GPU FORECASTERS: LANGUAGE MODELS AS SPECULATIVE SURROGATES FOR KERNEL RUNTIME OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

GPU kernels are the workhorse of modern deep learning, and optimizing them (via evolutionary search or coding agents) usually requires repeated measurement on target hardware. While these measurements provide the ground-truth signal necessary for kernel search, they are costly, because each evaluation of a kernel requires compilation and repeated execution on a GPU. As improvements in LLM inference reduce the cost of writing novel kernels and LLM-driven searches scale to large search budgets, on-device evaluation becomes a bottleneck. To address this, we study how LLMs can serve as calibrated, speculative surrogates of the GPU: probabilistic performance models that forecast the speedup of a candidate kernel from its source. During a kernel search algorithm's generation phase, the surrogate forecasts runtimes for each generated kernel, and only the top-ranked kernels are scored by the GPU. The search uses the speculative forecasts by the surrogate as scores for the remaining kernels. We first evaluate four off-the-shelf LLMs as surrogates. We find that LLMs can act as effective surrogates, helping searches find fast kernels while measuring only a fraction of generated candidates on the GPU. On 1,347 parent-child pairs from real searches, LLMs identify discovery moments, i.e., children that are much faster than their parents, at rates above chance. We then integrate a surrogate into evolutionary kernel search to prioritize candidates for GPU measurement. Measuring only the most promising generated candidates, this search matches or outperforms GPU-only search on most benchmark tasks. Finally, we train a surrogate with reinforcement learning and find that calibration rewards improve calibration without hurting ranking, while correctness rewards alone make confidence less reliable. We release the 12,388 measured LLM-generated kernels produced by these searches. These results suggest that LLMs can play a broader role in kernel optimization, by acting as virtual models of a GPU rather than solely as kernel generators for search.[1]

1 INTRODUCTION

GPU kernels are the low-level programs that run tensor operations on GPUs. They are critical to modern deep learning, and small differences in their efficiency translate into large differences in the cost and feasibility of training and inference. Writing high-performance kernels is among the most specialized tasks in software engineering, requiring detailed knowledge of memory hierarchies, tensor core instructions, and architecture-specific tradeoffs that vary across hardware generations. Recent work has applied LLMs to this problem, with LLM-powered search methods such as CUDA Agent (Dai et al., 2026), TTT-Discover (Yuksekgonul et al., 2026), and K-Search (Cao et al., 2026) demonstrating strong results. The core of these search procedures is a generation-evaluation cycle where generated kernels are evaluated for correctness and speed. However, each evaluation is expensive, requiring compilation and on-GPU execution, and searches use many evaluations (e.g., Yuksekgonul et al. (2026) use 25,600 H100 GPU evaluations for optimizing the `TriMul` kernel). In addition, evaluation throughput can be an order of magnitude slower ($\frac{1}{10}$-th) than generation throughput. These factors make on-GPU evaluation a bottleneck for kernel optimization. Although a large body of prior work has focused on proposing new kernels (Chen et al., 2026; Dai et al., 2026;

[1] All code and data are provided in the supplementary materials.

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.

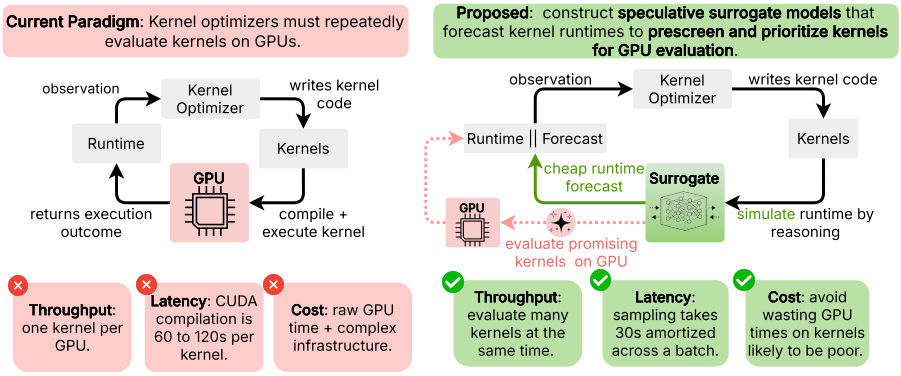

Figure 1: Left: LLM-driven kernel optimizers make heavy use of GPU evaluations during search to score generated kernels, making search costly when many candidates must be compiled, executed, and profiled. Right: a speculative surrogate forecasts the runtime of each candidate kernel by reasoning about its source code. The candidates with the most promising forecasts are measured on the GPU. The remaining candidates keep their forecasts as speculative scores, which a later measurement can replace, so the search runs on a mix of measured and forecast runtimes and can consider more candidates than the GPU budget allows.

Cao et al., 2026; Ouyang et al., 2025; Saba et al., 2026; Hari et al., 2026), the problem of reducing their evaluation cost has received less attention.

One way to reduce the number of on-device evaluations is to use a model of the device to predict the outcome of running a kernel before running it. Surrogate-assisted optimization (Jones et al., 1998) uses such a model, a *surrogate*, to pre-screen candidates before evaluating them on an expensive objective. A probabilistic surrogate outputs a predictive distribution over the objective, and the search applies an acquisition rule to this distribution to choose which candidates to evaluate. The ability of LLMs to reason about how general purpose programs will execute (Gu et al., 2024; FAIR CodeGen Team, 2025; Prasad et al., 2025) suggests they may be effective surrogates.

Can LLM predictions help evolutionary kernel search explore more candidate kernels cheaply, without measuring every candidate on a GPU?

To answer this question, we first evaluate off-the-shelf LLMs as probabilistic GPU surrogates, testing whether their speedup forecasts can rank candidate kernels and identify large improvements in real search traces. We then integrate a surrogate into evolutionary search by using it as a *speculative surrogate*. The surrogate predicts a runtime for each generated kernel during the mutation steps of the search. We evaluate the highest-ranked kernel candidates on a GPU while keeping the speculative score for the other kernels until the search decides to return to them. Finally, we train a surrogate with reinforcement learning to study how different rewards affect forecast calibration and candidate ranking.

This paper makes three contributions. (1) We find that LLMs can rank candidate kernels well enough that measuring only their top picks often finds kernels among the fastest available, and can identify the mutations that make a kernel much faster than its parent in real search traces at rates above chance. (2) The surrogate lets evolutionary search consider more candidate kernels within the same GPU measurement budget, matching or improving on GPU-only search on 4 out of 6 kernel optimization tasks. (3) We study the effect of reinforcement learning on surrogates and find that calibration rewards combined with correctness rewards improve the utility of forecasts, while correctness rewards alone damage it. Measuring a kernel's runtime requires exclusive use of the GPU, while the surrogate can forecast multiple candidates concurrently. With 16 forecasts in flight, it achieves an estimated 11× the throughput of Triton evaluation and 83× that of CUDA evaluation, at 12× and 47× lower cost per candidate, respectively (Table 3), greatly increasing the number of kernels a search can explore for a fixed GPU evaluation budget. We also release a collection of 12,388 LLM-generated GPU kernels with measured runtimes, spanning 118 problems, CUDA and Triton, and three GPU types, which supplies the evaluation and training set used in this paper (Section C).

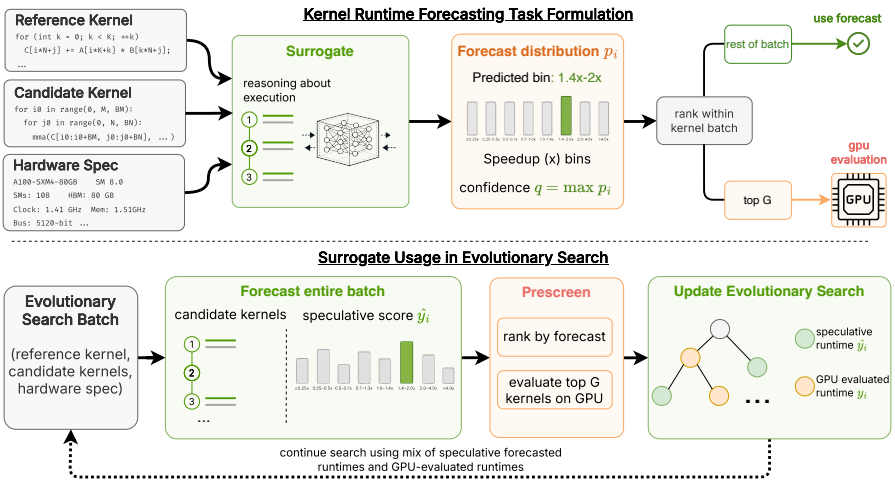

Figure 2: Top: the surrogate reads the reference kernel, the candidate kernel, and the hardware specification, and produces a distributional forecast of the candidate kernel's speedup over the reference. The candidates in a batch are ranked by expected speedup, and only the top G are evaluated on the GPU. Bottom: use of the surrogate inside an evolutionary search. In each generation step, the surrogate forecasts every candidate in the batch, the top G candidates are evaluated on the GPU, and all candidates enter the search archive with either a measured runtime or a speculative score. The search continues on this mix of measured and forecast runtimes.

2 Method: LLM as Virtual GPU Surrogate

2.1 Task Formulation

Kernel Evaluation. A kernel optimization problem starts with a target operation and a reference implementation that defines the correct outputs on a fixed set of test inputs. For example, one task in our evaluation is `TriMul`,[2] a triangle multiplicative update forward pass from AlphaFold-style protein structure models (Abramson et al., 2024). LLM-driven kernel search procedures can rapidly generate thousands of candidate kernels intended to implement the same operation, as shown in Figure 1. Ordinarily, each candidate kernel is compiled, run on the test inputs, checked against the reference outputs, and timed on the target GPU. The GPU outcome for a candidate kernel has two parts: validity and speed. Invalidity means that the candidate fails to compile, fails at runtime, or fails correctness checking. If the candidate is valid, we measure its runtime and compare it to the reference: speedup above $1\times$ means the candidate is both correct and faster, while speedup below $1\times$ means it is correct but slower.

Forecasting Relative Performance. For candidate i, let $x_i = (r, k_i, h)$ denote the tuple of reference kernel implementation, candidate kernel implementation, and hardware specification. Let $S_i = T_{\text{ref}}/T_i$ be its measured speedup over the reference. If the candidate is valid, $S_i > 1$ means the candidate is faster and $S_i < 1$ means it is slower. We implement the *surrogate* as an LLM prompted to take x_i as input, generate a brief chain of reasoning, and forecast the candidate's speedup bin on the device.

Predicting an absolute runtime from source code alone is intractable, as it would require modeling compiler behavior, launch overheads, memory hierarchy effects, and other hardware-dependent execution details. We instead define the prediction target as the candidate's speedup S_i relative to the reference, discretized into eight logarithmic (half-octave) bins (Table 6). Coarse factor-level judgments, such as whether a candidate is much slower, near parity, or much faster, are what a surrogate forecasting from source can realistically make, and they preserve the distinctions a kernel

[2] https://www.gpumode.com/leaderboard/496

optimizer needs. The surrogate emits a probability distribution over the eight bins:

$$p_i = (p_{i,1}, \ldots, p_{i,8}), \qquad p_{i,b} \geq 0, \qquad \sum_{b=1}^{8} p_{i,b} = 1.$$

The predicted bin is $\hat{y}_i = \arg\max_b p_{i,b}$, and the stated confidence is $q_i = \max_b p_{i,b}$. In the structured tool call, this forecast consists of the eight probabilities p_i, the predicted bin $\hat{y}_i$, and a short rationale.

Use in Search. The search uses the surrogate to pre-screen candidates before measurement. At each step the mutator proposes $c \times G$ candidates, where $G \in \mathbb{N}$ is the number of candidates we want to evaluate on the GPU and $c \in \mathbb{N}$ is an overgeneration factor. The surrogate forecasts all $c \times G$ candidates, and the top G candidates with the highest score under an acquisition rule are evaluated on the GPU to obtain their true speedups. We use the simplest rule, the expected speedup under the forecast distribution, $\hat{S}_i = \sum_b p_{i,b}\, m_b$, where m_b is the representative speedup of bin b (Table 6). This is the pre-selection pattern of surrogate-assisted evolutionary optimization (Emmerich et al., 2002; Ulmer et al., 2003; Jin, 2011). A non-standard element is what happens to the $cG - G$ candidates that are not measured. A conventional surrogate-assisted optimizer discards them. We instead keep them in the search archive with their forecast as *speculative scores*: the search treats a speculative score like a measured one when selecting parents to expand, and if the candidate is later measured, the measurement replaces the speculative score. We call a surrogate used this way a speculative surrogate, because the search proceeds speculatively, on a mix of ground-truth speedups and speculative predicted speedups. Because the acquisition rule consumes the full predictive distribution, the calibration of that distribution matters, as a miscalibrated distribution yields a biased expected speedup. We study the problem of improving the quality of the distribution next.

2.2 Reinforcement Learning to Improve Forecasting

We use reinforcement learning (RL) to study how training changes an open-weights surrogate, GPT-OSS-20B (OpenAI, 2025). This forecasting task is a natural fit for reinforcement learning with verifiable rewards. Because the acquisition rule uses the full predictive distribution, we use rewards that score the full distribution in addition to a reward on the argmax bin. We ask the model to reason about a candidate kernel and predict its speedup bin; after the rollout, we can compare that prediction to the measured speedup bin from the GPU. We then reinforce reasoning chains that lead to better forecasts, using rewards defined on the final structured prediction.

The training examples have the same form as the evaluation queries: a reference kernel, a candidate kernel, the target hardware, and the candidate's measured speedup bin relative to the reference. We do not train on failures; every RL example has a successful device measurement and a label in one of the eight speedup bins. Let $y_i \in \{1, \ldots, 8\}$ be the measured speedup bin for training example i.

A rollout is one sampled chain of reasoning that ends in a structured forecast tool call. Let p_i be the probability distribution returned by the forecast schema defined in Section 2.1, and let $\hat{y}_i$ be its predicted bin. If the rollout does not produce a parseable forecast, it receives zero reward.

Rewards. The correctness reward is $r_{\text{corr}} = \mathbf{1}[\hat{y}_i = y_i]$. It rewards the argmax bin and ignores the rest of the distribution. Following prior work that uses RL rewards to train language models to report useful uncertainty (Stengel-Eskin et al., 2024; Damani et al., 2025), we add one of two proper scoring rules that score the full distribution. The Brier score treats the eight bins as distinct classes and compares p_i to a one-hot target. The continuous ranked probability score (CRPS) uses the ordering of the bins and compares the cumulative forecast $F_k(p_i) = \sum_{b \leq k} p_{i,b}$ to the cumulative target, so probability mass one bin from the truth is penalized less than mass many bins away:

$$\text{Brier}(p_i, y_i) = \tfrac{1}{2} \sum_{b=1}^{8} \left(p_{i,b} - \mathbf{1}[b = y_i]\right)^2, \qquad \text{CRPS}(p_i, y_i) = \tfrac{1}{7} \sum_{k=1}^{7} \left(F_k(p_i) - \mathbf{1}[y_i \leq k]\right)^2.$$

Each calibration term is used either alone, as $1 - \text{Brier}$ or $1 - \text{CRPS}$, or added to the correctness reward. Implementation details for the RL runs, including model, training settings, rollout count, decoding settings, and dataset construction, are given in Section H.

3 EXPERIMENTS

3.1 SETUP AND DATA

All kernels in this paper were generated by running LLM-driven evolutionary search. The searches use max-reward tree search with a PUCT selection rule (Section G), the outer loop of TTT-Discover (Yuksekgonul et al., 2026), with a frontier or open-weights LLM as the mutator. Their measured kernels form an archive of 12,388 unique LLM-generated kernels with measured runtimes, which we release. The evaluations below use three subsets of the archive, while the end-to-end searches are described in Section 3.4.

Reference-ranking eval set. We evaluate surrogate forecasts on a set of 480 Triton kernels covering six tasks from the GPU Mode[3] kernel optimization competition: TriMul, cross-entropy, two Gated DeltaNet forward kernels, GDN-W/U recompute, and FP8 quantization[4]. These tasks are realistic workloads with human-engineered reference implementations, and prior LLM-driven kernel search (Yuksekgonul et al., 2026; Cao et al., 2026) has used them for evaluation (Section C). The kernels come from Gemini-3 Flash and Pro searches and are measured on A100 against each task's competition reference. The runtime distribution is approximately uniform across all tasks so that no single performance regime dominates the evaluation. We stratify by speedup bin because the raw output of a kernel search consists mostly of slow and near-parity kernels; without stratification, fast kernels would be underrepresented.

Discovery-moment pairs. The second evaluation uses the unstratified output of real searches. We collect 1,347 parent-child pairs from 27 completed searches, covering 21 problems (the six GPU Mode tasks and 15 KernelBench Level 3 problems) on A100 and L40S GPUs, in both Triton and CUDA, keeping every pair in which both kernels executed correctly. Most children are slower than or at parity with their parents, and fast children are rare.

RL training pool and protocol. For RL training we use successful candidates from GPT-OSS-20B max-reward tree search runs on a single task (TriMul), disjoint from the held-out set. All device measurements use NVIDIA A100 GPUs, except where noted, and we sample three times from each surrogate at temperature 1.0, with tables reporting mean $\pm$ population standard deviation across the three repeats. Full eval-set construction, training-pool construction, and the compute footprint of the underlying searches are deferred to Sections C and H.

3.2 METRICS

We evaluate each surrogate with three metrics. Formal definitions can be found in Section D. **Speedup recovered** asks how close a search would get to the best available kernel if it measured only the candidates the surrogate ranks highest by expected speedup $\hat{S}_i$ (Section 2.1): for a budget c, we take the top $c\%$ of candidates and report the best measured speedup among them as a percentage of the best in the full set, averaged over $c \in \{1, 5, 10, 25, 50\}\%$. **Expected calibration error (ECE)** asks whether the stated confidence q_i matches empirical accuracy, comparing average confidence to exact-bin accuracy over ten confidence buckets (Naeini et al., 2015). **Forecast error** is the mean absolute gap, in $\times$ units, between the representative speedup $m_{\hat{y}_i}$ of the predicted bin and the measured speedup S_i.

3.3 CAN LLMS FORECAST GPU OUTCOMES?

We evaluate off-the-shelf surrogates on two tasks: ranking candidate kernels against a task reference on the balanced eval set, and identifying discovery moments in the unstratified output of real searches.

Ranking against a reference. Table 1 reports the three metrics for four off-the-shelf surrogates. The off-the-shelf GPT-OSS-20B scores 82.9%, which corresponds to finding candidates within 30%

[3] https://www.gpumode.com/home

[4] Despite its name, this task benchmarks a float32 kernel that prepares weights for an FP8 quantization pipeline; it does not execute FP8 operations or use FP8 tensor cores or TMA.

Table 1: **Off-the-shelf LLM surrogates recover most of the available speedup under small GPU-measurement budgets, while calibration varies substantially.** Speedup recovered is averaged over top-ranked 1%–50% GPU-measurement budgets. ECE and forecast error measure calibration and prediction error; lower is better. Cells show mean ± standard deviation across three repeats.

Surrogate	Speedup recovered (%) ↑	ECE ↓	Forecast error ↓
Gemini-3 Flash	93.12 ± 0.97	0.624 ± 0.002	0.421 ± 0.021
GPT-OSS-120B	86.85 ± 2.51	0.479 ± 0.005	0.632 ± 0.029
GPT-OSS-20B	82.88 ± 3.85	0.441 ± 0.011	0.643 ± 0.011
DeepSeek-V4 (DeepSeek-AI, 2026)	79.86 ± 3.98	0.401 ± 0.009	0.805 ± 0.023

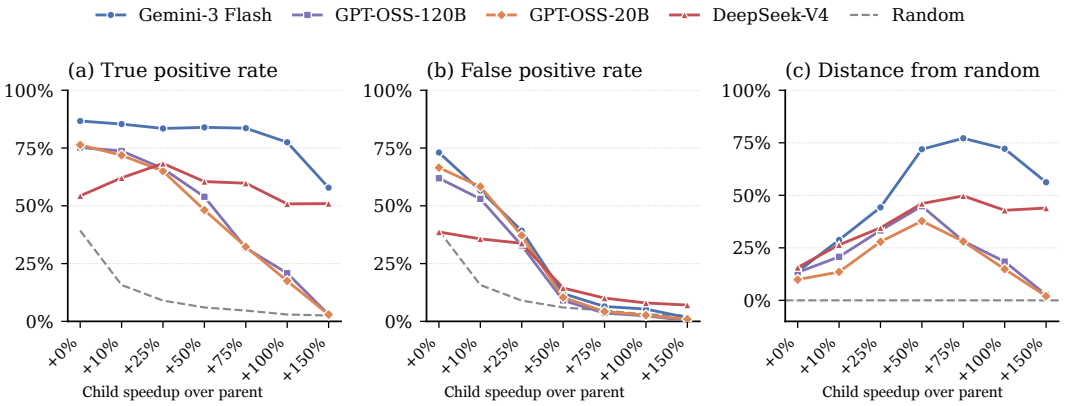

Figure 3: **LLM surrogates detect rare, large speedups over the parent kernel better than chance.** A child is labeled as at least $x\%$ faster when the surrogate's expected speedup of the child over its parent is at least $1 + x$. True positive rate at x is how often the surrogate correctly identifies kernels that are $x\%$ faster than their parents, while false positive rate is how often the surrogate incorrectly identifies a kernel as such. Distance from random is true positive rate minus false positive rate. Rates are computed over three repeats on 1,347 parent-child pairs.

of the true best at a 1% measurement budget and within 6% of it at 50%. Gemini-3 Flash has the best ranking but the worst calibration. Gemini-3 Flash is also a closed model, and API costs make it expensive to use as a surrogate at search scale, which motivates training an open model in Section 3.5.

Discovery moments. How good is the surrogate at telling when a child kernel is much faster than the parent it was mutated from? We call such a step a *discovery moment*. These are the steps that break stagnation in a search, so a surrogate that can identify them could prioritize GPU evaluations for the changes that matter most. This task is harder than ranking against a fixed reference, because the surrogate must judge an incremental improvement over the parent rather than absolute quality, and because most children in the unstratified output of a search are not improvements. For each x from 0% to 150%, we label a child as at least $x\%$ faster than its parent when the surrogate's expected speedup $\hat{S}$ of the child over the parent is at least $1 + x$. Figure 3 reports the true and false positive rates of this labeling (Section E). For Gemini-3 Flash, the true and false positive rates are 84% and 12% at $x = 50\%$, and 78% and 5% at $x = 100\%$. At small x, all four surrogates are near random, because $\hat{S} \geq 1$ for most children, including 39–73% of the children that are actually slower than their parent. For the GPT-OSS models, $\hat{S} \geq 2$ is rare, and the true positive rate at $x = 100\%$ is about 20%. DeepSeek-V4, last in Table 1, has a true positive rate near 50% up to $x = 150\%$ and the second-highest difference between true and false positive rates for $x \geq 10\%$.

Takeaway 1: LLMs can forecast useful GPU outcomes

Off-the-shelf LLMs can rank candidate kernels well enough that measuring only the top-ranked 1%–50% recovers most of the available speedup. They can also identify discovery moments in real search output better than chance, but not precisely enough to skip GPU confirmation.

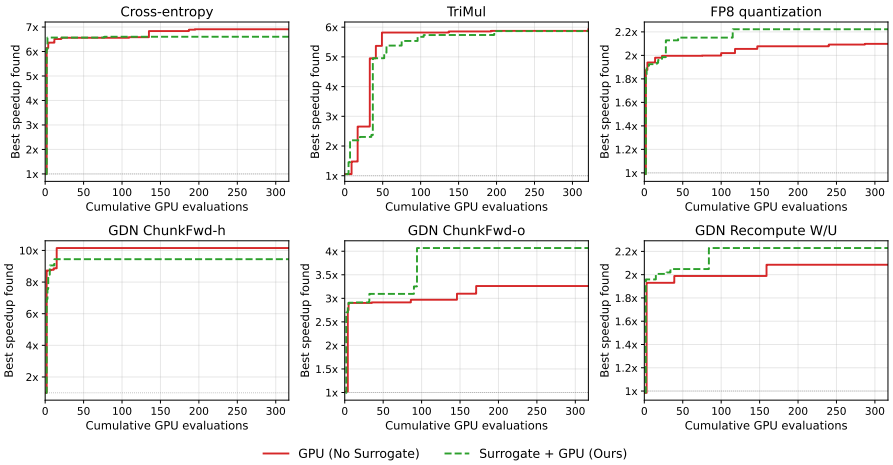

Figure 4: **The search with the surrogate matches or improves on the baseline's best speedup on four of the six tasks, often at substantially fewer GPU measurements, with gaps of 5% and 7% on the remaining two.** Each panel is one task; the y-axis is the best measured speedup so far and the x-axis is the cumulative number of GPU measurements.

3.4 Does pre-screening with a speculative surrogate still find fast kernels?

We compare two PUCT searches (Section G) on the six tasks of Section 3.1, with Gemini-3 Flash as the mutator and about 320 GPU measurements per run. Without the surrogate, the search measures every candidate the mutator proposes. With the surrogate, the search proposes four times as many candidates per parent ($c = 4$, $G = 4$) and measures only the four with the highest expected speedup. Both procedures spend the same number of GPU measurements per step. Each configuration is run once per task, as in prior work on LLM-driven kernel search (Yuksekgonul et al., 2026; Cao et al., 2026), due to the cost of a run. The twelve unmeasured candidates enter the search archive with their forecast as a speculative score (Section 2.1), so the search algorithm has a score for every candidate and does not distinguish forecasts from measurements. Invalid candidates are handled as in the baseline search (Section G). The surrogate is a GPT-OSS-20B model trained with the correctness and Brier rewards (Section 3.5).

Figure 4 plots the best measured speedup against cumulative GPU measurements for each procedure on each task. On three of the six tasks (FP8 quantization, GDN ChunkFwd-o, GDN Recompute W/U), the search with the surrogate finds a faster kernel than the baseline and reaches or exceeds the baseline's best speedup using substantially fewer measurements. On TriMul the two procedures reach essentially the same speedup. On the remaining two tasks the baseline finds a faster kernel. On cross-entropy the gap is about 5%, and on GDN ChunkFwd-h both procedures saturate within the first 15 measurements with the baseline 7% above. The two tasks where the baseline wins are ones where the search appears saturated, as kernels found in the first steps of search are already very close to the final best kernel.

Takeaway 2: The surrogate can select good kernels at a high overgeneration ratio

The surrogate can identify promising kernels even when only 1 in 4 generated candidates is evaluated on the GPU. Search with surrogate selection matches or outperforms GPU-only search on 4 of 6 tasks at the same evaluation budget.

3.5 What happens if you train the surrogate?

We use RL to train GPT-OSS-20B as a surrogate with the rewards of Section 2.2. Training an open model is attractive because a search produces its own training data in the form of measured kernels, and because API-based surrogates are expensive at search scale. We are interested in calibration as well as ranking because the acquisition rule consumes the full predictive distribution. We train on kernels from one task (TriMul) generated by GPT-OSS-20B searches and evaluate on all six

Table 2: **RL training changes calibration more than ranking, and the reward determines whether confidence becomes more or less reliable.** All rows are GPT-OSS-20B scored on the eval set with three repeats (mean ± population standard deviation across repeats). Speedup recovered is reported on all six tasks and on the five tasks not used in training. Δ_{mono} measures how far forecast error deviates from decreasing monotonically in stated confidence (0 is monotone; lower is better; Section I). Best is bold and second best is italic.

Training condition	Speedup recovered (%) ↑		ECE ↓	Δ_{mono} ↓
	All six tasks	Five held-out tasks		
No training	*82.88 ± 3.85*	84.08 ± 2.91	0.441 ± 0.011	0.237
+ correctness	81.59 ± 2.67	83.45 ± 2.33	0.410 ± 0.005	0.52
+ calibration (Brier)	79.12 ± 5.95	80.16 ± 5.17	**0.221 ± 0.015**	0.346
+ calibration (CRPS)	81.45 ± 2.07	*84.07 ± 3.26*	0.358 ± 0.009	**0.054**
+ correctness + calibration (Brier)	**84.94 ± 1.55**	**88.65 ± 2.02**	*0.335 ± 0.012*	*0.088*
+ correctness + calibration (CRPS)	81.83 ± 1.82	81.73 ± 1.50	0.416 ± 0.005	0.121

Table 3: **Forecasting a kernel with an LLM surrogate is faster and cheaper than measuring it on a GPU.** Times are per kernel on one A100. GPU cost is A100 rental time, and surrogate cost is the GPT-OSS-20B token price. Kernel timing requires sole use of the GPU, while the surrogate can serve 16 forecasts at once; the time rows give per-kernel time with one or 16 candidates in flight. Bold gains are GPU values divided by surrogate values. † Values doubled as a safety margin, so the corresponding advantages are conservative.

Triton

Metric	GPU	LLM	**Advantage**
Cost per kernel ($)	0.0140	0.0011	**12**×
Time, 1 in flight (s)	20.2	14.7	**1.4**×
Time, 16 in flight (s)	20.2	1.8†	**11**×

CUDA

Metric	GPU	LLM	**Advantage**
Cost per kernel ($)	0.1059	0.0022†	**47**×
Time, 1 in flight (s)	152.6	29.4†	**5.2**×
Time, 16 in flight (s)	152.6	1.8†	**83**×

tasks using the Gemini-generated eval set, so the other five tasks and all eval kernels are held out. Implementation details are in Section H; Section I contains further analysis, including results on two other backbones.

Table 2 compares the untrained model to each reward. Training changes calibration more than ranking. All calibration rewards lower ECE (by half for Brier alone). Speedup recovered does not change significantly for any reward except Brier alone, which lowers it. The correctness reward is sparse because predicting the exact bin is difficult; with a group size of 8, 56% of training groups receive identical rewards and produce no gradient. Training on correctness alone also makes confidence less reliable, raising Δ_{mono} from 0.237 to 0.52. Correctness combined with Brier is the only reward that improves both calibration and ranking, and its ranking gain is larger on the five held-out tasks than on the full set. All trained variants increase forecast error, which depends only on the argmax bin, because training changes the full distribution rather than the most likely bin (Section I). The same recipe lowers ECE and Δ_{mono} on GPT-OSS-120B and Qwen3-8B (Table 5).

Takeaway 3: Calibration rewards change the surrogate more than correctness rewards

RL with calibration rewards improves the surrogate's calibration without hurting its ranking, while RL with the correctness reward alone makes its confidence less reliable. Combining correctness and Brier rewards improves both calibration and ranking on held-out tasks.

3.6 How much does a forecast cost?

Table 3 compares the cost of a forecast with the cost of a GPU measurement. A single forecast takes 15 seconds, only 1.4× faster than measuring a Triton kernel, but timing a kernel requires sole use of the GPU, while one GPU can serve many forecasts at once. With the 16 candidates of each search step in Section 3.4 forecast together, forecasting is 11× faster and 12× cheaper per Triton kernel,

and $83\times$ faster and $47\times$ cheaper per CUDA kernel, whose measurement is dominated by `nvcc` compilation (Section F).

4 Related Work

Kernel optimization and code-execution reasoning. Several LLM-driven kernel optimization methods rely on per-candidate device measurement as the search signal, including reinforcement-learned agents (Dai et al., 2026), test-time training (Yuksekgonul et al., 2026), and evolutionary loops over strategies (Cao et al., 2026), which adapt the broader paradigm of LLM-driven evolutionary program search (Novikov et al., 2025; Lange et al., 2025; Georgiev et al., 2025; Wang et al., 2025) to kernels. Each measurement involves compilation, correctness checking, and on-device profiling, which together significantly increase the cost of search. Other work reduces this cost through domain-specific languages (Hari et al., 2026; Saba et al., 2026), but every generated candidate is still measured on the device. Neither direction reduces the number of device measurements. We pursue an alternative: studying whether the GPU can be replaced with a surrogate. This relates to work on the simulatability of code by LLMs, which has examined whether LLMs can predict program input-output behavior (Gu et al., 2024), simulate step-by-step Python execution (FAIR CodeGen Team, 2025), simulate unit test outputs (Prasad et al., 2025), and count CUDA kernel FLOPs (Bolet et al., 2025) from source. These results suggest that the same capability could allow LLMs to act as surrogates of physical devices.

Surrogate-assisted optimization and learned cost models. Surrogate-assisted optimization uses a learned predictor as a stand-in for an expensive objective during search (Jones et al., 1998). In surrogate-assisted evolutionary algorithms, the surrogate pre-screens offspring so that only the most promising are evaluated on the true objective (Emmerich et al., 2002; Ulmer et al., 2003; Jin, 2011), and Bayesian optimization applies an acquisition rule to a probabilistic surrogate's predictive distribution to choose the next evaluation (Snoek et al., 2012; Shahriari et al., 2016). Compiler autotuners use learned cost models that predict the runtime of a candidate program or schedule from its features and rank candidates for measurement (Chen et al., 2018; Zheng et al., 2020; Zhai et al., 2023). Our surrogate plays the same role, and its forecasts also stand in for measurements inside the search. Best-of-N selection with a learned reward model or verifier in LLM post-training uses a scorer in a similar way (Cobbe et al., 2021; Lightman et al., 2024). Model-based reinforcement learning learns a predictive model of an environment from interaction (Sutton, 1991; Ha & Schmidhuber, 2018; Schrittwieser et al., 2020; Hafner et al., 2025), and recent work asks whether LLMs can learn such models from limited interaction in grid and game environments (Warrier et al., 2025; Piriyakulkij et al., 2025; Khan et al., 2026).

LLM forecasting and calibration training. LLMs have been evaluated as forecasters of international events (Ye et al., 2024), prediction-market resolutions (Yang et al., 2025), and open-ended forecasting questions about current events (Chandak et al., 2025). These results show that LLMs carry world knowledge applicable to complex predictions. Recent work augments RL correctness rewards with a proper scoring rule to train LLMs for calibration (Damani et al., 2025), since LLMs are not by default well-calibrated.

5 Conclusion

GPU measurements are the ground-truth signal for kernel optimization, but they are also a major cost of search. We studied whether LLMs can reduce this cost as speculative surrogates of a physical GPU: forecasting a distribution over a candidate's speedup, selecting which candidates the device measures, and standing in for measurements until they are made. Off-the-shelf LLMs rank candidates well enough to recover most of the available speedup under small measurement budgets, a search that measures 1 in 4 generated candidates matches or outperforms GPU-only search on 4 of 6 tasks, and RL with calibration rewards improves a surrogate's calibration without hurting its ranking. These results suggest a practical path toward kernel optimizers that treat hardware measurement as a scarce resource.

AI USE

We used generative AI tools to generate synthetic data in the form of candidate GPU kernels, implement research methods, and support qualitative and thematic data analysis. We did not use these tools to develop theoretical models or conceptual frameworks, formulate mathematical claims or proofs, propose or refine hypotheses, design research methodology or experiments, translate material, clean or reformat datasets, or interpret results. We also used generative AI tools to modify scientific figures, edit software code, and draft parts of the manuscript. The authors reviewed all AI-assisted work and take responsibility for the final text, claims, code, figures, and analyses.

REPRODUCIBILITY

Sections 2.1 and 3 describe the forecasting task, search setup, and evaluation metrics. The appendices document evaluation-set and training-data construction (Section C), the search algorithm (Section G), surrogate training and scoring (Section H), prompts (Section K), and the GPU measurement protocol, benchmark cases, and hyperparameters (Section J). The code used for kernel search, surrogate training and evaluation, and the measured kernel dataset are provided in the supplementary materials.

ETHICS

This work studies the performance of generated GPU kernels using benchmark measurements. It does not involve human participants, personal data, or sensitive datasets, and we are not aware of any specific ethical concerns arising from the methods or results.

REFERENCES

Josh Abramson, Jonas Adler, Jack Dunger, Richard Evans, Tim Green, Alexander Pritzel, Olaf Ronneberger, Lindsay Willmore, Andrew J. Ballard, Joshua Bambrick, et al. Accurate structure prediction of biomolecular interactions with AlphaFold 3. *Nature*, 630:493–500, 2024. doi: 10.1038/s41586-024-07487-w.

Gregory Bolet, Giorgis Georgakoudis, Konstantinos Parasyris, Harshitha Menon, Niranjan Hasabnis, Kirk W. Cameron, and Gal Oren. Counting without running: Evaluating LLMs' reasoning about code complexity. *arXiv preprint arXiv:2512.04355*, 2025.

Shiyi Cao, Ziming Mao, Joseph E. Gonzalez, and Ion Stoica. K-search: Llm kernel generation via co-evolving intrinsic world model. *arXiv preprint arXiv:2602.19128*, 2026.

Nikhil Chandak, Shashwat Goel, Ameya Prabhu, Moritz Hardt, and Jonas Geiping. Scaling open-ended reasoning to predict the future. *arXiv preprint arXiv:2512.25070*, 2025.

Terry Chen, Zhifan Ye, Bing Xu, Zihao Ye, Timmy Liu, Ali Hassani, Tianqi Chen, Andrew Kerr, Haicheng Wu, Yang Xu, Yu-Jung Chen, Hanfeng Chen, Aditya Kane, Ronny Krashinsky, Ming-Yu Liu, Vinod Grover, Luis Ceze, Roger Bringmann, John Tran, Wei Liu, Fung Xie, Michael Lightstone, and Humphrey Shi. Avo: Agentic variation operators for autonomous evolutionary search, 2026. URL https://arxiv.org/abs/2603.24517.

Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, 2018.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Weinan Dai, Hanlin Wu, Qiying Yu, Huan-ang Gao, Jiahao Li, Chengquan Jiang, Weiqiang Lou, Yufan Song, Hongli Yu, Jiaze Chen, Wei-Ying Ma, Ya-Qin Zhang, Jingjing Liu, Mingxuan Wang, Xin Liu, and Hao Zhou. Cuda agent: Large-scale agentic rl for high-performance cuda kernel generation. *arXiv preprint arXiv:2602.24286*, 2026.

Mehul Damani, Isha Puri, Stewart Slocum, Idan Shenfeld, Leshem Choshen, Yoon Kim, and Jacob Andreas. Beyond binary rewards: Training LMs to reason about their uncertainty. *arXiv preprint arXiv:2507.16806*, 2025.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

Michael Emmerich, Alexios Giotis, Mutlu Özdemir, Thomas Bäck, and Kyriakos Giannakoglou. Metamodel-assisted evolution strategies. In *Parallel Problem Solving from Nature (PPSN VII)*, pp. 361–370, 2002.

Meta FAIR CodeGen Team. Cwm: An open-weights llm for research on code generation with world models, 2025. URL https://arxiv.org/abs/2510.02387.

Bogdan Georgiev, Javier Gómez-Serrano, Terence Tao, and Adam Zsolt Wagner. Mathematical exploration and discovery at scale. *arXiv preprint arXiv:2511.02864*, 2025.

Google DeepMind. Gemini 3 pro model card, 2025. URL https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Pro-Model-Card.pdf.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. CRUXEval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*, 2024.

David Ha and Jürgen Schmidhuber. Recurrent world models facilitate policy evolution. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. Mastering diverse control tasks through world models. *Nature*, 640:647–653, 2025. doi: 10.1038/s41586-025-08744-2.

Siva Kumar Sastry Hari, Vignesh Balaji, Sana Damani, Qijing Huang, and Christos Kozyrakis. Improving efficiency of GPU kernel optimization agents using a domain-specific language and speed-of-light guidance. *arXiv preprint arXiv:2603.29010*, 2026.

Yaochu Jin. Surrogate-assisted evolutionary computation: Recent advances and future challenges. *Swarm and Evolutionary Computation*, 1(2):61–70, 2011.

Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, 13:455–492, 1998. doi: 10.1023/A:1008306431147.

Zaid Khan, Archiki Prasad, Elias Stengel-Eskin, Jaemin Cho, and Mohit Bansal. One life to learn: Inferring symbolic world models for stochastic environments from unguided exploration. In *International Conference on Learning Representations (ICLR)*, 2026.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution. *arXiv preprint arXiv:2509.19349*, 2025.

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. In *International Conference on Learning Representations*, 2024.

Mahdi Pakdaman Naeini, Gregory Cooper, and Milos Hauskrecht. Obtaining well calibrated probabilities using bayesian binning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 29, 2015.

Alexander Novikov et al. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

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?, 2025. URL https://arxiv.org/abs/2502.10517.

Wasu Top Piriyakulkij, Yichao Liang, Hao Tang, Adrian Weller, Marta Kryven, and Kevin Ellis. PoE-World: Compositional world modeling with products of programmatic experts. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Archiki Prasad, Elias Stengel-Eskin, Justin Chih-Yao Chen, Zaid Khan, and Mohit Bansal. Learning to generate unit tests for automated debugging. In *Second Conference on Language Modeling*, 2025.

Christopher D. Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61:203–230, 2011. doi: 10.1007/s10472-011-9258-6.

Tara Saba, Zhiyang Chen, Jikai Jason Li, Anne Ouyang, Xujie Si, and Fan Long. CuTeGen: An LLM-based agentic framework for generation and optimization of high-performance GPU kernels using CuTe. *arXiv preprint arXiv:2604.01489*, 2026.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588: 604–609, 2020. doi: 10.1038/s41586-020-03051-4.

Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. Taking the human out of the loop: A review of Bayesian optimization. *Proceedings of the IEEE*, 104(1): 148–175, 2016.

Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical Bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, 2012.

Elias Stengel-Eskin, Peter Hase, and Mohit Bansal. LACIE: Listener-aware finetuning for confidence calibration in large language models. *Advances in Neural Information Processing Systems*, 37: 43080–43106, 2024.

Richard S. Sutton. Dyna, an integrated architecture for learning, planning, and reacting. *ACM SIGART Bulletin*, 2(4):160–163, 1991.

Holger Ulmer, Felix Streichert, and Andreas Zell. Evolution strategies assisted by Gaussian processes with improved pre-selection criterion. In *Congress on Evolutionary Computation (CEC)*, pp. 692–699, 2003.

Yiping Wang, Shao-Rong Su, Zhiyuan Zeng, Eva Xu, Liliang Ren, Xinyu Yang, Zeyi Huang, Xuehai He, Luyao Ma, Baolin Peng, Hao Cheng, Pengcheng He, Weizhu Chen, Shuohang Wang, Simon Shaolei Du, and Yelong Shen. Thetaevolve: Test-time learning on open problems. *arXiv preprint 2511.23473*, 2025.

Archana Warrier, Dat Nguyen, Michelangelo Naim, Moksh Jain, Yichao Liang, Karen Schroeder, Cambridge Yang, Joshua B. Tenenbaum, Sebastian Vollmer, Kevin Ellis, and Zenna Tavares. Benchmarking world-model learning with environment-level queries, 2025. URL https://arxiv.org/abs/2510.19788.

Qingchuan Yang, Simon Mahns, Sida Li, Anri Gu, Jibang Wu, and Haifeng Xu. LLM-as-a-prophet: Understanding predictive intelligence with Prophet Arena. *arXiv preprint arXiv:2510.17638*, 2025.

Chenchen Ye, Ziniu Hu, Yihe Deng, Zijie Huang, Mingyu Derek Ma, Yanqiao Zhu, and Wei Wang. MIRAI: Evaluating LLM agents for event forecasting. *arXiv preprint arXiv:2407.01231*, 2024.

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

Yi Zhai, Yu Zhang, Shuo Liu, Xiaomeng Chu, Jie Peng, Jianmin Ji, and Yanyong Zhang. TLP: A deep learning-based cost model for tensor program tuning. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 833–845, 2023.

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 *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 863–879, 2020.

A EXAMPLE EVAL-SET KERNELS

This appendix shows two kernels drawn from the FP8 per-token-group quantization eval set, one near the lower end of the speedup distribution and one near the upper end. Both kernels target the same operation against the same A100 reference implementation, so the comparison isolates implementation quality rather than task or hardware difficulty. We selected these examples from the held-out eval set described in Section C as representative cases that a surrogate is asked to forecast.

Low-speedup candidate ($S_i = 0.996$). Box A.1 reshapes the input into per-token groups, computes the per-group absolute maximum, derives the scale, and writes the quantized values back through a sequence of unfused PyTorch tensor operations. Each step launches its own kernel and materializes an intermediate tensor, so per-element memory traffic dominates and the candidate runs slightly slower than the reference. The implementation is correct and otherwise unremarkable, which makes it a representative example of the naive end of what an LLM proposes before any fusion or layout work.

High-speedup candidate ($S_i = 1.970$). Box A.2 fuses the absmax reduction, scale derivation, and clamped quantization into a single Triton launch. The host-side wrapper picks `BLOCK_GROUPS` so that the dispatch grid is deep enough to saturate the streaming multiprocessors of an A100 while bounding per-thread register pressure, and it precomputes the reciprocal of the per-group scale so the elementwise quantization issues a multiplication rather than a division. The reduction over each group is performed inside registers without a shared-memory round trip. Compared to the low-speedup candidate, this implementation removes both the unfused intermediate tensors and the per-step launch overhead, which accounts for the measured $1.97\times$ speedup over the reference.

Box A.1 | Low-speedup FP8 quantization candidate

Measured speedup $S_i = 0.996$ on A100.

```
import torch


FP8_MAX = 448.0
FP8_MIN = -448.0
FP8_EPS = 1e-10


def custom_kernel(data):
    """Pure-PyTorch per-token-group FP8 quantization.

    Args:
        data: tuple ``(x, x_q, x_s)`` where
            - x:   [num_tokens, hidden_dim]                float32
             ↪  on CUDA --- input
            - x_q: [num_tokens, hidden_dim]                float32
             ↪  on CUDA --- pre-allocated output buffer
            - x_s: [num_tokens, hidden_dim // group_size]  float32
             ↪  on CUDA --- pre-allocated scale buffer

    Returns:
        Tuple ``(x_q, x_s)`` with the same buffers populated
         ↪  in-place.
    """
    x, x_q, x_s = data
    num_tokens, hidden_dim = x.shape
    num_groups = x_s.shape[1]
```

```
    group_size = hidden_dim // num_groups

    x_f32 = x.float()
    x_grouped = x_f32.reshape(num_tokens, num_groups, group_size)

    absmax = x_grouped.abs().amax(dim=-1).clamp(min=FP8_EPS)
    scale = absmax / FP8_MAX

    quantized = (x_grouped / scale.unsqueeze(-1)).clamp(FP8_MIN,
    ↪  FP8_MAX)
    quantized = quantized.reshape(num_tokens, hidden_dim)

    x_q[...] = quantized
    x_s[...] = scale
    return x_q, x_s
```

Box A.2| High-speedup FP8 quantization candidate

Measured speedup $S_i = 1.970$ on A100.

```
"""
Optimized Fused Per-Token-Group FP8 (E4M3) Quantization Kernel.

Algorithm:
1. Translates the 2D tensor `(num_tokens, hidden_dim)` into a flat
↪  1D grid of `total_groups`.
2. Dynamically adjusts `BLOCK_GROUPS` strictly to balance SM wave
↪  occupancy and register limit.
   - Targeting >= 512 blocks ensures massive wave overlap on A100's
   ↪  108 SMs to natively hide DRAM latency.
   - Capping `BLOCK_GROUPS` at 128 guarantees <= 64 elements per
   ↪  thread, completely eliminating
     register spilling which otherwise drastically hurts
     ↪  performance on large shapes.
3. Groups are processed entirely within registers natively
↪  (intra-warp reduction), bypassing
   shared memory round-trips.
4. Drops PTX masking globally when the dispatch grid perfectly
↪  overlaps the tensor boundary.
"""

import torch
import triton
import triton.language as tl

@triton.jit
def fp8_quantize_kernel(
    x_ptr, x_q_ptr, x_s_ptr,
    total_groups,
    GROUP_SIZE: tl.constexpr,
    BLOCK_GROUPS: tl.constexpr,
    NEEDS_MASK: tl.constexpr,
):
    pid = tl.program_id(0)
    group_start = pid * BLOCK_GROUPS

    group_offsets = group_start + tl.arange(0, BLOCK_GROUPS)
    elem_offsets = tl.arange(0, GROUP_SIZE)

    # [BLOCK_GROUPS, GROUP_SIZE] layout ensures vectorized load
    ↪  instructions (ld.global.v4)
    offsets = group_offsets[:, None] * GROUP_SIZE +
    ↪  elem_offsets[None, :]
```

```
    if NEEDS_MASK:
        mask_1d = group_offsets < total_groups
        mask = mask_1d[:, None]
        x = tl.load(x_ptr + offsets, mask=mask, other=0.0)
    else:
        x = tl.load(x_ptr + offsets)

    # Intra-warp reduction over elements (no shared memory needed
    ↪  as GROUP_SIZE fits in local thread registers)
    absmax = tl.max(tl.abs(x), axis=1)
    absmax = tl.maximum(absmax, 1e-10)

    scale = absmax / 448.0
    inv_scale = 448.0 / absmax

    x_q = x * inv_scale[:, None]
    x_q = tl.minimum(tl.maximum(x_q, -448.0), 448.0)

    if NEEDS_MASK:
        tl.store(x_s_ptr + group_offsets, scale, mask=mask_1d)
        tl.store(x_q_ptr + offsets, x_q, mask=mask)
    else:
        tl.store(x_s_ptr + group_offsets, scale)
        tl.store(x_q_ptr + offsets, x_q)


def custom_kernel(data):
    x, x_q, x_s = data

    num_tokens, hidden_dim = x.shape
    num_groups = x_s.shape[1]
    group_size = hidden_dim // num_groups
    total_groups = num_tokens * num_groups

    # Cap BLOCK_GROUPS at 128. For a group_size of 128, this
    ↪  equates to 16,384 elements per block.
    # Distributed over 8 warps (256 threads), each thread holds 64
    ↪  elements, sitting safely
    # below the A100 register bottleneck limits.
    BLOCK_GROUPS = 128

    # Scale down linearly for smaller dimensions to guarantee a
    ↪  deep dispatch grid.
    # Target ~512 blocks to allow ~4-5 wave parallelism across the
    ↪  108 SMs.
    while BLOCK_GROUPS > 16 and (total_groups // BLOCK_GROUPS) <
    ↪  512:
        BLOCK_GROUPS //= 2

    if BLOCK_GROUPS >= 128:
        num_warps = 8
    else:
        num_warps = 4

    grid = ((total_groups + BLOCK_GROUPS - 1) // BLOCK_GROUPS, )
    NEEDS_MASK = (total_groups % BLOCK_GROUPS) != 0

    fp8_quantize_kernel[grid](
        x, x_q, x_s,
        total_groups,
        GROUP_SIZE=group_size,
        BLOCK_GROUPS=BLOCK_GROUPS,
        NEEDS_MASK=NEEDS_MASK,
```

```
        num_warps=num_warps
    )

    return x_q, x_s
```

B SUCCESS CASES AND FAILURE CASES

We walk through two reasoning traces from the untrained surrogate on TriMul prompts. The two candidates make similar structural changes to the reference (a candidate-side rewrite of the core triangle product, with bfloat16 or float16 cast). The measured speedups are very different: the first candidate runs $1.71\times$ faster than the reference, the second runs at parity with the reference ($1.02\times$). The grounded trace places the first in the minor-speedup bin, one bin below its true bin (significant speedup). The poorly-grounded trace mistakes the second for a more-than-$4\times$ speedup.

Grounded trace, candidate at $1.71\times$ measured speedup, predicted bin 5. The reference applies separate float32 `nn.Linear` projections, gating sigmoids, and a bfloat16 `einsum` for the triangle product. The candidate folds the projections, gating, and product into a float16 path that reuses tensor cores end to end. The model's full reasoning trace is reproduced here verbatim:

> *We need to analyze speedup of candidate relative to reference. The reference uses torch layers: nn.Linear, LayerNorm etc. The torch layers under the hood likely use cuBLAS GEMMs for linear operations; the reference uses float32 operations. The candidate uses FP16 for most computations, with some cast overhead. It fuses ops etc. [...] Reference utilizes torch einsum for core TriMul:* `einsum('... i k d, ... j k d -> ... i j d', left.to(torch.bfloat16), right.to(torch.bfloat16))`*. Two input projected to shape $[B, N, N, H]$ then einsum over k dimension (H). That is a batched matrix multiplication: the same dimension as left and right projecting; similar to the candidate's torch.matmul of permuted, I'm not sure if einsum performance similar or better. Reference uses bfloat16 conversions. Candidate uses half. On A100, tensor cores support fp16 and bf16, with similar speeds? For large matrices maybe BF16 might be slightly faster (because of memory bandwidth? Actually A100 can handle BF16 and FP16 in Tensor Cores identical). Candidate uses half for core multiplication too; could yield similar speed. [...] Overall, candidate likely faster: maybe $1.2\times$–$1.4\times$? On A100 fp16 can be up to $2\times$ faster than fp32 for GEMMs. But since reference uses bf16 (which also uses Tensor Cores for FP32? Actually A100 supports BF16 in Tensor Cores; it's similar hardware). But reference also uses dtype float32 for LayerNorm etc. [...] Thus likely candidate has FI moderate speedup: ~ 1.3. This puts in bin 5 (minor speedup).*

The trace is grounded along three lines that each match the actual hardware behavior on A100. The model recognizes that PyTorch's `einsum` on a four-dimensional contraction dispatches to a tensor-core matmul, not to a generic semiring kernel. It correctly identifies that bfloat16 and float16 have similar tensor-core throughput on A100. It realizes the candidate's win comes from running the surrounding LayerNorm and projections in float16 rather than float32, not from the core triangle product itself. The numerical estimate that follows ($\sim 1.3\times$) is conservative against the measured $1.71\times$, which reflects model uncertainty rather than incorrect reasoning, and the predicted bin (5) is one below the true bin (6).

Poorly-grounded trace, candidate at $1.02\times$ measured speedup, predicted bin 8. A different candidate on a similar TriMul prompt rewrites the bfloat16 `einsum` as a permuted batched `torch.matmul`, leaving the surrounding projections in float32 as in the reference. The measured candidate runtime is essentially equal to the reference's, but the model places probability 0.65 on bins 6, 7, and 8 (more than $1.41\times$ speedup), with the argmax on bin 8 at probability 0.25 and an additional 0.15 on bin 5, the true bin. The full reasoning trace:

> *We have to compare reference kernel, which uses PyTorch operations including many module calls (Linear, LayerNorm, einstein summation). The candidate kernel rewires inner einsum into batched matmul which is efficient. Reference uses einsum which is less efficient, older baseline is naive and may be not optimized by cuBLAS, but PyTorch's einsum uses semiring operations maybe less efficient. So candidate likely much faster. How many bits/flops? For large seq length $N \geq$? Provided config unspecified but typical large can be up to 1024. Implementation rewriting ensures using cuBLAS batched gemms. So expect high speedup maybe $10\times$? But we have to pick bins up to extreme $> 4\times$. So candidate in bin 8. [...] So reference einsum is less efficient than cuBLAS batched matmul. Also candidate uses torch.matmul which uses batched gemm; it's optimized. So speedup maybe moderate to high? Suppose reference $4\times$ slower? Might be $> 4\times$, but maybe $8\times$? Bin 8 (extreme) plausible. [...] Confidence high that candidate faster. Without measuring, but we expect $> 4\times$ due to einsum inefficiency.*

The reasoning collapses on a prior that PyTorch's `einsum` "uses semiring operations maybe less efficient" than `cuBLAS batched gemm`. This prior is not supported by the source code in front of the model and is also not supported by the runtime. The contraction pattern `... i k d, ... j k d -> ... i j d` is exactly a batched matrix multiplication, and modern PyTorch's `einsum` dispatches it to the same cuBLAS path that the candidate's `torch.matmul` does. Once the model commits to the einsum-is-slow prior it stops looking for confirming evidence. The trace fabricates a sequence-length default of $N{=}1024$ to motivate a FLOP estimate, declares the candidate "maybe $8\times$" faster, and exits to a bin-8 argmax at 0.25 probability with most of the remaining mass on bins 6 and 7 rather than on the true bin 5. The candidate is in fact at parity with the reference, so the prediction is wrong by three bins, and the source of the error is fully visible in the CoT but not in the short rationale string the surrogate emits to the tool call.

The two traces share the same vocabulary, the same overall structure, and even the same uncertainty about how PyTorch dispatches `einsum`. The grounded trace resolves the uncertainty by referencing a checkable property of the hardware (BF16 and FP16 have the same A100 tensor-core throughput) and arrives at a calibrated speedup estimate. The poorly-grounded trace resolves the same uncertainty by appealing to an unverified prior ("einsum maybe semiring, less efficient") and arrives at a three-bin error.

C Eval-Set Construction and Dataset Compute Footprint

Task selection. The six GPU Mode tasks (TriMul, cross-entropy, two Gated DeltaNet forward kernels, GDN-W/U recompute, FP8 quantization) were selected to span operations that appear in current production ML workloads and that have published, human-tuned reference implementations. Two considerations drove the choice. First, prior LLM-driven kernel search work (Yuksekgonul et al., 2026; Cao et al., 2026) has used the same competition as its evaluation surface, so the operations have an established record as challenging targets for automated kernel optimization. Second, each task is compositionally non-trivial: it consists of multiple memory and compute stages whose fusion patterns and tile choices materially affect runtime. We deliberately go deep on a small number of tasks, sweeping a wide range of generated speedups per task, rather than wide across many simpler kernels, so that a surrogate's behavior can be characterized across the full range of speedup outcomes a real kernel optimizer will encounter.

Eval-set assembly. For each task, we run the max-reward tree search procedure described in Section G using Gemini-3 Flash and Gemini-3 Pro (Google DeepMind, 2025) as the mutation provider, with the search reward shaped to drive coverage across all eight speedup bins rather than only the fastest tier. We harvest every successful candidate from each search, measure its runtime against the task's competition reference on an A100, and assign it to one of the eight speedup bins from Table 6. We then cap the number of kernels per bin per task to enforce uniform coverage so heavily explored bins do not dominate the loaded view. The resulting eval set contains 480 kernels: six tasks $\times$ eight bins $\times$ up to ten kernels per bin, with bins approximately uniformly populated subject to availability of correct candidates in each bin.

Training-pool construction. The RL training pool is sourced from six GPT-OSS-20B max-reward tree search runs (Section G), one per task, run on A100 hardware and disjoint from the searches that produced the eval set. We harvest every successful candidate from those six traces, then a pair-construction stage emits three kinds of comparisons used as training rows: *seed-anchored* (each candidate against the task reference), *parent-edit* (each candidate against its parent in the search tree, when both are correct), and *within-pack* (sampled correct-against-correct pairs within a task, stratified by log-speedup magnitude). Failed candidates do not contribute training rows, and rows are deduplicated by anchor and candidate code with priority seed > parent-edit > within-pack. Further training-procedure details are in Section H.

Compute footprint of the underlying searches. The kernels in this paper are drawn from a larger archive of 12,388 unique LLM-generated GPU kernels with measured runtimes that we release alongside the paper. The full archive consumed approximately 411M LLM tokens for candidate generation and 618 GPU-hours of A100, L40S, and H100 measurement across all problems and backends. The portion supplying the GPU Mode held-out eval set used roughly 47 A100-hours of measurement and 62M LLM tokens; the six search runs supplying the RL training pool sit within this archive and represent a small fraction of that total.

D Metric Definitions

This appendix gives the formal definitions of the metrics of Section 3.2.

Speedup recovered. For candidate i, the forecast distribution p_i defines an expected speedup score $\hat{S}_i$, and for a measurement budget c we take the top $c\%$ of candidates under this score and report the best measured speedup in that subset as a percentage of the best measured speedup in the full set, averaged over budgets $\mathcal{C} = \{1\%, 5\%, 10\%, 25\%, 50\%\}$. This can also be thought of as a measure of regret: when ranking kernels and selecting the top $c\%$, how much worse is the best kernel in that top $c\%$ than the best kernel present in the superset of candidates.

$$\hat{S}_i = \sum_{b=1}^{8} p_{i,b}\, m_b, \qquad \text{Recovered} = \frac{100}{|\mathcal{C}|} \sum_{c \in \mathcal{C}} \frac{\max_{i \in \text{Top}_c(\hat{S})} S_i}{\max_i S_i}.$$

Here m_b is the representative speedup assigned to bin b, defined in Table 6.

Expected Calibration Error (ECE). Confidence is $q_i = \max_b p_{i,b}$. We sort predictions into ten confidence buckets following Naeini et al. (2015); within each bucket, $\text{conf}(B_m)$ is the average confidence and $\text{acc}(B_m)$ is the fraction of predictions whose speedup bin exactly matches the measured bin.

$$\text{ECE} = \sum_{m=1}^{10} \frac{|B_m|}{n} \left|\text{acc}(B_m) - \text{conf}(B_m)\right|.$$

Forecast error.

$$\text{ForecastError} = \frac{1}{n} \sum_{i=1}^{n} \left|m_{\hat{y}_i} - S_i\right|.$$

E Discovery-Moment Evaluation Protocol

The surrogate takes the code of two kernels, K and K', and predicts a distribution over their speedup $g = t(K)/t(K')$, where $t(\cdot)$ is the measured runtime and $g > 1$ means K' is faster. In the reference-ranking evaluation, K was the task reference and K' a candidate kernel. Here K is the parent and K' is the child of a mutation step. For a threshold x, the ground-truth label y is 1 when $g \geq 1 + x$ and 0 otherwise. Pairs with $y = 0$ include steps that change nothing and steps that make the kernel slower. The predicted label $\hat{y}$ is 1 when the expected speedup $\hat{S}$ (Section 3.2) is at least $1 + x$. A pair the surrogate fails to score has $\hat{y} = 0$. For example, at $x = 100\%$, a child that runs $2.3\times$ faster than its parent has $y = 1$, and it has $\hat{y} = 1$ when the surrogate's expected speedup for the pair is at least 2. The true positive rate is the fraction of pairs with $y = 1$ that have $\hat{y} = 1$, and the false positive rate is the fraction of pairs with $y = 0$ that have $\hat{y} = 1$. We use $x \in \{0\%, 10\%, 25\%, 50\%, 75\%, 100\%, 150\%\}$ and pool pairs over three repeats. Figure 3 reports both rates and their difference at each threshold.

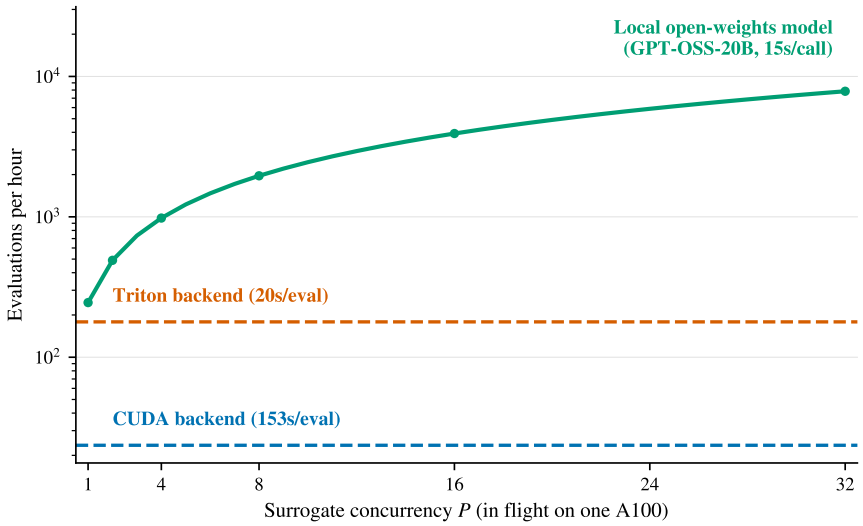

Figure 5: **Evaluations per hour on a single A100 for the three evaluation modes used in this paper.** The two dashed lines are the Triton and CUDA kernel benchmarks, fixed at one in flight because timing a kernel against the reference requires sole occupancy of the GPU. The solid line is the LLM surrogate (GPT-OSS-20B), which has no such constraint and serves multiple forecasts concurrently from the same GPU, so its throughput grows with the number P of forecasts in flight. Latencies in parentheses are mean per-call wall clock measured during the runs that produced this paper's results.

F Per-Evaluation Latency and Throughput on a Single A100

This appendix reports the per-evaluation latency that motivates the surrogate as a low-latency alternative to on-device measurement, and converts that latency into evaluations per hour on a single A100. Figure 5 summarizes the three evaluation modes used in the paper.

Measurement procedure. For each of the three evaluation modes, we recorded the wall clock from the moment a candidate is submitted to the moment its outcome (a measured speedup for the kernel benchmark, a forecast distribution for the surrogate) is returned to the search procedure. All measurements were taken on a single A100, using the same evaluator implementations the paper uses everywhere else, so the reported latency reflects production conditions rather than a microbenchmark. We then took the mean across all evaluations performed in the relevant runs, and converted to evaluations per hour by computing $3600 \cdot P/\mathrm{latency}$, where P is the number of evaluations the device serves concurrently.

Three evaluation modes. The Triton kernel benchmark compiles each candidate, checks its outputs against the reference, and times it against the reference on the device. Its mean per-call latency is 20.16 seconds, averaged over $1{,}515$ evaluations performed across the six GPU Mode kernel-search runs in this paper. The CUDA kernel benchmark performs the same compile, check, and time steps for CUDA kernels. Its mean per-call latency is 152.55 seconds, averaged over $6{,}446$ evaluations performed across the eleven KernelBench Level 3 search runs in this paper, and is dominated by `nvcc` compilation rather than on-device execution. The LLM surrogate runs GPT-OSS-20B locally and returns a calibrated forecast distribution over speedup bins together with a brief chain of reasoning. Its mean per-call latency is 14.69 seconds, averaged across the surrogate-scoring run that produced the headline open-weights numbers in the main text.

Concurrency and throughput. Throughput depends on how many evaluations a single A100 can serve concurrently. Timing a kernel against a reference is a measurement of execution time on the device, and is only meaningful when no other workload competes for the GPU, so the kernel

benchmark is restricted to one evaluation in flight at a time. The surrogate has no such requirement: it is a forward pass of an LLM, and the model server hosting GPT-OSS-20B can hold many forecasts in flight simultaneously on the same GPU, so its throughput scales close to linearly with the number of concurrent forecasts P until the device saturates. Even at $P{=}1$, the surrogate already produces more evaluations per hour than the Triton benchmark, and at modest concurrencies it produces between one and two orders of magnitude more evaluations per hour than the CUDA benchmark. This is the latency gap that the surrogate exploits: when a forecast is good enough to act on, the search can replace a costly on-device measurement with a much cheaper LLM call, and reserve on-device measurement for the candidates whose forecasts are not trustworthy enough to act on.

G KERNEL SEARCH VIA MAX-REWARD TREE SEARCH

The runs that produced the held-out eval set and the RL training pool used in this paper share the same outer search procedure: max-reward tree search with a PUCT selection rule (Rosin, 2011) over an archive of correct candidates. We refer to it as *max-reward tree search* below, since the backup is over the maximum descendant reward rather than the mean as in textbook MCTS, and PUCT names only the selection score the search uses to choose which archive node to expand next. This is the same outer loop used by TTT-Discover (Yuksekgonul et al., 2026) for kernel optimization. We retain the search structure and replace TTT-Discover's RL-trained mutation policy with a frontier or open-weights LLM that generates kernel edits from a fixed prompt, so the search behavior is determined by the base model rather than by an additionally trained policy. The broader kernel archive we release alongside the paper includes runs that used other generation procedures (e.g. agentic CLI loops and exploratory variants); those runs are not used in the experiments reported here.

State and statistics. The search maintains an archive of correct candidate kernels, each with a measured reward (the candidate's speedup over the task reference). For every node s in the archive, the search tracks a visit count $n(s)$, the best reward observed across s's children $m(s)$, and a global expansion counter T.

Selection score. At each step, every archive node is scored by

$$\text{score}(s) = Q(s) + c_{\text{puct}}\,\sigma(\mathcal{A})\,P(s)\,\frac{\sqrt{1+T}}{1+n(s)},$$

where $Q(s) = m(s)$ if s has been expanded and the candidate's own reward otherwise, $\sigma(\mathcal{A})$ is the spread of non-seed rewards in the archive, and $P(s)$ is a rank-based prior over archive rewards normalized to sum to one. The first term exploits known reward; the second drives exploration toward high-ranked, under-visited nodes and decays as their visit count grows.

Batched expansion with lineage blocking. The top-scoring node is expanded each step. When the search runs with a batch size greater than one, additional parents are added greedily in score order, but no two parents in a single batch may share an ancestor or descendant, so a batch covers distinct branches of the archive. Each selected parent is sent to the LLM mutation provider, which proposes K candidate edits to the parent's kernel source. Every proposed candidate is then compiled, correctness-checked, and timed on the target GPU as a single batch evaluation.

Backpropagation and archive maintenance. After a batch evaluation, the best valid child reward per parent is folded into $m(\cdot)$ for the direct parent only, and visit counts $n(\cdot)$ are incremented along the parent and all of its ancestors. T is incremented once per parent expanded. A parent that produces zero valid children still has its visit counts incremented, so a node that consistently fails has its exploration bonus decay. Children are inserted into the archive subject to a top-k-per-parent cap (by reward), deduplication by program text, and a global capacity bound that preserves seed nodes.

Mutation prompt and reward shaping. The LLM mutation provider receives the parent kernel, a description of the target operation, and (in the goal-conditioned configuration used for the eval set) a target speedup bin. The reward is the geometric-mean speedup over the task's reference inputs, with invalid candidates assigned a sentinel that excludes them from m updates and from archive insertion. Concrete hyperparameters used in this paper (batch size, samples per parent, k per parent, total step

Algorithm 1 Max-reward tree search with a PUCT selection rule, our primary method to search for kernels. The mutation and evaluation providers are the only components that change between runs: the mutation provider is the LLM that proposes kernel edits, and the evaluation provider compiles, correctness-checks, and times each candidate on the target GPU.

```
def max_reward_tree_search(seed_program, mutation_provider, evaluation_provider,
                           total_steps, batch_size, samples_per_parent, k_per_parent,
                           c_puct=1.0, capacity=1000):
    root = Node(seed_program, evaluate(seed_program), is_seed=True)
    archive, seed_ids = [root], {root.id}
    n, m, T = {}, {}, 0

    for step in range(total_steps):
        # SELECTION: score every archive node, pick a batch of parents
        # under lineage blocking (no ancestor/descendant pair in one batch).
        scored = puct_scores(archive, n, m, T, seed_ids, c_puct)
        parents = select_with_lineage_blocking(scored, batch_size)
        if not parents:
            break

        # EXPANSION: LLM proposes samples_per_parent edits per parent;
        # all candidates are compiled, correctness-checked, and timed.
        candidates, parent_of = [], []
        for p in parents:
            for code in mutation_provider.mutate(p, samples_per_parent):
                candidates.append(code)
                parent_of.append(p)
        evals = evaluation_provider.batch_evaluate(candidates)
        children = [Node(c, e) for c, e in zip(candidates, evals)]

        # BACKPROPAGATION: m on direct parent, n on parent + ancestors,
        # T += 1 per expanded parent. Failed parents still decay.
        T = backpropagate(children, parent_of, n, m, T)
        for p in parents_with_no_valid_child(parents, children, parent_of):
            T = record_failed_rollout(p, n, T)

        # ARCHIVE UPDATE: top-k children per parent (by reward),
        # dedup by program text, truncate to capacity preserving seeds.
        update_archive(archive, children, parent_of, seed_ids,
                       k_per_parent=k_per_parent, capacity=capacity)

    return archive
```

budget, c_{puct}, archive capacity) follow the per-experiment configurations in our released code; the search procedure itself is identical across runs.

Surrogate-gated search and invalid candidates. In the surrogate-gated configuration of Section 3.4, each parent yields cG candidates, the surrogate forecasts all of them, the top G by expected speedup are measured, and the remaining candidates enter the archive carrying their forecast as a speculative score. A measured candidate that fails to compile, run, or pass correctness checks has its forecast replaced by the invalid sentinel, so invalid candidates, which make up about 15% of proposals across tasks and up to 45% on TriMul, are handled exactly as they are in the baseline search. We use $c = G = 4$.

Pseudocode. Algorithm 1 summarizes the loop.

H Reinforcement Learning Implementation Details

All reinforcement-learning variants start from GPT-OSS-20B. We sample the model with the same reasoning renderer used for the untrained GPT-OSS-20B baseline, so training and scoring share the same prompt format, tool-call schema, and parsing path.

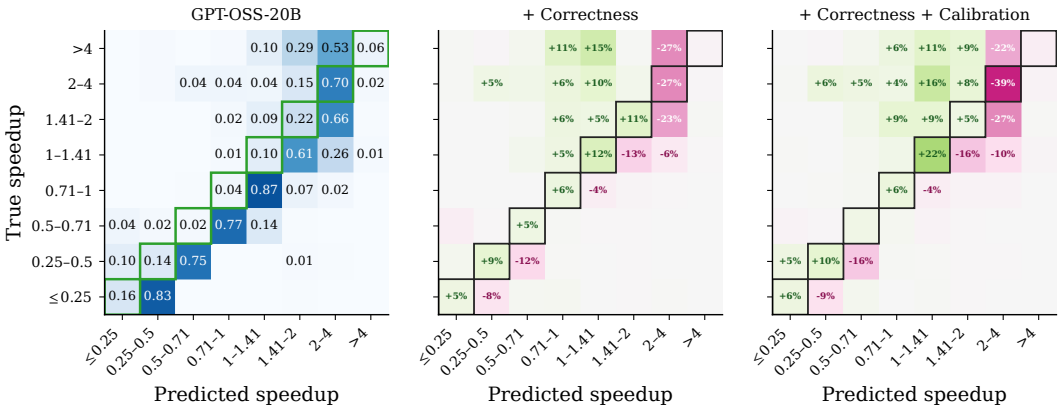

Figure 6: **The base surrogate overestimates slow kernels and underestimates the fastest kernels, while RL changes where those errors go.** Rows are measured speedup bins and columns are predicted bins for GPT-OSS-20B; each cell is the fraction of examples in that measured bin receiving that prediction. The middle and right panels show how training changes those row fractions relative to the base (% change): green cells gain probability mass after training and magenta cells lose it.

The training set is built from GPT-OSS-20B kernel-search traces on a single task (TriMul). Each row contains an anchor kernel, a candidate kernel, the A100 hardware context, and the candidate's measured speedup relative to the anchor. Rows are generated from seed-anchored comparisons, parent-edit comparisons, and within-kernel pair comparisons between successful candidates. Failed candidates are not used for reinforcement learning.

Each rollout is a single-turn interaction. The prompt contains the reference kernel, the candidate kernel, and hardware properties of the target GPU. The model samples a chain of reasoning and then calls a structured tool. In the predict-only variants, this tool returns the predicted speedup bin, a probability distribution over the eight speedup bins, and a short rationale. We also implemented an exploratory native-abstention variant, in which the model chooses between the forecast tool and a defer tool, but we do not use this variant in the main evaluation.

We train LoRA adapters on GPT-OSS-20B with reinforcement learning. Each training batch contains groups of rollouts for the same kernel comparison, and the update uses the relative rewards within those groups. The main text gives the reward definitions; all trained variants use the same training rows and the same training budget unless otherwise noted. The variants discussed here use twenty reinforcement-learning iterations, group size eight, learning rate 4×10^{-5}, rank-32 adapters, temperature 1.0, and a maximum rollout length of 8192 tokens.

After training, each checkpoint is scored on the held-out evaluation using the same forecast schema as the off-the-shelf surrogates. Each held-out row is sampled three times at temperature 1.0. The headline tables compare these scored checkpoints against the same GPT-OSS-20B model before reinforcement learning.

I Additional Analysis of the Trained Surrogates

This appendix collects analysis of the RL-trained GPT-OSS-20B surrogates that supports Section 3.5: where training moves probability mass, what each reward costs in reasoning length and latency, and how the ranking change varies with the measurement budget.

Where training moves probability mass. Figure 6 shows that the untrained GPT-OSS-20B predicts slow / moderate kernels will be faster than they are, while predicting $> 4\times$ kernels to be slower than they are. Kernels $4\times$ faster than the reference are likely hard to spot for the model. RL redistributes probability mass across speedup bins, and Table 4 shows the tradeoff.

Table 4 compares each RL reward to the GPT-OSS-20B base. All three rewards improve calibration, but they differ in search utility: correctness-only training loses 1.29% in speedup recovered, CRPS

Table 4: **RL changes several surrogate properties at once.** All entries are changes from the untrained GPT-OSS-20B base; rows differ only in reward. Length is reasoning chain length. Latency is per-call latency.

Rewards		Change vs. GPT-OSS-20B base				
Correctness	Calibration	Speedup recovered (%) ↑	ECE ↓	Forecast error ↓	Length (tokens) ↓	Latency (s) ↓
✓	–	−1.29	−0.031	+0.118	−812	−3.39
✓	CRPS	−1.05	−0.025	+0.100	+184	+17.80
✓	Brier	+2.06	−0.106	+0.130	−982	−5.02

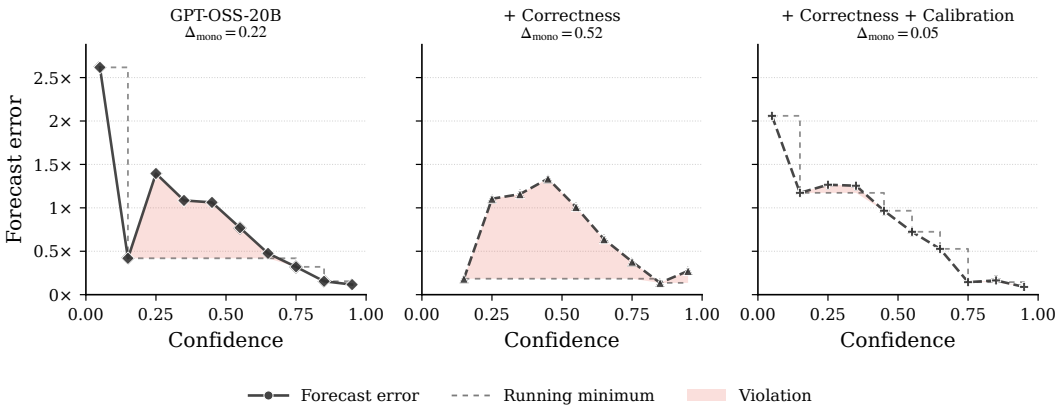

Figure 7: **Calibration rewards make confidence track forecast error, while correctness-only RL weakens this relationship.** Examples are bucketed by stated confidence, $q_i = \max_b p_{i,b}$. The y-axis is mean forecast error within each bucket; lower and more decreasing curves indicate more reliable confidence. Shading marks violations of monotone decrease, summarized by Δ_{mono} with 0 indicating no violation.

is roughly neutral, and Brier gains 2.06%. The correctness reward only checks whether the argmax bin is correct, so it can improve exact-bin decisions while ignoring how useful the rest of the probability vector is. Brier and CRPS instead score the full distribution p_i. That matters for search because candidates are ranked by expected speedup, $\hat{S}_i = \sum_b p_{i,b} m_b$, not just by the argmax bin. A probability vector with informative mass across nearby bins can separate candidates that share the same predicted bin while a binary vector cannot. The tradeoff is that all trained variants increase forecast error. Forecast error uses only the argmax-bin representative $m_{\hat{y}_i}$, so it can worsen when training shifts the most likely bin even if the full distribution becomes better calibrated or more useful for ranking.

Does stated confidence track forecast quality? The surrogate's confidence is useful for selective use only if it correlates with forecast quality, that is, if a candidate the surrogate labels confidently is one it really gets right. Figure 7 tests this by plotting mean forecast error against stated confidence, bucket by bucket. A desirable surrogate would produce a curve that decreases monotonically (i.e., forecast error is monotonically lower for higher-confidence predictions), and Δ_{mono} quantifies this by measuring how far the curve deviates from monotonicity, with 0 meaning monotone-decreasing and 1 meaning monotone-increasing; lower is better. The off-the-shelf base satisfies the property roughly ($\Delta_{\text{mono}} = 0.237$), with high-confidence forecasts off by less than $0.5\times$ on average and low-confidence forecasts off by more than $2\times$. Training on correctness alone destroys it ($\Delta_{\text{mono}} = 0.52$): the model becomes confident on candidates it gets wrong by a wide margin. Adding a Brier shaping term to the same correctness reward improves it even beyond the base model ($\Delta_{\text{mono}} = 0.088$).

Ranking change across measurement budgets. Figure 8 asks how useful the surrogate would be inside a kernel search loop where GPU measurements are the scarce resource. A searcher would rank generated candidates by predicted expected speedup, measure from the top of that list downward, and hope to find a fast kernel before exhausting its budget. We therefore simulate this process at

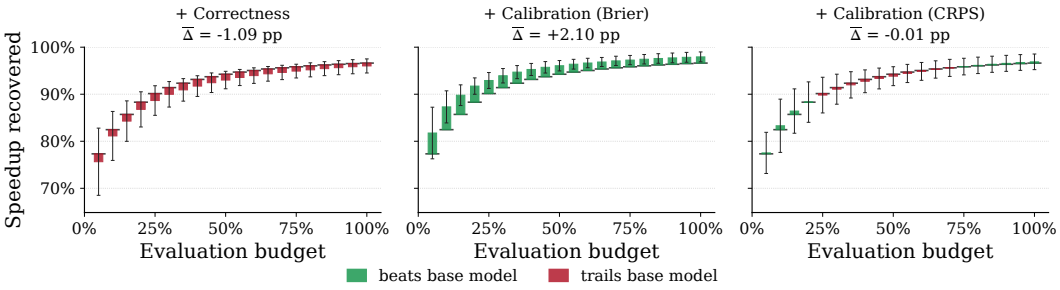

Figure 8: **Brier calibration improves the speedup found under most measurement budgets; correctness-only and CRPS training do not.** Budgets are the fraction of candidates GPU-measured after sorting by predicted expected speedup. Gray dashes show the GPT-OSS-20B base; colored bars show trained-minus-base speedup recovered, with green above and red below the base. Error bars show $\pm 1\sigma$ across repeats, and $\overline{\Delta}$ is the area-averaged change across the plotted budget curve in percentage points.

different measurement budgets and report the fraction of the best available speedup recovered. Table 1 summarizes this metric over the small-budget points $1\%, 5\%, 10\%, 25\%, 50\%$, while Figure 8 shows the trained-minus-base change across the full plotted sweep and reports its area-averaged delta. The base is already strong, recovering 77% of the best available speedup after measuring the top-ranked 5% of candidates and 94% after measuring 50%. Across the full plotted curve, Brier-shaped training averages $+2.10$ pp, CRPS shaping averages -0.01 pp, and correctness-only averages -1.09 pp. The pattern within proper scoring rules is unexpected. CRPS gives partial credit when the prediction is close to the right bin, while Brier penalizes every wrong bin equally, so a priori one would expect CRPS to give the better ranking. The data shows the opposite: bin-order awareness in the loss does not, on its own, lift the candidate ranking the surrogate produces. The CRPS reward barely moved over training (see below), which limits what this comparison can show.

Other backbones. Table 5 applies the correctness + Brier recipe to GPT-OSS-120B and Qwen3-8B. Both start with a larger calibration gap than GPT-OSS-20B, and training narrows it for both.

Table 5: **The calibration improvement from correctness + Brier training appears on two other backbones.** ECE and Δ_{mono} before and after training with the same recipe; lower is better.

Backbone	ECE untrained	ECE trained	Δ_{mono} untrained	Δ_{mono} trained
GPT-OSS-20B	0.441 ± 0.011	$\mathbf{0.335 \pm 0.012}$	0.237	**0.088**
GPT-OSS-120B	0.479 ± 0.005	$\mathbf{0.335 \pm 0.009}$	0.474	**0.264**
Qwen3-8B	0.623 ± 0.005	$\mathbf{0.218 \pm 0.013}$	0.488	**0.172**

Training dynamics. For the correctness-only, correctness + Brier, and correctness + CRPS rewards, mean reward at the first and last of the twenty training iterations was $0.062 \to 0.234$, $0.551 \to 0.828$, and $0.971 \to 0.970$ respectively, with peaks of 0.281 and 0.858 at iteration 10 for the first two and 1.218 at iteration 18 for CRPS. Reward for the first two conditions plateaued around the midpoint of training. Under the correctness-only reward, 89 of 160 training groups (55.6%) had no reward variance and contributed no gradient; under both calibration rewards, none did.

J Additional Reproducibility Details

This appendix collects the concrete implementation settings for the main-paper experiments.

GPU measurement protocol. We use the same measurement protocol as KernelBench (Ouyang et al., 2025). Each candidate is evaluated independently on a Modal A100-80GB worker. For each benchmark case, the evaluator sets the Python, NumPy, PyTorch CPU, and PyTorch CUDA RNG seeds to 42, imports the candidate in an isolated temporary module, generates the case input on the

GPU, and runs one obligatory correctness pass before timing. Candidate timing is measured only after the candidate passes correctness. Timing uses CUDA events with a synchronize before and after every measured call. The timed function receives a fresh clone of the input data on every iteration, so kernels that mutate their inputs in place do not affect later timing iterations. The evaluator first times the candidate, then times the reference on the same generated input. The reported per-case runtime is the mean over an adaptive loop with at least three iterations and at most 100 iterations; the loop stops early when the standard error divided by the mean is below 0.001, when mean runtime times number of runs exceeds 10 seconds, or when benchmark wall clock exceeds 120 seconds. Per-case speedup is reference mean runtime divided by candidate mean runtime, and the scalar reward used by search and eval-set construction is the geometric mean of per-case speedups across benchmark cases.

Correctness checks and benchmark cases. The held-out evaluation set uses six GPU Mode packs, with correctness and benchmark cases taken from the corresponding GPU Mode task definitions. Correctness cases are separate for TriMul, Gated DeltaNet, and FP8 quantization; cross-entropy uses the same shape family for correctness and runtime with different seeds. Shape and dtype are checked before numerical closeness for the multi-output packs. Benchmark cases are the cases used to compute the geometric-mean speedup label.

- **TriMul.** Correctness uses rtol 2×10^{-2} and atol 2×10^{-2} on 18 public task-specification cases covering normal/Cauchy inputs, mask/no-mask, sequence lengths 32–1024, batch sizes 1–2, dim 128–768, and hidden dim 128. Benchmark cases are $(N, B, D, H, \text{mask}, \text{dist})$: (256,2,128,128,no,normal), (768,1,128,128,no,Cauchy), (256,2,384,128,yes,normal), (512,1,128,128,no,normal), (1024,1,128,128,no,Cauchy), (768,1,384,128,yes,normal), and (1024,1,384,128,no,normal).
- **Cross-entropy.** Correctness uses rtol 10^{-2} and atol 10^{-3}. Correctness and benchmark cases both use $B = 4096$ with vocabulary sizes 32000, 50264, and 128256.
- **GDN chunk-fwd-h.** Correctness uses rtol 10^{-3} and atol 10^{-3} on (B, T, H, K, V) cases (1,64,2,64,64), (2,128,4,64,64), and (1,256,4,64,128). Benchmark cases are (1,64,1,64,64), (2,512,3,64,64), and (2,1024,3,64,64).
- **GDN chunk-fwd-o.** Correctness uses rtol 10^{-3} and atol 10^{-3} with the same Gated DeltaNet correctness and benchmark cases as GDN chunk-fwd-h.
- **GDN recompute-w-u.** Correctness uses rtol 10^{-3} and atol 10^{-3} with the same Gated DeltaNet correctness and benchmark cases as GDN chunk-fwd-h.
- **FP8 quantization.** Correctness uses rtol 10^{-3} and atol 10^{-3} on (tokens, D, G) cases (1,256,64), (4,512,128), (16,1024,64), (1,4096,128), and (8,4096,128). Benchmark cases are (256,4096,128), (256,8192,128), and (4096,7168,128).

Eval-set construction and search hyperparameters. The held-out evaluation set contains the six packs listed above. For each pack, the evaluation set is capped at 10 kernels per non-failure speedup bin, so the scored evaluation set has at most 80 rows per pack and 480 rows total. Rows are sorted by bin before capping. The source searches used to harvest held-out candidates were disjoint from the GPT-OSS-20B searches used to build the RL training rows.

For the source PUCT searches, cross-entropy and GDN chunk-fwd-h were generated with Gemini-3 Flash; GDN chunk-fwd-o, GDN recompute-w-u, and FP8 quantization were generated with Gemini-3 Pro; TriMul was harvested from a separate Gemini-3 Pro A100 search. The source PUCT searches used one run, 40 budget steps, batch size 2, four samples per parent, top-2 children per parent, archive capacity 1000, $c_{\text{puct}} = 1.0$, max LLM concurrency 8, temperature 1.0, geomean reward aggregation, and A100-80GB evaluation. The TriMul source search used 10 budget steps, batch size 4, four samples per parent, top-4 children per parent, max LLM concurrency 8, Gemini-3 Pro, A100-80GB evaluation, and geomean aggregation.

Goal-conditioned bin filling used the same setting for every held-out pack: four budget steps, batch size 1, four samples per parent, top-4 children per parent, archive capacity 32, $c_{\text{puct}} = 1.0$, 600-second per-request timeout, max LLM concurrency 8, four retries, temperature 1.0, Triton 3.3.1, A100-SXM4-80GB hardware in the prompt, A100-80GB evaluation, max 10 evaluations in flight, max 10 Modal containers, 1200-second result timeout, and geomean aggregation. Depending on the pack, bin filling used Gemini-3 Flash, Gemini-3 Pro, or a combination of the two.

RL training and scoring settings. The three trained surrogates used in the paper are the final checkpoints from the correctness, correctness+Brier, and correctness+CRPS training runs. All start from `openai/gpt-oss-20b` and train LoRA adapters with GRPO through Tinker. The shared trainer settings are learning rate 4×10^{-5}, group size 8, 8 groups per batch, 20 training iterations, checkpoint save interval 5, maximum rollout length 8192 tokens, LoRA rank 32, rollout temperature 1.0, and the same GPT-OSS-20B prompt/reasoning format used for baseline scoring. Constant-reward groups are removed before update.

The training set is built from six GPT-OSS-20B source searches over the same packs but disjoint from the held-out evaluation searches. Rows are successful measurements only and use three anchor sources: seed-anchored rows against the pack reference, parent-edit rows against the successful parent in the PUCT tree, and within-pack successful-candidate pairs. Within-pack pairs use 500 sampled pairs per pack, RNG seed 0, log-ratio strata $[0, 0.1)$, $[0.1, 0.5)$, $[0.5, 1.0)$, $[1.0, 2.0)$, and $[2.0, \infty)$, and no additional tie filtering.

The correctness-only checkpoint uses reward $r = \mathbf{1}[\hat{y} = y]$. The Brier checkpoint uses $r = \mathbf{1}[\hat{y} = y] + 1 - \mathrm{Brier}(p, y)$. The CRPS checkpoint uses $r = \mathbf{1}[\hat{y} = y] + 1 - \mathrm{CRPS}(p, y)$.

Surrogate scoring settings. The off-the-shelf Gemini row is scored with LiteLLM model slug `gemini/gemini-3-flash-preview`; GPT-OSS-120B is scored with `together_ai/openai/gpt-oss-120b`; the untrained GPT-OSS-20B row is scored through Tinker with base model `openai/gpt-oss-20b`. Gemini and GPT-OSS-120B use temperature 1.0, max 32000 output tokens, 900-second request timeout, four retries, three repeats per row, and max scoring concurrency 8. The GPT-OSS-20B baseline and all trained checkpoints use temperature 1.0, max 16384 output tokens, three repeats per row, max concurrency 8, and the same GPT-OSS-20B prompt/reasoning format. Each trained scoring run uses the final checkpoint from the corresponding training run and otherwise uses the same scoring pipeline as the untrained GPT-OSS-20B baseline.

K PROMPTS

Table 6: **Speedup bins used as the surrogate prediction target.** The speedup ratio $S_i = T_{\mathrm{ref}}/T_i$ is the candidate runtime improvement relative to the reference implementation, so $S_i > 1$ means the candidate is faster and $S_i < 1$ means it is slower. The representative value m_b is used when converting a bin distribution or predicted bin into a numeric speedup; interior-bin representatives are geometric centers, and open-bin representatives are empirical means over examples in that bin.

Bin	Name	Speedup range	m_b	Interpretation
1	Severe slowdown	$S_i \leq 0.25$	empirical mean	Candidate is at least 4× slower
2	Significant slowdown	$0.25 < S_i \leq 0.5$	0.35	Candidate is 2–4× slower
3	Moderate slowdown	$0.5 < S_i \leq 0.71$	0.59	Candidate is about 1.4–2× slower
4	Minor slowdown	$0.71 < S_i \leq 1.0$	0.84	Candidate is near baseline but slower
5	Minor speedup	$1.0 < S_i \leq 1.41$	1.19	Candidate is slightly faster
6	Significant speedup	$1.41 < S_i \leq 2.0$	1.68	Candidate is about 1.4–2× faster
7	High speedup	$2.0 < S_i \leq 4.0$	2.83	Candidate is 2–4× faster
8	Extreme speedup	$S_i > 4.0$	empirical mean	Candidate is more than 4× faster

This appendix contains the exact prompt templates used in the paper. The surrogate prompts (boxes K.1 to K.4) are rendered into the LLM forecaster that predicts a candidate kernel's speedup bin. The mutation prompts (boxes K.5, K.6 and K.8) are rendered into the LLM that proposes the next kernel edit during search. Templates are shown verbatim with their Jinja2 placeholders. Pack-specific descriptions used by the GPU Mode mutation prompt are not reproduced in full. We show the rendered structure with a single example (TriMul) to illustrate the format.

K.1 SURROGATE FORECASTER PROMPTS

Box K.1| Surrogate system prompt (predict-only)

```
You are a GPU execution simulator. Your task is to predict the
relative speedup of a candidate CUDA kernel compared to a reference
kernel implementation. You will reason step-by-step about how each
kernel executes on GPU hardware, then predict which speedup bin the
candidate falls into.

## Speedup Definition

Speedup S = (reference runtime) / (candidate runtime).
- S > 1 means the candidate is FASTER than the reference.
- S < 1 means the candidate is SLOWER than the reference.
- S = 1 means identical performance.

## Speedup Bins

You must predict one of these bins:

| Bin | Name | Speedup Range | Description |
|-----|------|---------------|-------------|
| 1 | SEVERE_SLOWDOWN | S <= 0.25 | Candidate is 4x+ slower |
| 2 | SIGNIFICANT_SLOWDOWN | 0.25 < S <= 0.5 | Candidate is 2x-4x
↪  slower |
| 3 | MODERATE_SLOWDOWN | 0.5 < S <= 0.71 | Candidate is ~1.4x-2x
↪  slower |
| 4 | MINOR_SLOWDOWN | 0.71 < S <= 1.0 | Candidate is slightly
↪  slower or equal |
| 5 | MINOR_SPEEDUP | 1.0 < S <= 1.41 | Candidate is slightly
↪  faster |
| 6 | SIGNIFICANT_SPEEDUP | 1.41 < S <= 2.0 | Candidate is ~1.4x-2x
↪  faster |
| 7 | HIGH_SPEEDUP | 2.0 < S <= 4.0 | Candidate is 2x-4x faster |
| 8 | EXTREME_SPEEDUP | S > 4.0 | Candidate is 4x+ faster |

## Probability Distribution

Your prediction is a probability distribution over bins 1 through
↪  8.
For each bin, assign a probability in [0, 1] reflecting how likely
the candidate's speedup is to fall in that bin. The eight
probabilities must sum to 1.

Your distribution should reflect genuine uncertainty. For most
predictions, the probability mass is concentrated on 1-2 adjacent
bins (your best guess and a neighbor), with smaller mass on the
rest. Avoid placing all mass on a single bin unless your analysis
is decisive.

## How to Analyze

Simulate the execution of both kernels on the GPU. Consider these
factors systematically:

1. **Algorithmic complexity**: How many FLOPs does each kernel
   perform? Does the candidate reduce total work (e.g., fused
   operations, fewer passes over data)?

2. **Memory access patterns**: Are global memory accesses
   coalesced? Does the candidate use shared memory or registers
   to reduce global memory traffic? How many bytes are read/written
   per thread?
```

```
3. **Arithmetic intensity**: What is the ratio of compute to memory
   operations? Is the kernel compute-bound or memory-bandwidth-
   bound? This determines which optimizations matter.

4. **Thread divergence**: Do conditionals cause warp divergence?
   Does the candidate reduce branch divergence compared to the
   reference?

5. **Occupancy and resource pressure**: How many registers per
   thread? How much shared memory per block? These limit the
   number of concurrent warps and can bottleneck throughput.

6. **Parallelism and grid dimensions**: Does the candidate expose
   more parallelism? Are there enough threads to saturate the GPU?
   Is the work evenly distributed across blocks?

7. **Synchronization overhead**: Does the kernel use
   __syncthreads(), atomics, or other synchronization primitives?
   These can serialize execution.

8. **Kernel launch overhead**: For very fast kernels, launch
   overhead can dominate. Multiple kernel launches vs. a single
   fused kernel matters.

9. **Library calls vs. custom code**: Does the reference use highly
   optimized vendor libraries (cuBLAS, cuDNN)? Custom kernels
   rarely beat these for standard operations (GEMM, convolution).

10. **Data types and special hardware**: Does the candidate use
    lower precision (FP16, INT8) or tensor cores? These can
    provide large speedups for eligible operations.

Think carefully about the RELATIVE performance. A candidate might
be well-written but still slower than a PyTorch reference that
dispatches to cuBLAS. Conversely, a simple-looking candidate might
achieve large speedups by fusing operations that the reference
executes as separate kernel launches.

## Output Format

Reason step by step about both kernels, then submit your prediction
by calling the tool `{{ tool_name }}` exactly **once**. Do not
respond in plain text and do not call any other tool.

The tool's arguments are:
- `predicted_bin`: integer in [1, 8], the most likely bin.
- `p_severe_slowdown` ... `p_extreme_speedup`: eight floats in
  [0, 1] that sum to 1, one per bin in order.
- `reasoning`: 1-3 sentence rationale for the prediction.
```

Box K.2| Surrogate user prompt (predict-only)

```
## Task: {{ task.op_name }} (Level {{ task.level_id }})

Predict the speedup of the candidate kernel relative to the
reference kernel.

## Reference Kernel: {{ reference.kernel_name }}

```python
{{ reference.code }}
```
```

```
## Candidate Kernel: {{ candidate.kernel_name }}

```cuda
{{ candidate.code }}
```
{% if hardware %}

Hardware Context

Property	Value
Device	{{ hardware.device_name }}
Compute Capability	{{ hardware.compute_capability[0] }}.{{
↪ hardware.compute_capability[1] }}
Global Memory	{{ "%.1f"|format(hardware.total_global_memory_gb)
↪ }} GB
Multiprocessors	{{ hardware.multiprocessor_count }}
Max Threads/SM	{{ hardware.max_threads_per_multiprocessor }}
Clock Rate	{{ "%.2f"|format(hardware.clock_rate_ghz) }} GHz
Memory Clock	{{ "%.2f"|format(hardware.memory_clock_rate_ghz)
↪ }} GHz
Memory Bus Width	{{ hardware.memory_bus_width_bits }} bits
{% endif %}

Analyze both kernels and predict the speedup bin. Submit your
prediction by calling the `{{ tool_name }}` tool exactly once.
```

Box K.3| Surrogate system prompt (predict-or-defer variant)

```
You are a GPU execution simulator. Your task is to either predict
the relative speedup of a candidate CUDA kernel compared to a
reference kernel implementation, or, when you are too uncertain to
predict reliably, defer the candidate to a real GPU evaluator. You
will reason step-by-step about how each kernel executes on GPU
hardware, then either submit a probability distribution over
speedup bins or call the deferral tool.

## Speedup Definition

Speedup S = (reference runtime) / (candidate runtime).
- S > 1 means the candidate is FASTER than the reference.
- S < 1 means the candidate is SLOWER than the reference.
- S = 1 means identical performance.

## Speedup Bins

When you predict, you must select one of these bins as the most
likely outcome:

| Bin | Name | Speedup Range | Description |
|-----|------|---------------|-------------|
| 1 | SEVERE_SLOWDOWN | S <= 0.25 | Candidate is 4x+ slower |
| 2 | SIGNIFICANT_SLOWDOWN | 0.25 < S <= 0.5 | Candidate is 2x-4x
↪  slower |
| 3 | MODERATE_SLOWDOWN | 0.5 < S <= 0.71 | Candidate is ~1.4x-2x
↪  slower |
| 4 | MINOR_SLOWDOWN | 0.71 < S <= 1.0 | Candidate is slightly
↪  slower or equal |
| 5 | MINOR_SPEEDUP | 1.0 < S <= 1.41 | Candidate is slightly
↪  faster |
| 6 | SIGNIFICANT_SPEEDUP | 1.41 < S <= 2.0 | Candidate is ~1.4x-2x
↪  faster |
```

```
| 7 | HIGH_SPEEDUP | 2.0 < S <= 4.0 | Candidate is 2x-4x faster |
| 8 | EXTREME_SPEEDUP | S > 4.0 | Candidate is 4x+ faster |

## Probability Distribution

When you predict, you produce a probability distribution over bins
1 through 8. For each bin, assign a probability in [0, 1]
reflecting how likely the candidate's speedup is to fall in that
bin. The eight probabilities must sum to 1.

Your distribution should reflect genuine uncertainty. For most
predictions, the probability mass is concentrated on 1-2 adjacent
bins (your best guess and a neighbor), with smaller mass on the
rest. Avoid placing all mass on a single bin unless your analysis
is decisive.

## How to Analyze

Simulate the execution of both kernels on the GPU. Consider these
factors systematically:

1. **Algorithmic complexity**: How many FLOPs does each kernel
   perform? Does the candidate reduce total work (e.g., fused
   operations, fewer passes over data)?

2. **Memory access patterns**: Are global memory accesses
   coalesced? Does the candidate use shared memory or registers
   to reduce global memory traffic? How many bytes are read/written
   per thread?

3. **Arithmetic intensity**: What is the ratio of compute to memory
   operations? Is the kernel compute-bound or memory-bandwidth-
   bound? This determines which optimizations matter.

4. **Thread divergence**: Do conditionals cause warp divergence?
   Does the candidate reduce branch divergence compared to the
   reference?

5. **Occupancy and resource pressure**: How many registers per
   thread? How much shared memory per block? These limit the
   number of concurrent warps and can bottleneck throughput.

6. **Parallelism and grid dimensions**: Does the candidate expose
   more parallelism? Are there enough threads to saturate the GPU?
   Is the work evenly distributed across blocks?

7. **Synchronization overhead**: Does the kernel use
   __syncthreads(), atomics, or other synchronization primitives?
   These can serialize execution.

8. **Kernel launch overhead**: For very fast kernels, launch
   overhead can dominate. Multiple kernel launches vs. a single
   fused kernel matters.

9. **Library calls vs. custom code**: Does the reference use highly
   optimized vendor libraries (cuBLAS, cuDNN)? Custom kernels
   rarely beat these for standard operations (GEMM, convolution).

10. **Data types and special hardware**: Does the candidate use
    lower precision (FP16, INT8) or tensor cores? These can
    provide large speedups for eligible operations.

Think carefully about the RELATIVE performance. A candidate might
be well-written but still slower than a PyTorch reference that
```

```
dispatches to cuBLAS. Conversely, a simple-looking candidate might
achieve large speedups by fusing operations that the reference
executes as separate kernel launches.

## Predict or Defer

You have two tools available and must call exactly one of them:

- **`{{ predict_tool_name }}`** -- submit your speedup-bin
  distribution as a prediction.
- **`{{ defer_tool_name }}`** -- defer this candidate to a real GPU
  evaluator instead of predicting.

Use **`{{ predict_tool_name }}`** when your analysis converges on a
small set of plausible bins. It is acceptable for the distribution
to span two or three adjacent bins; that is what the probability
simplex is for.

Use **`{{ defer_tool_name }}`** when, after analysis, you would not
bet on your top bin being correct. Concretely, defer when:

- your analysis leaves two or more bins close enough that you
  cannot order them confidently, and they span a range wide enough
  that getting the top bin wrong is meaningful (e.g., a 2x or
  larger speedup range);
- the candidate uses an optimization or hardware feature whose
  effect on this specific reference cannot be reasoned about from
  the code alone (e.g., the runtime depends on cache behavior or on
  a tunable parameter whose value matters more than the algorithm);
- the candidate has multiple plausible failure modes (e.g., it
  might be incorrect, or it might compile but produce numerically
  unstable results) and these modes would land it in different
  bins;
- the reference is a vendor-library dispatch and you cannot tell
  whether the candidate is competitive with that library on this
  hardware.

Deferral is not free. In deployment, a deferred candidate is sent
to a real GPU for evaluation, which is expensive and slow. Reserve
deferral for the candidates where you genuinely would not bet on
your prediction. Predicting confidently and being right is the best
outcome; predicting confidently and being wrong is the worst;
deferring is a middle ground when you cannot tell.

## Output Format

Reason step by step about both kernels. Then call exactly one tool:
either **`{{ predict_tool_name }}`** with your bin distribution, or
**`{{ defer_tool_name }}`** with a brief reason. Do not respond in
plain text, and do not call both tools.

Arguments of `{{ predict_tool_name }}`:
- `predicted_bin`: integer in [1, 8], the most likely bin.
- `p_severe_slowdown` ... `p_extreme_speedup`: eight floats in
  [0, 1] that sum to 1, one per bin in order.
- `reasoning`: 1-3 sentence rationale for the prediction.

Arguments of `{{ defer_tool_name }}`:
- `reason`: 1-2 sentences naming the specific source of uncertainty
  that motivates deferral.
```

Box K.4| Surrogate user prompt (predict-or-defer variant)

```
## Task: {{ task.op_name }} (Level {{ task.level_id }})

Either predict the speedup of the candidate kernel relative to the
reference kernel, or defer this candidate to a real GPU evaluator
if you are too uncertain to predict reliably.

## Reference Kernel: {{ reference.kernel_name }}

```python
{{ reference.code }}
```

Candidate Kernel: {{ candidate.kernel_name }}

```cuda
{{ candidate.code }}
```
{% if hardware %}

Hardware Context

Property	Value
Device	{{ hardware.device_name }}
Compute Capability	{{ hardware.compute_capability[0] }}.{{
↪ hardware.compute_capability[1] }}
Global Memory	{{ "%.1f"|format(hardware.total_global_memory_gb)
↪ }} GB
Multiprocessors	{{ hardware.multiprocessor_count }}
Max Threads/SM	{{ hardware.max_threads_per_multiprocessor }}
Clock Rate	{{ "%.2f"|format(hardware.clock_rate_ghz) }} GHz
Memory Clock	{{ "%.2f"|format(hardware.memory_clock_rate_ghz)
↪ }} GHz
Memory Bus Width	{{ hardware.memory_bus_width_bits }} bits
{% endif %}

Analyze both kernels. Then call exactly one of the two tools:
`{{ predict_tool_name }}` to submit a prediction, or
`{{ defer_tool_name }}` if you are too uncertain to predict.
```

K.2 MUTATION PROMPTS

The GPU Mode mutation prompt is assembled from a per-kernel description body, a fixed rules block, and a feedback section that depends on the previous candidate's evaluation outcome. Box K.5 shows the rules block appended after every per-kernel description, and box K.6 shows the feedback section appended once a previous candidate has been evaluated. The kernel description body is pack-specific. Box K.7 shows the body for the TriMul kernel as a representative example. Box K.8 shows the KernelBench mutation prompt, which uses a single Jinja2 template covering the seed and feedback cases.

Box K.5| GPU Mode mutation prompt: rules block (appended to every per-kernel description)

```
Rules:
- The tensors arguments passed in will be already on your cuda
↪  device.
- Define all of your code in one final ```python ``` block.
- We will test the correctness of your kernel on multiple input
↪  shapes, make sure to support different potential test cases.
- You are allowed to use mixed precision computations, but make
↪  sure your final output is in float32.
```

```
- You must use triton {triton_version} and these kernels will be
↪  run on an Nvidia {gpu_name}.
- You do not have to implement everything in triton, you may choose
↪  to have some of the operations done in pytorch. However, you
↪  must implement at least part of the operations in a kernel.
- Include a short docstring at the top summarizing your algorithm.
```

Box K.6| GPU Mode mutation prompt: feedback section (appended after a previous candidate has been evaluated)

```
<base prompt>

Here is your latest implementation:
```python
<previous_kernel_code>
```

Your custom_kernel was evaluated on GPU.

Here is the evaluation result:
<one of the four arms below>

CompileFailed arm
Your kernel failed to compile.

Compilation error:
<compilation_error, head-truncated to 2000 chars>

Please fix the errors and try again.

RuntimeError arm
Your kernel raised an exception at runtime.

Error type: <runtime_error_name>

Error message:
<runtime_error, head-truncated to 1000 chars>

Traceback:
<traceback, tail-truncated to 3000 chars>

Please fix the errors and try again.

Incorrect arm
Your kernel produced incorrect output compared to the reference.

Correctness issue:
<error_message, head-truncated to 2000 chars>

Please fix the correctness issues and try again.

Success arm
You are iteratively optimizing runtime (microseconds).

Your kernel is correct. Aggregated speedup: <speedup>x (aggregation
↪ method: <method>).

Per-case breakdown (slowest first):
  <pack-specific case formatting, one line per test case>
  ...

Please rewrite the entire kernel to be as fast as possible. Focus
↪ on the slowest configurations listed above.
```

Box K.7| GPU Mode mutation prompt: kernel description body (TriMul example)

```
You are an expert Triton engineer tasked with translating PyTorch
↪  code into highly optimized Triton kernel code.

You will be implementing a Triangle Multiplicative Update (TriMul)
↪  module that is a core operation
for AlphaFold3, Chai, Protenix, and other protein structure
↪  prediction models in BioML.

The TriMul operator operates over a 4D tensor of shape [B, N, N,
↪  C].

Your task:
- Implement the "outgoing" version of the TriMul operator from the
↪  AlphaFold3 paper.
- You will not have to compute or store gradients for this version.
↪  You will only need to implement the forward pass.

Your function should be defined as 'custom_kernel' with the
↪  following signature:
Input:
- `data`: Tuple of (input: torch.Tensor, weights: Dict[str,
↪  torch.Tensor], config: Dict)
    - input: Input tensor of shape [bs, seq_len, seq_len, dim]
    - mask: Mask tensor of shape [bs, seq_len, seq_len]
    - weights: Dictionary containing model weights
    - config: Dictionary containing model configuration parameters

Output:
- output: Processed tensor [bs, seq_len, seq_len, dim]

**Problem Constraints:**
- B in {1,2}, N in {128,256,512,1024}, c in {128}, c_z in
↪  {128,384,768}
- The input distribution will be sampled from a standard Normal
↪  distribution, or a heavy-tailed Cauchy distribution (gamma =
↪  2).
- There will either be no mask, or a randomly sampled mask over the
↪  inputs.

**Remarks.** So why is this problem so annoying? Because you have
↪  to choose whether to load / deal with either the channel
↪  dimensions c,c_z that the LayerNorms require (otherwise you
↪  have to do a synchronize to compute the statistics like mean /
↪  variance) or the sequence dimension N.
The sequence dimension is particularly annoying because it's quite
↪  large, but also because we compute pair-wise operations at the
↪  last operation that sum over another sequence dimension (this
↪  is N^3!).
However, I really like this kernel because it only consists of
↪  "simple" operations, and is really easy to understand. It is a
↪  true test of "fusions" that torch.compile() doesn't do that
↪  well.

Here is a pytorch implementation of the TriMul module. You will
↪  want to implement a kernel for the operations in the forward
↪  call:

```python
import torch
from torch import nn, einsum
import math
```
```

````
# Reference code in PyTorch
class TriMul(nn.Module):
    def __init__(self, dim: int, hidden_dim: int):
        super().__init__()
        self.norm = nn.LayerNorm(dim)
        self.left_proj = nn.Linear(dim, hidden_dim, bias=False)
        self.right_proj = nn.Linear(dim, hidden_dim, bias=False)
        self.left_gate = nn.Linear(dim, hidden_dim, bias=False)
        self.right_gate = nn.Linear(dim, hidden_dim, bias=False)
        self.out_gate = nn.Linear(dim, hidden_dim, bias=False)
        self.to_out_norm = nn.LayerNorm(hidden_dim)
        self.to_out = nn.Linear(hidden_dim, dim, bias=False)

    def forward(self, x: torch.Tensor, mask: torch.Tensor) ->
    ↪  torch.Tensor:
        batch_size, seq_len, _, dim = x.shape
        x = self.norm(x)
        left = self.left_proj(x)
        right = self.right_proj(x)
        mask = mask.unsqueeze(-1)
        left = left * mask
        right = right * mask
        left_gate = self.left_gate(x).sigmoid()
        right_gate = self.right_gate(x).sigmoid()
        out_gate = self.out_gate(x).sigmoid()
        left = left * left_gate
        right = right * right_gate
        out = einsum('... i k d, ... j k d -> ... i j d', left,
        ↪  right)
        out = self.to_out_norm(out)
        out = out * out_gate
        return self.to_out(out)
```

Here is some example skeleton code of the entrypoint function you
↪  will create:
```python
def custom_kernel(data):
    input_tensor, mask, weights, config = data
    dim, hidden_dim = config["dim"], config["hidden_dim"]
    norm_weight = weights["norm.weight"]
    norm_bias = weights["norm.bias"]
    left_proj_weight = weights["left_proj.weight"]
    right_proj_weight = weights["right_proj.weight"]
    left_gate_weight = weights["left_gate.weight"]
    right_gate_weight = weights["right_gate.weight"]
    out_gate_weight = weights["out_gate.weight"]
    to_out_norm_weight = weights["to_out_norm.weight"]
    to_out_norm_bias = weights["to_out_norm.bias"]
    to_out_weight = weights["to_out.weight"]
    # Perform TriMul
    return out
```

A few general triton tips:
- tl.arange only takes in constexpr arguments (static or
↪  tl.constexpr)
- You cannot use continue in your kernel code
- tl.dot can only take in two input tensors
- There is no tl.mean

Here are the different configs that your kernel will be tested on
↪  ("nomask" sets whether there will be no mask, or a randomly
↪  sampled mask over the inputs):
````
````

```
Test Cases for correctness and runtime (optimize runtime for
↪  these):
  - {"seqlen": 256, "bs": 2, "dim": 128, "hidden_dim": 128,
  ↪  "nomask": True, "distribution": "normal"}
  - {"seqlen": 768, "bs": 1, "dim": 128, "hidden_dim": 128,
  ↪  "nomask": True, "distribution": "cauchy"}
  - {"seqlen": 256, "bs": 2, "dim": 384, "hidden_dim": 128,
  ↪  "nomask": False, "distribution": "normal"}
  - {"seqlen": 512, "bs": 1, "dim": 128, "hidden_dim": 128,
  ↪  "nomask": True, "distribution": "normal"}
  - {"seqlen": 1024, "bs": 1, "dim": 128, "hidden_dim": 128,
  ↪  "nomask": True, "distribution": "cauchy"}
  - {"seqlen": 768, "bs": 1, "dim": 384, "hidden_dim": 128,
  ↪  "nomask": False, "distribution": "normal"}
  - {"seqlen": 1024, "bs": 1, "dim": 384, "hidden_dim": 128,
  ↪  "nomask": True, "distribution": "normal"}
```

Box K.8| KernelBench mutation prompt

```
You are optimizing a PyTorch module by writing custom CUDA kernels.

# Task

Below is a PyTorch module called `Model`. Your job is to produce a
↪  new module called `ModelNew` that computes the same outputs but
↪  runs faster on GPU. The mechanism is **writing custom CUDA
↪  kernels** and compiling them inline with
↪  `torch.utils.cpp_extension.load_inline`, then using them inside
↪  `ModelNew.forward`. You decide which operations to target: fuse
↪  and replace where it pays back, and leave the rest as ordinary
↪  PyTorch (e.g. keep a large `torch.mm` since cuBLAS is hard to
↪  beat). **`ModelNew` must call at least one custom CUDA kernel
↪  that you write yourself** -- pure-PyTorch rewrites do not
↪  count, even if they happen to be faster.

Reference module:

```python
{{ reference_module }}
```
{% if feedback is not none %}

Previous attempt

Here is your previous `ModelNew`:

```python
{{ previous_kernel_code }}
```

It was evaluated on GPU. Result:
{% if feedback.kind == "compile_failed" %}

Compilation failed.
{% if feedback.compilation_error_name %}

Error type: `{{ feedback.compilation_error_name }}`
{% endif %}

Compiler output:
```

```
```
{{ compilation_error_truncated }}
```

Fix the compilation error. Keep the rest of the design unless the
↪  fix requires changing it.
{% elif feedback.kind == "runtime_error" %}

**Runtime error during execution.**
{% if feedback.runtime_error_name %}

Error type: `{{ feedback.runtime_error_name }}`
{% endif %}

Message:

```
{{ runtime_error_truncated }}
```

Traceback:

```
{{ runtime_error_traceback_truncated }}
```

Fix the runtime error. Common culprits: out-of-bounds indexing,
↪ dtype mismatch, missing `.contiguous()`, a kernel launch grid
↪ that does not cover all elements, or a CUDA-side assertion such
↪ as a misaligned address.
{% elif feedback.kind == "incorrect" %}

Output does not match the reference.

Issue: {{ correctness_issue_truncated }}
{% if feedback.max_difference %}

Max absolute differences observed across test cases: {{
↪ feedback.max_difference }}
{% endif %}
{% if feedback.avg_difference %}
Average absolute differences observed across test cases: {{
↪ feedback.avg_difference }}
{% endif %}

Make the output match the reference within standard tolerance
↪ (atol=1e-2, rtol=1e-2). Do not change the architecture beyond
↪ what is required to fix correctness.
{% elif feedback.kind == "success" %}

Correct. Speedup: {{ "%.3f"|format(feedback.speedup) }}x
↪ (kernel = {{ "%.1f"|format(feedback.runtime_us) }} us,
↪ reference = {{ "%.1f"|format(feedback.ref_runtime_us) }} us).

Now make it faster. Rewrite the entire `ModelNew` from scratch with
↪ a more aggressive optimization strategy. Do not tune the
↪ existing kernel incrementally -- propose a different approach
↪ (e.g. fuse more operations across the forward pass, change the
↪ tiling, switch to a more coalesced memory access pattern, or
↪ use cuBLAS where it beats handwritten CUDA).
{% endif %}
{% endif %}

Engineering guidance
```

```
Treat correctness as a hard prerequisite and speed as the
↪  objective.

- **Pick replacements that pay back.** Fusing several small
↪  element-wise ops, replacing a launch-bound sequence, or making
↪  memory accesses more coalesced tends to win. Reimplementing a
↪  single large GEMM in handwritten CUDA rarely beats cuBLAS.
- **Precision.** Inputs are `float32` on a CUDA device. Your final
↪  output must be `float32`. Mixed-precision inside is fine if the
↪  result still matches the reference.
- **Shapes.** Test shapes are determined by `get_inputs()` and
↪  `get_init_inputs()` in the reference module above. Size your
↪  kernels around what those imply. Do not hardcode shapes that
↪  those functions do not fix.
- **Compile flags.** You may pass `extra_cuda_cflags=["-O3"]` (or
↪  similar) to `load_inline`. Do not assume non-default headers.
- **Isolation.** Your code is `exec`'d in an isolated namespace. No
↪  filesystem access, no network, no `__file__`.

# How to ship code

Your final code is loaded via
↪  `torch.utils.cpp_extension.load_inline` and instantiated as
↪  `ModelNew`. For this to work:

- Define a class named exactly **`ModelNew`** that subclasses
↪  `torch.nn.Module`. It is the entry point.
- Compile your CUDA source(s) with
↪  `torch.utils.cpp_extension.load_inline(...)` at module scope
↪  (top-level), not inside `__init__`.
- `ModelNew.__init__` must accept the same arguments as
↪  `Model.__init__` (whatever `get_init_inputs()` returns).
- `ModelNew.forward` must accept the same arguments and return the
↪  same shape and dtype as `Model.forward`.

A minimal example of the skeleton -- pattern only, not
↪  problem-relevant:

```python
{{ skeleton_example }}
```

Output format

You may think through the design above as much as you like. Put
↪ your **final, complete `ModelNew` module in a single
↪ ```python``` code block at the end of your response**. The
↪ grader extracts the last python code block; everything above it
↪ is ignored.
```

L LICENSES FOR EXTERNAL ASSETS

This appendix lists the third-party assets used in the paper, their sources, and their licenses. Each asset is also cited at first use in the main text.

GPU Mode kernel optimization competition. Reference kernels and task specifications are drawn from the GPU Mode leaderboard at https://www.gpumode.com/. The supporting kernel-bot infrastructure (https://github.com/gpu-mode/kernelbot) is released under the MIT License. Individual reference implementations are contributed by the community and used here under their respective open-source licenses as indicated on the leaderboard.

KernelBench. KernelBench (Ouyang et al., 2025) is released by the Scaling Intelligence Lab at Stanford under the MIT License (https://github.com/ScalingIntelligence/KernelBench). We use the published Level 3 task set without modification to the reference implementations.

Sakana AI CUDA Engineer Archive. The AI CUDA Engineer Archive is released by Sakana AI on Hugging Face (https://huggingface.co/datasets/SakanaAI/AI-CUDA-Engineer-Archive) under the CC-BY-4.0 license. We refer to it as an external comparison point for kernel-archive scale.

Models. GPT-OSS-20B is released under the Apache-2.0 license. Gemini 3 Flash and Gemini 3 Pro are accessed through the Google AI API under its standard terms of service. DeepSeek models are used under the DeepSeek License.