
  def forward(self, x):
    x = self.conv_transpose(x)
    x = self.batch_norm(x)
    x = triton_fused_avg_pool(x)
    return x
```

Figure 15. Pairs 6 of Table 15.

(a) Depthwise convolution, level 1, problem 84. `DeepSeek-V4-Flash` alone, 0.91×.

```python
@triton.jit
def depthwise_conv2d_kernel(input_ptr, weight_ptr,
    output_ptr, ..., input_height, input_width,
    output_height, output_width, kernel_height,
    kernel_width, num_channels, stride: tl.int32,
    BLOCK_SIZE: tl.constexpr):
  # ... program ids, (batch, channel), col_start
  col_offsets = col_start + tl.arange(0, BLOCK_SIZE)
  # ... col_mask and base pointers
  acc = tl.zeros([BLOCK_SIZE], dtype=tl.float32)
  for kr in range(kernel_height):          run-time loop bound
    input_row = pid_row * stride + kr
    row_valid = input_row < input_height
    for kc in range(kernel_width):
      w = tl.load(w_base + kr * kernel_width + kc)
      input_col = col_offsets * stride + kc
      mask = row_valid & (input_col < input_width)
      in_val = tl.load(in_base + input_row
          * in_height_stride + input_col
          * in_width_stride, mask=mask, other=0.0)
      acc += w * in_val
  # ... store acc for this row and column block

def triton_depthwise_conv2d(x, weight, in_channels,
    kernel_size, stride=1, padding=0, bias=None):
  # ... output shape and strides
  BLOCK_SIZE = 128                  128 columns per program
  # ... grid over (batch * channel, row, column block)
  depthwise_conv2d_kernel[grid](x, weight, out, ...,
      kernel_size, kernel_size, in_channels, stride,
      BLOCK_SIZE=BLOCK_SIZE)
```

(b) The same problem. `DeepSeek-V4-Flash` with the GUIDED SEARCH, 1.39×.

```python
@triton.jit
def depthwise_conv_kernel(x_ptr, w_ptr, y_ptr, N, C,
    H_in, W_in, H_out, W_out, kw: tl.constexpr,
    BLOCK_SPATIAL: tl.constexpr):
  # ... program ids, (batch, channel)
  spatial_start = pid_spatial * BLOCK_SPATIAL
  offsets = spatial_start + tl.arange(0,
      BLOCK_SPATIAL)
  mask = offsets < (H_out * W_out)
  # ... (oh, ow) and base pointers
  acc = tl.zeros([BLOCK_SPATIAL], dtype=tl.float32)
  for kh in range(kw):                  compile-time loop bound
    for kk in range(kw):
      ih = oh + kh
      iw = ow + kk
      in_bounds = (ih < H_in) & (iw < W_in)
      input_idx = ih * W_in + iw
      inp = tl.load(input_base + input_idx,
                    mask=in_bounds, other=0.0)
      weight_val = tl.load(weight_base + kh * kw + kk)
      acc += inp * weight_val
  tl.store(output_base + offsets, acc, mask=mask)

def triton_depthwise_conv2d(x, weight, stride,
    padding):
  # ... PyTorch fallback, output shape
  BLOCK_SPATIAL = 512               512 pixels per program
  # ... grid over (batch * channel, pixel block)
  depthwise_conv_kernel[grid](x, weight, y,
      N, C, H_in, W_in, H_out, W_out, kw=k,
      BLOCK_SPATIAL=BLOCK_SPATIAL)
```

Figure 16. Pairs 7 of Table 15.

(a) The same problem. `GPT-OSS-120B` alone, 0.02×.

```python
@triton.jit
def depthwise_conv_kernel(in_ptr, weight_ptr,
    out_ptr, B: tl.constexpr, C: tl.constexpr,
    H: tl.constexpr, W: tl.constexpr,
    H_out: tl.constexpr, W_out: tl.constexpr,
    K: tl.constexpr):
  pid = tl.program_id(0)
  total = B * C * H_out * W_out
  if pid >= total:
    return
  # ... decode (b, c, h_out, w_out) from pid
  acc = 0.0                         one output per program
  for i in range(K):                compile-time loop bound
    for j in range(K):
      # ... input and weight offsets of tap (i, j)
      x = tl.load(in_ptr + input_offset)
      w = tl.load(weight_ptr + weight_offset)
      acc += x * w
  tl.store(out_ptr + pid, acc)

def triton_depthwise_conv(x, weight, bias=None):
  # ... shapes and output
  total = B * C * H_out * W_out
  grid = (total,)
```

(b) The same problem. `GPT-OSS-120B` with the GUIDED SEARCH, 1.11×.

```python
@triton.jit
def depthwise_conv_kernel(inp_ptr, w_ptr, out_ptr,
    N, C, H, W, K, H_out, W_out,
    stride: tl.constexpr, padding: tl.constexpr,
    BLOCK_SIZE: tl.constexpr):
  pid = tl.program_id(0)
  idx = pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)
  total = N * C * H_out * W_out
  mask = idx < total
  # ... decode (n, c, h_out, w_out), base offsets
  acc = tl.zeros([BLOCK_SIZE], dtype=tl.float32)
  for ki in range(K):                   run-time loop bound
    for kj in range(K):
      h_in = h_out * stride - padding + ki
      w_in = w_out * stride - padding + kj
      # ... valid mask of tap (ki, kj)
      inp_idx = inp_offset + h_in * W + w_in
      inp_val = tl.load(inp_ptr + inp_idx,
                        mask=valid & mask, other=0.0)
      w_idx = w_offset + ki * K + kj
      w_val = tl.load(w_ptr + w_idx, mask=mask,
                      other=0.0)
      acc += inp_val * w_val
  tl.store(out_ptr + out_offset, acc, mask=mask)

def triton_depthwise_conv(x, weight, stride=1,
    padding=0):
  # ... shapes, output and total_out
  BLOCK_SIZE = 1024                 1,024 outputs per program
  # ... grid over blocks of BLOCK_SIZE outputs
```

Figure 17. Pairs 8 of Table 15.

(a) Exclusive cumulative sum, level 1, problem 92. `Muse-Glimmer-30B` alone, 0.15×.

```python
@triton.jit
def exclusive_cumsum_kernel(x_ptr, out_ptr, n_rows,
    n_cols, stride0, stride1,
    BLOCK_SIZE: tl.constexpr):
  row = tl.program_id(0)
  if row >= n_rows:
    return
  row_offset = row * stride0
  acc = 0.0
  col = 0
  while col < n_cols:                          # one element at a time
    for i in range(BLOCK_SIZE):
      cur_col = col + i
      # ... an empty branch for cur_col >= n_cols
      if cur_col < n_cols:
        val = tl.load(x_ptr + row_offset
                      + cur_col * stride1)
        tl.store(out_ptr + row_offset
                 + cur_col * stride1, acc)
        acc += val
    col += BLOCK_SIZE

def triton_exclusive_cumsum(x):
  # ... torch.cumsum fallback for inputs not 2D
  BLOCK_SIZE = 256
  grid = (n_rows,)
```

(b) The same problem. `Muse-Glimmer-30B` with the GUIDED SEARCH, 1.44×.

```python
@triton.jit
def exclusive_sub_kernel(inclusive_ptr, x_ptr,
    out_ptr, n_elements, BLOCK_SIZE: tl.constexpr):
  pid = tl.program_id(0)
  off = pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)
  mask = off < n_elements
  inc = tl.load(inclusive_ptr + off, mask=mask)
  x = tl.load(x_ptr + off, mask=mask)
  tl.store(out_ptr + off, inc - x,             # one subtraction
           mask=mask)

class ModelNew(nn.Module):
  def forward(self, x):
    x = x.contiguous()
    inclusive = torch.cumsum(x, dim=self.dim)  # PyTorch
    out = torch.empty_like(x)
    if self.dim == 1 and x.dim() == 2:
      triton_exclusive(inclusive, x, out)
      return out
    # ... fallback for other dims
```

Figure 18. Pair 9 of Table 15. The left panel shows the faster correct kernel of nucleus sampling, and the right panel shows the kernel shipped by its GUIDED SEARCH. Speedups are against eager `PyTorch` on one H100. Against `torch.compile` the kernels run at 0.04× in (a) and 0.38× in (b).

(c) Group normalization, level 1, problem 35. `Qwen3.5-122B` alone, 0.84×.

```python
@triton.jit
def group_norm_forward_kernel(x_ptr, out_ptr,
    weight_ptr, bias_ptr, mean_ptr, rstd_ptr,
    N: tl.constexpr, C_per_group: tl.constexpr,
    HxW: tl.constexpr, eps: tl.constexpr,
    BLOCK_C: tl.constexpr, BLOCK_HW: tl.constexpr,
    HAS_WEIGHT: tl.constexpr, HAS_BIAS: tl.constexpr):
  pid = tl.program_id(0)
  mean = 0.0
  for start_c in range(0, C_per_group, BLOCK_C):
    # ... c_offsets
    for start_hw in range(0, HxW, BLOCK_HW):
      # ... hw_offsets, indices and mask
      x = tl.load(x_ptr + indices, mask=mask,  # read 1
                  other=0.0)
      mean += tl.sum(x)
  mean = mean / (C_per_group * HxW)
  # ... store mean
  var_sum = 0.0
  for start_c in range(0, C_per_group, BLOCK_C):
    # ... c_offsets
    for start_hw in range(0, HxW, BLOCK_HW):
      # ... hw_offsets, indices and mask
      x = tl.load(x_ptr + indices, mask=mask,  # read 2
                  other=0.0)
      diff = x - mean
      var_sum += tl.sum(diff * diff)
  # ... var, rstd and store rstd
  for start_c in range(0, C_per_group, BLOCK_C):
    # ... third pass: load x again, normalize,  # read 3
    # ... apply weight and bias, store
```

(d) The same problem. `Qwen3.5-122B` with the GUIDED SEARCH, 1.28×.

```python
@triton.jit
def group_norm_kernel(X, Y, weight, bias,
    B: tl.constexpr, C: tl.constexpr, H: tl.constexpr,
    W: tl.constexpr, G: tl.constexpr,
    eps: tl.constexpr, BLOCK_SIZE: tl.constexpr):
  # ... (b, g) from pid, channel range,
  # ... HW and total_elements
  mean = tl.zeros([1], dtype=tl.float32)
  sum_sq = tl.zeros([1], dtype=tl.float32)
  for c in range(c_start, c_end):
    for hw in range(0, HW, BLOCK_SIZE):
      offsets = hw + tl.arange(0, BLOCK_SIZE)
      mask = offsets < HW
      x = tl.load(X + b * C * HW + c * HW + offsets,
                  mask=mask, other=0.0)        # read 1
      mean += tl.sum(x)
      sum_sq += tl.sum(x * x)
  mean = mean / total_elements
  variance = sum_sq / total_elements - mean * mean
  rstd = 1.0 / tl.sqrt(variance + eps)
  for c in range(c_start, c_end):
    for hw in range(0, HW, BLOCK_SIZE):
      # ... offsets and mask
      x = tl.load(X + b * C * HW + c * HW + offsets,
                  mask=mask, other=0.0)        # read 2
      x_norm = (x - mean) * rstd
      w = tl.load(weight + c)
      b_val = tl.load(bias + c)
      out = x_norm * w + b_val
      tl.store(Y + b * C * HW + c * HW + offsets,
               out, mask=mask)
```

Figure 19. Pair 10 of Table 15

C.7 Qualitative Analysis of Search and Selection

Efficiency-Aware Kernel Search. We show in Figure 20 a concrete example of the kernels generated by Muse-Glimmer-30B for exclusive cumulative sum (KernelBench Level 1 problem 92). Specifically, we report the learned scores and measured H100 execution time alongside the code excerpts. As observed, direct generation constructs a custom row-wise scan that serializes work along the column dimension and takes 86.6 ms. In contrast, guided search discovers a library-core plus custom-tail implementation: `torch.cumsum` performs the prefix scan, and a generated Triton kernel converts the inclusive result to an exclusive result in parallel. The ORM selects an 8.93 ms candidate, which is $9.70\times$ faster than direct generation. All 35 candidates in this pool pass the saved correctness checks, yet their runtimes range from 8.93 to 25.7 ms, allowing us to examine the efficiency signal without confounding it with correctness failures.

(a) Exclusive cumulative sum, Level 1, problem 92. `Muse-Glimmer-30B` direct sample 0: 86.6 ms ($0.149\times$ eager).

```
@triton.jit
def exclusive_cumsum_kernel(x_ptr, out_ptr, n_rows,
    n_cols, stride0, stride1,
    BLOCK_SIZE: tl.constexpr):
  row = tl.program_id(0)
  row_offset = row * stride0
  acc = 0.0
  col = 0
  while col < n_cols:                          serial columns
    for i in range(BLOCK_SIZE):
      cur_col = col + i
      if cur_col < n_cols:
        val = tl.load(x_ptr + row_offset
                      + cur_col * stride1)
        tl.store(out_ptr + row_offset
                 + cur_col * stride1, acc)
        acc += val
    col += BLOCK_SIZE

def triton_exclusive_cumsum(x):
  # ... allocate output and read dimensions
  grid = (n_rows,)                          one program per row
  exclusive_cumsum_kernel[grid](...)
```

(b) The same task and generator with guided search. ORM-selected sample 2: 8.93 ms ($1.445\times$ eager).

```
def forward(self, x):
  x = x.contiguous()
  inclusive = torch.cumsum(x, dim=self.dim)    library scan
  out = torch.empty_like(x)
  triton_exclusive(inclusive, x, out)          parallel tail
  return out

@triton.jit
def exclusive_sub_kernel(inclusive, x, out, n,
    BLOCK_SIZE: tl.constexpr):
  pid = tl.program_id(0)
  off = pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)
  mask = off < n
  inc = tl.load(inclusive + off, mask=mask)
  val = tl.load(x + off, mask=mask)
  tl.store(out + off, inc - val, mask=mask)

def triton_exclusive(inclusive, x, out):
  n = inclusive.numel()
  BLOCK_SIZE = 1 << 14              16,384 elements per program
  grid = ((n + BLOCK_SIZE - 1) // BLOCK_SIZE,)
  exclusive_sub_kernel[grid](...)
```

Figure 20. Guided search changes the library–custom boundary for the same exclusive-cumulative-sum task. Both panels use Muse-Glimmer-30B on KernelBench Level 1 problem 92. **Left:** direct sample 0 implements a row-wise custom scan. **Right:** ORM-selected PRM-pool sample 2 delegates the scan to `torch.cumsum` and uses a generated Triton kernel for the parallel inclusive-to-exclusive conversion. Yellow highlights the contrasted statements. direct generation reaches $0.149\times$ compared with PyTorch-eager baseline eager, whereas the ORM-selected implementation reaches $1.445\times$ eager. The selected implementation is also $9.70\times$ faster than direct sample 0 (86.6 ms versus 8.93 ms; 89.7% lower latency). Excerpts preserve the recorded operations, with omissions and formatting changes for readability.