KERNELWEAVER: A SHAPE-AWARE KERNEL AGENT FOR GPU KERNEL OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM)-based kernel agents automate GPU kernel development, but the kernels they generate are often optimized for only a few representative input shapes and struggle to maintain performance across the broader range encountered at deployment. This mismatch arises because shape changes alter the dominant hardware bottleneck, requiring different execution configurations. Although generating a dedicated kernel for every shape can address this mismatch, it increases compilation and code-size costs. We present KERNELWEAVER, a shape-aware kernel agent that optimizes cross-shape performance by composing a small set of complementary kernels under deployment budgets. To construct this portfolio, KERNELWEAVER first partitions the shape space using hardware resource constraints to guide candidate generation, then selects kernels according to their marginal performance gain per unit cost. The resulting portfolio is deployed through a shape-aware router, whose execution feedback identifies gaps in offline coverage and guides targeted refinement. On nine common GPU kernel workloads, KERNELWEAVER achieves geometric-mean speedups over PyTorch eager of $1.59\times$ on the NVIDIA RTX 4090 and $1.44\times$ on the NVIDIA A800, compared with $1.27\times$ and $1.12\times$ for the strongest evaluated baseline on the respective platforms. In our deployment cost evaluation, KERNELWEAVER reduces deployment compilation time by 91.71% compared with `torch.compile`.

1 INTRODUCTION

With the growth of deep learning and large language models, GPU kernels increasingly determine the training and inference efficiency of an entire system (Vaswani et al., 2017; Brown et al., 2020). To reduce the dependence of kernel development on expert experience, the kernel agent turns kernel generation, compilation, correctness verification, and iterative refinement into an automatic workflow. CUDA Agent, KDA, and AutoKernel (Dai et al., 2026; Jaber & Jaber, 2026) all adopt this paradigm, and this paper takes it as the starting point. The workflow produces one kernel for an operator and judges that kernel by its speedup on a few representative shapes (Ouyang et al., 2025).

That objective does not match the range of shapes an operator must actually cover. Matrix multiplication in batched inference is wide and regular, whereas the same operator in mixture-of-experts routing is narrow enough to underfill a thread block (Zhou et al., 2022). Attention over short sequences and over 64K contexts differs in reduction depth by three orders of magnitude (Kitaev et al., 2020; Choromanski et al., 2021). Shape changes the bottleneck: narrow matrices are limited by parallelism and memory-access efficiency, large regular matrices by tensor core throughput and on-chip storage, and reduction operators move between cache-resident and streaming access. The thread-block configuration, tile shape, memory layout, and reduction strategy are fixed at compile time, so the single kernel produced by a kernel agent cannot meet these requirements at once. On nine operators and two platforms, an RTX 4090 and an A800, we compare that kernel with PyTorch eager (Paszke et al., 2019) across the shape range. In some scenarios, its maximum performance drop reaches 71.40%. Speedup on the representative shapes averages this drop away.

Generating a dedicated kernel for every shape recovers much of the gap, but cost grows linearly with the number of shapes. Raising the kernel count from one to thirty improves the geometric-mean speedup of Softmax, MatMul, LayerNorm, and FlashAttention by 14.13% to 25.55%, while compilation time and code size grow to several times, and in some cases tens of times, those of the

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.

single kernel. Moving from thirty to forty kernels brings only a marginal gain. The useful operating point is therefore neither the single kernel a kernel agent produces nor one kernel per shape, but a small set of complementary kernels that covers the shape space under a budget (Cai et al., 2020; 2019). Buying more kernels offline cannot recover the remaining shapes at a reasonable cost, so one generation round also leaves residuals.

We accordingly recast the kernel agent objective as an end-to-end problem. Given a GPU specification, an operator, a shape range, and a system budget, the output is a deployable kernel set, a shape routing table, and the feedback state for the next round. The objective is to maximize the geometric-mean speedup of this portfolio under the budget. The formulation asks three things at once: how to partition the shape space so that long-tail inputs are not missed and the partition migrates with the hardware specification; which kernels to keep under a budget on compilation time, code size, and routing overhead so that marginal gain per unit cost is maximized; and how to identify, from online execution, the residuals that one offline round leaves behind and repair them.

Given the shape-dependent bottlenecks of a single operator, we argue that **strong kernel performance requires *shape-aware composition*: covering the shape space with a small budgeted kernel set rather than one kernel or one kernel per shape.**

Based on this formulation and this claim, we propose KernelWeaver, a shape-aware kernel agent that performs three tightly coupled tasks. First, it derives continuous shape boundaries from hardware properties, including the SM count, SRAM capacity, register resources, peak compute throughput, and memory bandwidth, and generates candidates for both regular regions and extreme-shape families. Second, it profiles every candidate across the target shape space, selects a complementary portfolio by marginal performance gain per unit cost under the deployment budget, and constructs a shape-aware routing table. Third, after deployment, it analyzes routing feedback to identify low-gain shapes, unused candidates, and cross-family kernels, and directs targeted refinement through continued sampling, candidate down-weighting or pruning, and family splitting. The resulting candidates and deployment feedback are then fed into the next optimization round.

This paper makes the following contributions.

- We identify a cross-shape coverage problem in kernel-agent workflows and quantify the cost of addressing it through specialization. A single generated kernel exhibits performance regressions of up to 71.40% relative to PyTorch eager on Softmax, while expanding from one to 30 kernels across four operators improves geometric-mean speedup by 14.13–25.55% at 3.89–$13.18\times$ the compilation cost.
- We present KERNELWEAVER, a shape-aware kernel agent that constructs compact portfolios of complementary kernels under deployment budgets. It guides candidate generation with hardware-derived shape partitions, selects kernels by marginal performance gain per unit cost, and uses execution feedback from shape-aware routing to direct subsequent refinement toward poorly served inputs.
- We evaluate KERNELWEAVER on nine common GPU kernel workloads across the RTX 4090 and A800. It achieves geometric-mean speedups of $1.59\times$ and $1.44\times$ over PyTorch eager, compared with $1.27\times$ and $1.12\times$ for the strongest baseline, respectively, and ranks first in every workload–GPU pair. Ablation studies establish the contribution of each component, with shape-aware candidate generation accounting for the largest gain.

2 RELATED WORK

2.1 GPU KERNEL OPTIMIZATION

GPU kernel performance is jointly determined by thread-block configuration, memory access patterns, register usage, and Tensor core utilization. Vendor libraries such as cuBLAS and cuDNN (Tillet et al., 2019) provide hand-tuned implementations for common operators, while compiler and DSL approaches such as TVM/Ansor (Chen et al., 2018a;b; Zheng et al., 2020) and Triton (Tillet et al., 2019) search tile shapes and memory layouts for a given computation, typically committing to a single implementation per operator. Because the input shapes of one operator span a wide range, dominant bottlenecks differ across shapes. Shape specialization has therefore been explored in Roller (Zhu et al., 2022), which enumerates discrete tile configurations, and DietCode (Zheng et al., 2022), which

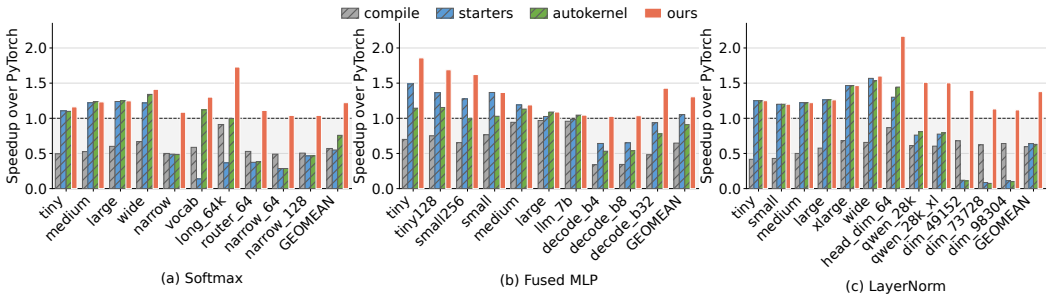

Figure 1: Cross-shape performance of kernel optimization methods.

clusters shapes but derives cluster boundaries from per-device runtime measurements, while the max-autotune mode of torch.compile (Ansel et al., 2024) tunes each shape independently. All of them pay a search or measurement cost that grows with the number of shapes or devices: tuning every shape separately is expensive (Li et al., 2020; Zhang et al., 2021; Ahn et al., 2020), and a boundary measured on one device does not transfer to another (Yu et al., 2021; Lee et al., 2025).

2.2 Kernel agents

Conventional kernel development requires experts to design algorithms, write code, tune parameters, and iterate on measurements, which cannot keep pace with rapidly evolving models and operators. A kernel agent instead organizes the whole development loop into an LLM-driven iterative workflow: it generates candidate kernels from an operator specification, compiles them, verifies correctness against a reference implementation, measures performance on the target device, and uses the resulting profile and error signals to guide the next round of generation (Yao et al., 2023). CUDA Agent, KDA, and AutoKernel (Dai et al., 2026; Jaber & Jaber, 2026) adopt this paradigm, and benchmarks such as KernelBench (Ouyang et al., 2025) standardize the evaluation of generated kernels. Because compilation and correctness feedback are cheap relative to human iteration, these systems can explore many candidate implementations of a single operator. Their evaluation, however, typically reports speedup over baselines such as PyTorch eager on a few representative shapes, so a kernel that regresses on other shapes is not penalized. Recent variants extend the loop with improved search strategies (Ran et al., 2026; Wiedemann et al., 2026; Pan et al., 2025), retrieval of similar kernels (Mavalankar et al., 2025), or hardware-aware prompting (Zhang et al., 2025; Wei et al., 2025a;b), yet they still optimize each operator in isolation and report speedup on a few representative shapes.

3 Motivation

3.1 Single-kernel optimization is insufficient

Observation #1: A single kernel optimized for a small set of representative shapes cannot maintain stable performance across the full shape space. Once generated, a kernel fixes its thread-block configuration, tile shape, reduction strategy, and memory-access layout, whereas different shapes impose structurally different requirements on parallelism, memory bandwidth, tensor core throughput, and on-chip storage.

To validate this observation, we build an evaluation suite spanning regular inference workloads (Kwon et al., 2023), long-context attention score computation (Dao et al., 2022; Xiao et al., 2024), language-model output projection (Frantar et al., 2022; Xiao et al., 2023), ultra-long contexts (Hooper et al., 2024; Liu et al., 2024), and extremely narrow inputs in MoE routing and expert dispatch (Zhou et al., 2022). With PyTorch eager (Paszke et al., 2019) as the baseline, as shown in Figure 1, a single kernel produced by existing kernel agent workflows is slower than eager on 5 of 10 Softmax shapes, with a maximum degradation of 71.40% and a geometric-mean speedup of 0.76×; on 4 of 10 fused-MLP shapes, with a maximum degradation of 46.88% and a geometric mean of 0.91×; and on 5 of 12 LayerNorm shapes, with a maximum degradation of 92.66% and a geometric mean of only 0.63×. A kernel tuned on representative shapes therefore fails on a large fraction of the space, and its gains on the remaining shapes do not offset these regressions.

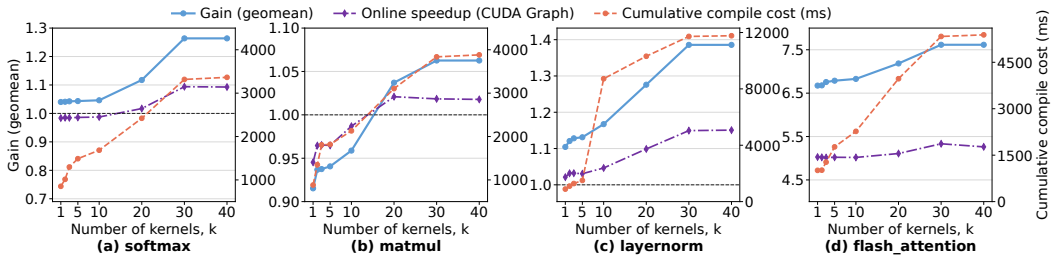

Figure 2: Performance-compilation cost trade-offs in kernel portfolio expansion.

3.2 Per-shape kernel generation is too expensive

Observation #2: Multiple specialized kernels improve aggregate performance, but cost grows faster than the gain. To mitigate the mismatch of a single kernel, a direct approach is to partition the shape space more finely and generate a specialized kernel for each region, with better-matched tiling, memory access, and reduction.

Figure 2 shows the trade-off on a separate scaling study suite. Raising the kernel count from 1 to 30 improves the geometric-mean speedup of Softmax, MatMul, LayerNorm, and FlashAttention by only 14.13% to 25.55%, and by 6.13% to 12.64% end-to-end with CUDAGraph; cumulative compilation cost grows to 3.89×–13.18× and code size to 27.81×–34.02× that of the single-kernel configuration. Each additional kernel must be generated, compiled, verified, and evaluated, and it increases code maintenance and online routing overhead. The gain is real but marginal relative to the cost it incurs.

3.3 From more kernels to kernel composition

Insight: Different kernels excel on different input shapes. A kernel with modest average performance can still be valuable if it accelerates shapes that other kernels handle poorly. Even a kernel with modest average performance can be valuable if it accelerates shapes that the current portfolio serves poorly. This perspective helps explain the diminishing gains from 30 to 40 kernels in Figure 2: additional kernels offer little benefit when their strengths overlap with those already represented. The resulting optimization principle is to select kernels by their marginal contribution to portfolio-level performance per unit cost.

Challenge: How can we effectively combine complementary kernels to improve performance across diverse input shapes? The system must cover the shape space with a small kernel set that maximizes geometric-mean speedup under joint budgets on kernel count, compilation time, code size, and routing overhead. This requires jointly reasoning about the complementarity and system cost of candidates, which decomposes into three questions: where complementary candidates come from, which subset to keep under a given deployment budget, and how to handle the shapes that a single offline round leaves uncovered.

4 KernelWeaver

4.1 System overview

KernelWeaver is a shape-aware kernel agent designed to optimize an operator across a specified range of input shapes on a target GPU. Given the operator specification, GPU characteristics, shape range, and deployment budget, it generates a compact set of complementary kernels and a routing table that assigns input shapes to them. It also retains execution feedback to guide refinement in subsequent rounds. Figure 3 summarizes the three components that realize this workflow.

The workflow begins with **shape-aware candidate generation**, which derives regions from hardware constraints and generates kernels for regular regions and extreme-shape families. These candidates are combined with hand-written starters and a kernel generated without shape partitioning. **Cost-aware portfolio selection and routing** then profiles each candidate across the target shapes, selects a complementary subset under the deployment budget, and constructs a routing table. Finally,

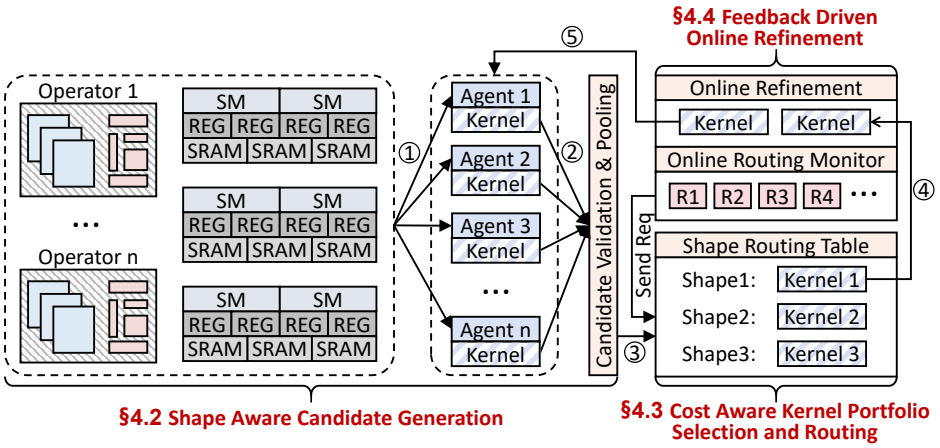

Figure 3: System overview.

feedback-driven online refinement uses deployment observations to identify poorly served shapes and guide the next round of generation and selection.

4.2 Shape-aware candidate generation

Insight and approach: For a given GPU operator, the best kernel configuration for an input shape can differ across devices because SM count, on-chip storage, and tensor core granularity differ. We derive hardware-dependent boundaries to generate kernel candidates for regular input shapes, while generating separate kernel candidates for extreme-shape families. The process comprises shape partitioning, kernel generation, and candidate validation and pooling.

Shape partition: To guide kernel generation, KernelWeaver reads the GPU's SM count S, shared memory per block M_{smem}, registers per SM, tensor core atom (m_0, n_0, k_0), peak throughput, memory bandwidth, and L2 capacity. For a tile (B_m, B_n, B_k), pipeline depth p, and element size b, three checks identify feasible configurations and likely bottlenecks:

- **Resource feasibility.** The tile uses $p(B_m B_k + B_k B_n)b$ bytes of shared memory. Configurations exceeding M_{smem} or the register budget at the target occupancy are excluded from generation by the kernel agent before expensive compilation begins.
- **Tensor core efficiency.** If an input dimension is smaller than the corresponding tensor core atom dimension, padding may waste work and lower effective throughput; the agent is prompted to consider a CUDA-core kernel for these narrow inputs.
- **Parallelism and memory behavior.** Fewer than S output tiles indicate limited parallelism. A working set larger than L2 with arithmetic intensity below peak throughput divided by memory bandwidth indicates streaming access. These labels guide the agent's tiling and memory-access choices during subsequent kernel generation.

For FP16 MatMul with B_k=64, p=2, and b=2, shared-memory use is $256(B_m{+}B_n)$ bytes. A 100 KiB limit gives $B_m{+}B_n \leq 400$, excluding $(256, 256)$; a 160 KiB limit gives $B_m{+}B_n \leq 640$, allowing it under the shared-memory check. If the selected Tensor Core atom has n_0=16, shapes with N<16 are flagged for CUDA-core generation. Thus, the same rules give the agent GPU-specific generation guidance without requiring fixed, device-specific shape thresholds.

Kernel agent generation: For each regular shape region, KernelWeaver provides the kernel agent with the hardware constraints defining that region, the corresponding GPU specifications, and the configurations excluded by those constraints. The agent uses this context to generate a candidate kernel, then compiles and verifies it, measures its latency, and revises the code based on the results. For extreme-shape families, specialized generation is invoked selectively. The agent first compares kernels in the existing pool with a kernel from a single generation pass. If no pooled candidate outperforms this reference on a family, the agent uses the measured performance difference to guide the generation of a specialized kernel for that family.

Candidate validation and pooling: The kernels generated above are checked for structural validity, compliance with the target GPU's hardware limits, and numerical correctness. We then measure their latency under the same conditions as PyTorch eager. Candidates that pass validation and meet the performance admission criterion are retained; the rest are discarded. For each retained kernel, we record its source code, shape region or family, applicable shape range, latency, compilation time, and code size. These kernels are combined with hand-written starters and the single-pass kernel to form the pool for portfolio selection.

4.3 Cost-aware kernel portfolio selection and routing

Insight and approach: A kernel's value depends on which shapes it improves beyond the current portfolio, not only on its standalone speed. Profiling every candidate across the target shapes makes these marginal gains measurable, allowing us to select complementary kernels under a deployment budget. The module comprises full-shape profiling, budget-aware greedy selection, and shape routing.

Full shape profiling: We measure the latency of every candidate $c \in \mathcal{C}$ on every shape $s \in \mathcal{S}$, forming a candidate–shape latency matrix. The speedup for each pair is

$$R(c, s) = \frac{T_{\text{eager}}(s)}{T(c, s)}. \tag{1}$$

For a selected portfolio $\mathcal{K} \subseteq \mathcal{C}$, the speedup on shape $s \in \mathcal{S}$ is $R(\mathcal{K}, s) = \max_{c \in \mathcal{K}} R(c, s)$. The portfolio score is the geometric mean of these speedups over $\mathcal{S}$. The single-pass kernel serves as the single-kernel baseline. The oracle selects the best candidate from $\mathcal{C}$ independently for each shape, providing an upper bound within the candidate pool without deployment budget constraints.

Budget-aware greedy selection: The candidate–shape latency matrix lets us measure how much each candidate improves the current portfolio. We use this marginal gain to select kernels under a deployment budget that caps kernel count, cumulative compilation time, and code size. Compilation time and code size are each normalized by the corresponding cost of a single kernel. Routing cost, measured by the number of routing-table intervals, is normalized against single-kernel routing. We sum these three normalized costs without separate weights and rank candidates by marginal gain in geometric-mean speedup per unit cost. Any candidate that would exceed the budget is skipped.

We initialize the portfolio with the budget-feasible candidate that achieves the highest geometric-mean speedup on its own. At each subsequent step, we add the candidate with the highest marginal-gain-per-cost score, provided it remains within the budget. Selection stops when no feasible candidate offers a gain greater than its added cost. Finally, we remove any selected kernel that is not the fastest on any measured shape, without reducing the portfolio's measured speedup.

Shape router: Once the portfolio is selected, KernelWeaver organizes its kernels into a routing table that associates shape intervals and match priorities with kernels. For each input, the router first checks for an exact shape match, then for an interval match whose hardware-derived bounds contain every input dimension. If neither matches, it considers the nearest interval and falls back to PyTorch eager when no kernel is applicable. The selected kernels and routing table are deployed together.

4.4 Feedback driven online refinement

Insight and approach: The portfolio is selected from a finite offline shape suite, but deployment may expose hold-out shapes or unexpected slowdowns. The shape router provides the inputs and latency measurements needed to locate these gaps. KernelWeaver uses this feedback to direct the kernel agent toward affected candidates and shape families, rather than restarting the full search. This feedback loop comprises online routing monitoring, residual analysis, and candidate refinement.

Online routing monitor: KernelWeaver records each request's input shape, selected kernel, latency, and speedup over PyTorch eager. Comparison with the offline candidate–shape latency matrix reveals routing discrepancies, unprofiled shapes, and unexpected slowdowns.

Residual analysis: Combining routing logs with the offline matrix, we identify three residual classes. A shape is low-gain if its routed speedup falls below the single-kernel baseline or substantially below

the per-shape oracle. A candidate is unused if it wins on no profiled shape. A kernel is cross-family if it wins in distinct shape regions, such as narrow and streaming regions.

Candidate refinement: These residual classes guide refinement. Low-gain shapes return to their regular region or extreme family for generation and rewriting. Unused candidates are pruned during pool merging or, if potentially useful, receive lower generation priority and fewer rewrite attempts. The agent creates separate prompts and candidates for each family of a cross-family winner.

5 EVALUATION

We evaluate KernelWeaver on nine common operators. We ask whether shape-aware kernel generation and composition improve performance across diverse input shapes, how much each module contributes, and whether the gains remain stable under different selection settings.

5.1 EXPERIMENTAL SETUP

Hardware. The experiments run on an RTX 4090 and an A800. Every method uses the same hardware and software environment, and each kernel is timed on one fixed GPU.

Operators. The nine common operators are Softmax, MatMul, Cross Entropy, FlashAttention Dao et al. (2022), Fused MLP, LayerNorm, Reduce, RMSNorm, and Rotary Embedding. Each shape suite covers regular inference, long sequences, wide and narrow matrices, deep reductions, vocabulary projections, and long-tail inputs to capture diverse shape-dependent performance regimes.

Baselines. We evaluate against three baselines. `torch.compile` uses `max-autotune`; we report its per-shape compilation time separately from online speedup. Starter Kernel is the initial single kernel of the generation workflow. AutoKernel (Jaber & Jaber, 2026) returns one kernel after 25 iterations, without shape partitioning or kernel composition.

Metrics. Let $T_m(s)$ and $T_{\text{eager}}(s)$ denote the measured latencies of method m and PyTorch eager on input shape s, respectively. We define the per-shape speedup and aggregate it over the N evaluated shapes of each operator using the geometric mean:

$$\text{Speedup}(m, s) = \frac{T_{\text{eager}}(s)}{T_m(s)}, \qquad \text{GeoMean}(m) = \left(\prod_{s=1}^{N} \text{Speedup}(m, s) \right)^{1/N}. \tag{2}$$

We also report compile time, the wall-clock time spent producing the kernels a method deploys for an operator (Appendix A.1). It is recorded separately and is not part of the speedup.

All methods share the shapes, data types, layouts, correctness checks, and timing procedure, with a common warm-up and synchronization before measurement. End-to-end latency includes online routing and graph replay, whereas ablations use fixed-shape offline timings to isolate module effects; their absolute speedups are not directly comparable. The generation model is GPT-5.6 Sol. The software stack is NGC PyTorch 24.04, PyTorch 2.3.0a0, and Triton 3.0.0 (Tillet et al., 2019). We also record the kernel count, cumulative compilation time, code size, and routing overhead.

5.2 OVERALL PERFORMANCE

KernelWeaver is the best method on both GPUs, and the gain is coverage of shapes a single kernel misses, not a higher peak on shapes a single kernel handles well. As shown in Figure 4, we evaluate performance under online deployment against the strongest single-kernel baseline, and the geometric-mean speedup rises from $1.27\times$ to $1.59\times$ on the RTX 4090 and from $1.12\times$ to $1.44\times$ on the A800. The baselines reach eager on only three or four of the nine operators. KernelWeaver does so on nearly all. The operators that move most are Cross Entropy, LayerNorm, and Reduce, the narrow-matrix, deep-reduction, and vocabulary-projection cases where Section 3 showed a single kernel failing.

The ranking follows the device rather than staying fixed. `torch.compile` falls from $1.12\times$ on the RTX 4090 to $0.71\times$ on the A800, so KernelWeaver is more than twice as fast as `torch.compile` on the A800 while staying above eager on both GPUs. Cross Entropy gains more on the A800, whereas MatMul and Rotary Embedding gain more on the RTX 4090. The shape boundaries are recomputed from the GPU specification rather than carried from one device to the other.

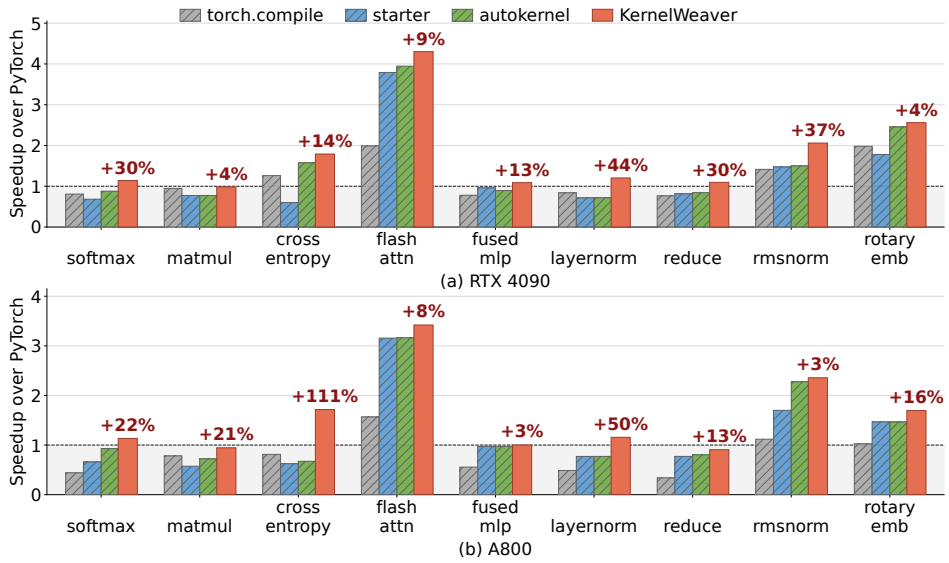

Figure 4: Geometric-mean speedup over PyTorch eager, for nine operators on two GPUs.

The speedup is not bought by compiling many kernels. All nine operators measure 96 candidates over the shape suite and deploy 28 of them, two to four per operator, cutting the Triton source from 14,955 lines to 4,131. Compiling those 28 kernels from scratch costs 13.53 s, about 1.61× the compilation time of Starter Kernel and 1.58× the compilation time of AutoKernel and 12.06× below the 163.20 s that `torch.compile` spends on the same operators. The constraint costs little: 3.43× fewer kernels are worth 3.7% of the geometric mean (Appendix A.1). The frozen system also generalizes beyond the suite, reaching 1.58× over eager on 63 shapes designed from scratch, above every baseline and at 94.61% of the per-shape oracle (Appendix A.2).

5.3 Ablation Study

To assess each module's contribution, we add them sequentially to AutoKernel and evaluate under offline settings. Here, the geometric mean aggregates results across the nine operators rather than reporting per-platform results. Figure 5 shows speedup rising from 1.53× for AutoKernel to 1.93×, 2.03×, and 2.10×. On a logarithmic scale, candidate generation contributes roughly three quarters of the gain, portfolio selection about one sixth, and online refinement the remainder. Candidate generation delivers the largest improvement; selection and refinement capture gains while keeping deployment costs in check.

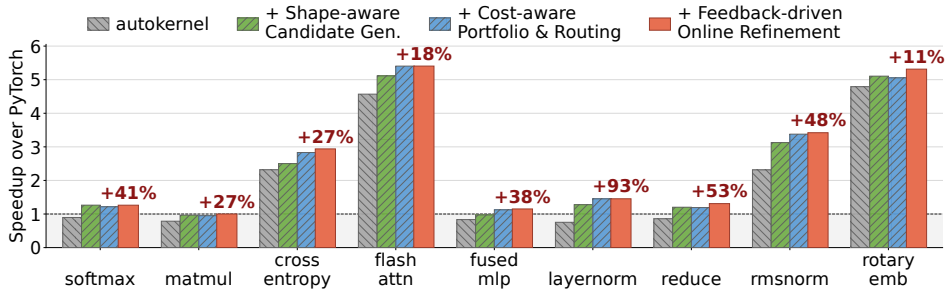

Figure 5: Speedup as the three modules are added one at a time.

Shape-aware candidate generation raises the aggregate geometric-mean speedup from 1.53× to 1.93× and increases the number of operators outperforming PyTorch eager from four to seven. Softmax and Reduce cross the 1.00× threshold, while LayerNorm rises from 0.75× to 1.28×, a 69.50% gain. The pronounced improvement on LayerNorm is consistent with the motivation for

shape-aware generation: as reduction depth varies, a single kernel configuration is unlikely to serve the full input range equally well. Generating candidates for different shape regimes expands the set of configurations available to the later selection stage.

Cost-aware kernel portfolio selection and routing raises the aggregate speedup from $1.93\times$ to $2.03\times$ and brings Fused MLP above eager, from $0.97\times$ to $1.13\times$. Cross Entropy rises from $2.50\times$ to $2.83\times$; selection accounts for more than half of its total improvement, indicating that its candidates provide complementary gains across shapes. The benefit is not uniform: Softmax falls from $1.27\times$ to $1.22\times$. This illustrates the portfolio's trade-off: retaining fewer kernels can sacrifice some per-operator speedup while making the candidate pool deployable under budget.

Feedback-driven online refinement raises the aggregate speedup from $2.03\times$ to $2.10\times$ by revisiting performance gaps left after selection. Reduce gains 9.90%, and Rotary Embedding rises from $5.05\times$ to $5.31\times$, with refinement contributing nearly half of Rotary Embedding's total absolute speedup increase. FlashAttention shows no change, suggesting that refinement is most useful where the earlier stages leave room for improvement. MatMul also moves from $0.95\times$ to $1.001\times$; although this brings the final operator above eager, the margin is too small to interpret as a substantial speedup.

5.4 Sensitivity analysis

We quantify the overhead of routing each request and replaying its graph relative to direct invocation. Table 1 reports results across 209 shapes. For 188 covered shapes, exact-match lookup takes 0.17–$0.19\,\mu s$; routing adds a median 0.30% of wall time and at most 2.76%. The remaining 21 shapes require nearest-neighbor lookup, taking 10.34–$21.69\,\mu s$ and imposing a larger cost on short-running kernels. Across all shapes, the routed-minus-direct difference has a median of $0.66\,\mu s$ and a 90th percentile of $1.41\,\mu s$. MatMul's negative median reflects timing drift among the 14 shapes taking over $10\,ms$, which swing by several microseconds per batch, far beyond the lookup cost; the 15 shapes below $10\,ms$ instead show a median overhead of $+0.25\,\mu s$, the nominal overhead. Overall, the routing overhead remains a small fraction of kernel runtime.

Table 1: Per-operator routing overhead measured by routed vs direct invocation difference.

Operator	n	Hit/miss	Dispatch (µs) p50 / p90 / max	Fallback (µs)	Wall share median / worst	Kernel share median
Softmax	25	21/4	0.72 / 1.06 / 1.60	14.06	0.81% / 1.93%	3.31%
MatMul	32	29/3	−0.04 / 2.50 / 35.27	21.69	0.00% / 1.05%	0.00%
Cross Entropy	25	23/2	0.72 / 1.33 / 2.20	14.65	0.71% / 1.43%	2.26%
FlashAttention	22	20/2	0.93 / 1.36 / 7.08	16.32	0.66% / 1.01%	2.29%
Fused MLP	23	21/2	0.77 / 9.71 / 20.72	16.10	0.08% / 0.80%	0.08%
LayerNorm	23	21/2	0.15 / 0.92 / 1.02	14.04	0.09% / 1.12%	0.08%
Reduce	17	15/2	0.71 / 1.06 / 1.10	10.34	0.69% / 1.14%	6.45%
RMSNorm	21	19/2	0.27 / 1.39 / 2.49	12.76	0.02% / 2.76%	0.02%
Rotary Embedding	21	19/2	0.76 / 1.21 / 1.62	15.91	0.71% / 1.51%	1.74%
Total	**209**	**188/21**	**0.66 / 1.41 / 35.27**	**10.34–21.69**	**0.30% / 2.76%**	**0.71%**

We also measure compilation cost under two bounding isolation protocols; the reported $13.53\,s$ corresponds to the deployment path (Appendix A.3). Sensitivity to the admission margin and greedy stopping threshold is reported in Appendix A.4.

6 Conclusion

We presented KernelWeaver, a shape-aware kernel agent that generates and composes a compact portfolio of complementary kernels under deployment budgets. Rather than optimizing a single kernel for representative shapes, KernelWeaver uses hardware-derived boundaries to guide candidate generation, selects kernels for their complementary gains, and refines poorly served shapes using deployment feedback. Across nine operators, it achieves geometric-mean speedups over PyTorch eager of $1.59\times$ on the RTX 4090 and $1.44\times$ on the A800, compared with $1.27\times$ and $1.12\times$ for the strongest evaluated baseline on each GPU. KernelWeaver leads in every operator–GPU combination while deploying only two to four kernels per operator.

AI USE STATEMENT

Large language models were used during manuscript preparation to assist with grammar correction, sentence restructuring, and clarity of presentation. Their use for kernel generation and refinement is part of the proposed methodology and is described in the main text. The authors were responsible for the research design, experimental evaluation, interpretation of results, and conclusions. All AI-assisted manuscript content was reviewed and verified by the authors, who take full responsibility for the accuracy and integrity of the final work.

REFERENCES

Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. Chameleon: Adaptive code optimization for expedited deep neural network compilation. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=rygG4AVFvH.

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

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=HylVB3AqYm.

Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=HylxE1HKwS.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, Carlsbad, CA, October 2018a. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/chen.

Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.,

2018b. URL https://proceedings.neurips.cc/paper_files/paper/2018/file/8b5700012be65c9da25f49408d959ca0-Paper.pdf.

Krzysztof Marcin Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Quincy Davis, Afroz Mohiuddin, Lukasz Kaiser, David Benjamin Belanger, Lucy J Colwell, and Adrian Weller. Rethinking attention with performers. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=Ua6zuk0WRH.

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, 2026. URL https://arxiv.org/abs/2602.24286.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 16344–16359. Curran Associates, Inc., 2022. doi: 10.52202/068431-1189. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/67d57c32e20fd0a7a302cb81d36e40d5-Paper-Conference.pdf.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *CoRR*, abs/2210.17323, 2022. URL https://doi.org/10.48550/arXiv.2210.17323.

Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W. Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 1270–1303. Curran Associates, Inc., 2024. doi: 10.52202/079017-0040. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/028fcbcf85435d39a40c4d61b42c99a4-Paper-Conference.pdf.

Jaber Jaber and Osama Jaber. Autokernel: Autonomous gpu kernel optimization via iterative agent-driven search, 2026. URL https://arxiv.org/abs/2603.21331.

Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=rkgNKkHtvB.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pp. 611–626, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Seonho Lee, Amar Phanishayee, and Divya Mahajan. Forecasting gpu performance for deep learning training and inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, pp. 493–508. ACM, March 2025. doi: 10.1145/3669940.3707265. URL http://dx.doi.org/10.1145/3669940.3707265.

Menghao Li, Minjia Zhang, Chi Wang, and Mingqin Li. Adatune: Adaptive tensor program compilation made efficient. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 14807–14819. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/a9b7ba70783b617e9998dc4dd82eb3c5-Paper.pdf.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. KIVI: A tuning-free asymmetric 2bit quantization for KV cache. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and

Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 32332–32344. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/liu24bz.html.

Aditi Mavalankar, Hassan Mansoor, Zita Marinho, Mariia Samsikova, and Tom Schaul. Aupair: Golden example pairs for code repair. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=GmqZ3WvkeV.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Re, and Azalia Mirhoseini. Kernelbench: Can LLMs write efficient GPU kernels? In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=yeoN1iQT1x.

Haolin Pan, Hongyu Lin, Haoran Luo, Yang Liu, Kaichun Yao, Libo Zhang, Mingjie Xing, and Yanjun Wu. Compiler-r1: Towards agentic compiler auto-tuning with reinforcement learning. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 88289–88311. Curran Associates, Inc., 2025. doi: 10.52202/085713-2951. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/7fc0aac0d700873ac520af9b05e3f2b6-Paper-Conference.pdf.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

Dezhi Ran, Shuxiao Xie, Mingfang Ji, Anmin Liu, Mengzhou Wu, Yuan Cao, Yuzhe Guo, Hao Yu, Linyi Li, Yitao Hu, Wei Yang, and Tao Xie. Kernelband: Steering LLM-based kernel optimization via hardware-aware multi-armed bandits. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=d8e2jBfMpE.

Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, pp. 10–19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367196. doi: 10.1145/3315508.3329973. URL https://doi.org/10.1145/3315508.3329973.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

Anjiang Wei, Allen Nie, Thiago S. F. X. Teixeira, Rohan Yadav, Wonchan Lee, Ke Wang, and Alex Aiken. Improving parallel program performance with LLM optimizers via agent-system interfaces. In *Forty-second International Conference on Machine Learning*, 2025a. URL https://openreview.net/forum?id=3h80HyStMH.

Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. Astra: A multi-agent system for gpu kernel performance optimization, 2025b. URL https://arxiv.org/abs/2509.07506.

Nina Wiedemann, Quentin Leboutet, Michael Paulitsch, Diana Wofk, and Benjamin Ummenhofer. Kernelfoundry: Hardware-aware evolutionary GPU kernel optimization. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=4wy2YAHKRF.

Guangxuan Xiao, Ji Lin, Mickaël Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *ICML*, pp. 38087–38099, 2023. URL https://proceedings.mlr.press/v202/xiao23c.html.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 21875–21895, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/5e5fd18f863cbe6d8ae392a93fd271c9-Paper-Conference.pdf.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Geoffrey X. Yu, Yubo Gao, Pavel Golikov, and Gennady Pekhimenko. Habitat: A Runtime-Based computational performance predictor for deep neural network training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 503–521. USENIX Association, July 2021. ISBN 978-1-939133-23-6. URL https://www.usenix.org/conference/atc21/presentation/yu.

Minjia Zhang, Menghao Li, Chi Wang, and Mingqin Li. Dynatune: Dynamic tensor program optimization in deep neural network compilation. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=GTGb3M_KcUl.

Zijian Zhang, Rong Wang, Shiyang Li, Yuebo Luo, Mingyi Hong, and Caiwen Ding. Cudaforge: An agent framework with hardware feedback for cuda kernel optimization, 2025. URL https://arxiv.org/abs/2511.01884.

Bojian Zheng, Ziheng Jiang, Cody Hao Yu, Haichen Shen, Joshua Fromm, Yizhi Liu, Yida Wang, Luis Ceze, Tianqi Chen, and Gennady Pekhimenko. Dietcode: Automatic optimization for dynamic tensor programs. In D. Marculescu, Y. Chi, and C. Wu (eds.), *Proceedings of Machine Learning and Systems*, volume 4, pp. 848–863, 2022. URL https://proceedings.mlsys.org/paper_files/paper/2022/file/f89b79c9a28d4cae22ef9e557d9fa191-Paper.pdf.

Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating High-Performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 863–879. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/zheng.

Yanqi Zhou, Tao Lei, Hanxiao Liu, Nan Du, Yanping Huang, Vincent Zhao, Andrew M Dai, zhifeng Chen, Quoc V Le, and James Laudon. Mixture-of-experts with expert choice routing. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 7103–7114. Curran Associates, Inc., 2022. doi: 10.52202/068431-0515. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/2f00ecd787b432c1d36f3de9800728eb-Paper-Conference.pdf.

Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. ROLLER: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 233–248, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/zhu.

A Appendix

A.1 Deployment Budget

We assess whether shape-aware kernel composition improves performance without making deployment prohibitively expensive. Table 2 shows that, across nine operators, KernelWeaver profiles 96 candidates but deploys only 28, or two to four per operator. Selecting this portfolio reduces Triton source code from $14{,}955$ to $4{,}131$ lines. Compiling the deployed kernels from scratch takes 13.53 s in total, about $1.61\times$ the compilation time of Starter Kernel and $1.58\times$ the compilation time of AutoKernel, but only one-twelfth of the 163.20 s required by `torch.compile` on the same operators. The additional time relative to the single-kernel baselines is concentrated in operators retaining multiple kernels, such as FlashAttention and Fused MLP. Compared with deploying the full candidate pool, the portfolio uses $3.43\times$ fewer kernels at a cost of 3.7% in geometric-mean speedup (Section 5.3). Routing overhead is also limited for covered shapes: exact lookup takes 0.17–0.19 µs, adding a median 0.30% of wall time and at most 2.76% on a launch-bound 6.1 µs kernel. The 21 uncovered shapes instead require nearest-neighbor lookup at 10.34–21.69 µs, making this fallback the main source of routing overhead.

Table 2: Per-operator deployment budget: evaluated kernel candidates, deployed kernels, Triton code lines, and from-scratch compilation latency on a single device.

	Kernels		LoC		Compile time (s)			
Operator	Pool	k	Pool	k	Starter	AutoKernel	`torch.compile`	KernelWeaver
Softmax	17	2	3,955	644	0.89	0.93	18.40	0.97
MatMul	14	3	2,447	509	0.89	0.94	11.30	1.66
Cross Entropy	10	3	845	233	0.87	0.84	19.20	1.30
FlashAttention	9	3	2,474	688	1.29	1.44	21.70	2.32
Fused MLP	8	4	1,325	667	1.09	1.05	24.40	1.98
LayerNorm	11	3	1,058	287	0.86	0.91	20.20	1.42
Reduce	8	3	1,081	410	0.82	0.80	12.70	1.21
RMSNorm	11	4	789	287	0.85	0.84	17.10	1.52
Rotary Embedding	8	3	981	406	0.82	0.82	18.20	1.15
Total (9)	**96**	**28**	**14,955**	**4,131**	**8.38**	**8.57**	**163.20**	**13.53**

A.2 Generalization to Held-Out Shapes

To assess generalization beyond benchmark shapes, we freeze the 28 kernels in Table 2 and their routing table, then evaluate 63 held-out shapes without tuning. As shown in Table 3, the portfolio achieves a $1.58\times$ geometric-mean speedup over PyTorch eager, exceeding AutoKernel ($1.39\times$), copy-taxed `torch.compile` ($1.13\times$), and Starter ($1.09\times$). It reaches 94.61% of the per-shape oracle's performance. Only MatMul and Fused MLP remain below eager; even the oracle gains 3.0% and 1.1% on these operators, respectively, indicating limited headroom within the candidate pool.

Table 3: Generalization to 63 held-out shapes (7 per operator), with dimensions unseen in the training suite. The kernel portfolio and routing table remain fixed and never observe these shapes. Geometric-mean speedup over eager under the same evaluation protocol as Figure 4.

Operator	shape	Starter	AutoKernel	`torch.compile`	KernelWeaver	Oracle
Softmax	7	0.71	1.05	0.82	**1.20**	1.21
MatMul	7	0.84	0.84	0.91	**0.95**	1.03
Cross Entropy	7	0.49	1.65	1.20	**1.72**	1.80
FlashAttention	7	5.26	6.21	2.03	**6.20**	6.20
Fused MLP	7	0.91	0.90	0.85	**0.93**	1.01
LayerNorm	7	0.64	0.65	0.80	**1.11**	1.24
Reduce	7	0.96	0.99	0.84	**1.04**	1.05
RMSNorm	7	1.40	1.41	1.39	**1.81**	2.03
Rotary Embedding	7	1.84	2.57	2.04	**2.65**	2.71
GeoMean	**63**	**1.09**	**1.39**	**1.13**	**1.58**	**1.67**

A.3 Compilation Protocol

To assess the effect of compiler isolation on compilation time, Table 4 compares three protocols using the same 28 deployed kernels reported in Table 2. Under Protocol A, each kernel is compiled in a fresh process with a cold cache. Under Protocol B, the kernels are compiled sequentially in one warm process, amortizing initialization costs. Protocol C compiles the kernels in parallel but takes $2.16\times$ as long as Protocol A, as the CPU-bound workers contend for resources. The reported 13.53 s reflects the online deployment path: one cold kernel initialization followed by 27 warm compilations. This result falls between the measurements from Protocols A and B.

Table 4: Compile cost (seconds) for the 28 deployed kernels under three isolation protocols. The Reported row is our deployment path, which pays one cold initialization and 27 warm compiles.

Protocol	Isolation	Total (s)	Per kernel (s)
A	a fresh process and cache per kernel	26.10	0.93
B	one warm process, serial	11.10	0.40
C	one warm process, parallel	56.40	2.01
Reported	**one fresh process, first kernel cold**	**13.53**	**0.48**

A.4 Admission Margin

The admission margin decides how far a candidate must beat the baseline to enter the pool. Figure 6 sweeps it from 0% to 4%. The nine-operator geometric mean drops from $2.10\times$ to $1.99\times$, and the loss is uneven. Softmax and MatMul, whose candidates come from already optimized library kernels, do not move. Cross Entropy loses about 14.62%, and the reduction operators lose several percent, because those pools contain moderate candidates that a looser gate lets in. This matches the ablation: the pool determines the result, not the point at which selection stops.

Figure 7 sweeps the greedy stopping threshold, how large a marginal gain must be before another kernel is kept, from 0.02 to 0.20. The nine-operator geometric mean is flat to within 0.2% out to 0.05 and declines after that, while the kernel set shrinks from about 30 to 21. The threshold therefore trades compile cost against a small speed loss rather than reshaping the portfolio.

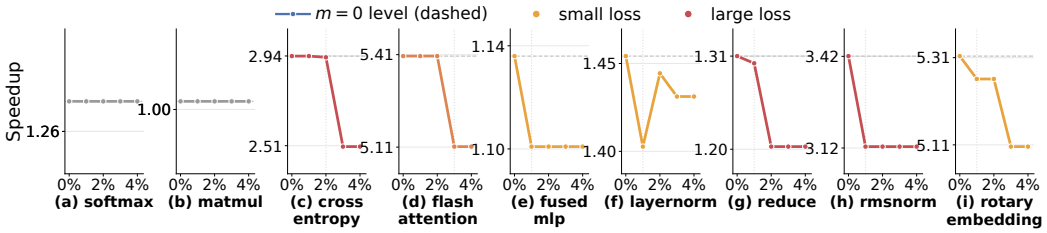

Figure 6: Effect of the candidate admission margin on per-operator speedup.

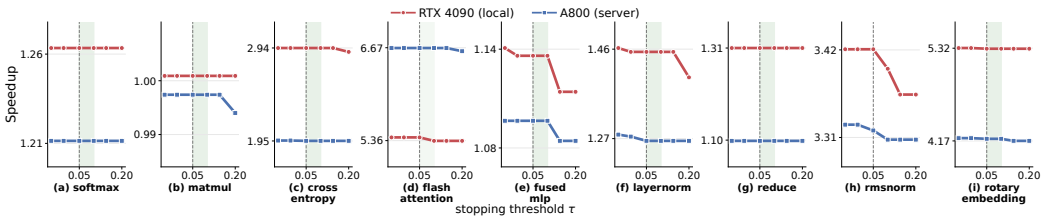

Figure 7: Effect of the greedy stopping threshold on per-operator speedup.