MKEvolve: A Modular Multi-Agent Framework for Kernel Code Generation

Jason Yoo [1 †] **Rajarshi Saha** [2] **Shaowei Zhu** [2] **Tao Yu** [2] **Wei Tang** [2] **Youngsuk Park** [2]

Abstract

Despite rapid progress in LLM-based code generation, writing correct and performant kernels for hardware accelerators remains a key bottleneck in scaling modern ML workloads. We present MKEVOLVE (**M**odular **K**ernel **E**volve), a framework that iteratively co-evolves a modular decomposition of complex PyTorch modules and the LLM-generated kernel for each submodule, refining the decomposition by splitting and fusing across iterations while independently improving each subkernel via LLM-driven beam search. The resulting kernels are programmatic compositions of independently verified subkernels, making them configurable (subkernel implementations are swappable), interpretable (errors and speedups are traceable to specific subkernels), and readily adaptable to related model architectures. Experiments with Triton on KernelBench L2 and L3, spanning multi-operator sequences and full model architectures, show that MKEVOLVE improves both correctness and speedup over end-to-end direct synthesis baselines while reducing LLM token usage by up to 35%.

1. Introduction

The rapid growth of modern machine learning workloads has sharply increased demand for efficient kernels that improve hardware utilization and throughput, directly reducing inference latency and energy cost. However, writing such kernels remains difficult even for domain experts since high performance depends on carefully orchestrating parallelism, memory movement, tiling, fusion, and numerical behavior in ways that are tightly coupled to both the target hardware and the specific structure of the workload. As a result, kernel development is often time-consuming, brittle, and inaccessible to most practitioners.

Recently, LLM-driven kernel generation has emerged as a promising way to reduce this burden, with Fisches et al. (2025); Li et al. (2025); Woo et al. (2026); Zhang et al. (2025); Wang et al. (2025) automatically synthesizing kernels from framework-level specifications such as Ouyang et al. (2025). However, the majority of existing approaches treat the kernel as a single monolithic artifact throughout the improvement loop, creating two key limitations: (i) repeatedly optimizing the entire kernel can be inefficient in terms of improvement effort and token usage since many useful improvements are local and affect only a small portion of the computation and (ii) the resulting solutions are often complex and difficult for both non-experts and LLMs to interpret, debug, or adapt.

While these challenges apply broadly to LLM-based kernel generation, this work focuses on Triton, a distinct and timely setting. Unlike CUDA, which benefits from abundant high-quality training data enabling effective post-training approaches such as CudaAgent (Dai et al., 2026), Triton lacks sufficient high-quality data. As a result, post-training methods are less effective and often exhibit reward hacking, as observed in TritonRL (Woo et al., 2026), making agentic test-time methods the current frontier for Triton synthesis.

This work introduces MKEVOLVE, a modular kernel generation framework that operates on kernels structured as compositions of subkernels, each solving an explicitly defined subproblem. Rather than optimizing a kernel end-to-end, MKEVOLVE repeatedly decomposes a complex PyTorch module into smaller submodules, independently improves kernels for each via LLM-driven beam search, and programmatically composes them into a complete implementation. This modularization makes the search space more structured and tractable: local improvements can target where they matter most, alternative implementations can be explored at the submodule level, and failures can be traced to specific subkernels rather than an opaque monolithic program. The resulting kernels are not only more reliable but also more interpretable and easier for engineers to modify, reuse, and extend to related architectural variants with common submodules.

Our contributions are threefold: (i) we introduce MKEVOLVE, the first kernel code generation framework that

[†]Work done during an internship at Amazon Web Services. [1]University of British Columbia [2]Amazon Web Services. Correspondence to: Jason Yoo <jasonyoo0116@gmail.com>, Rajarshi Saha <sahrajar@amazon.com>, Youngsuk Park <pyoungsu@amazon.com>.

Accepted to the 2nd Workshop on Compositional Learning at ICML 2026, Seoul, South Korea. Copyright 2026 by the author(s).

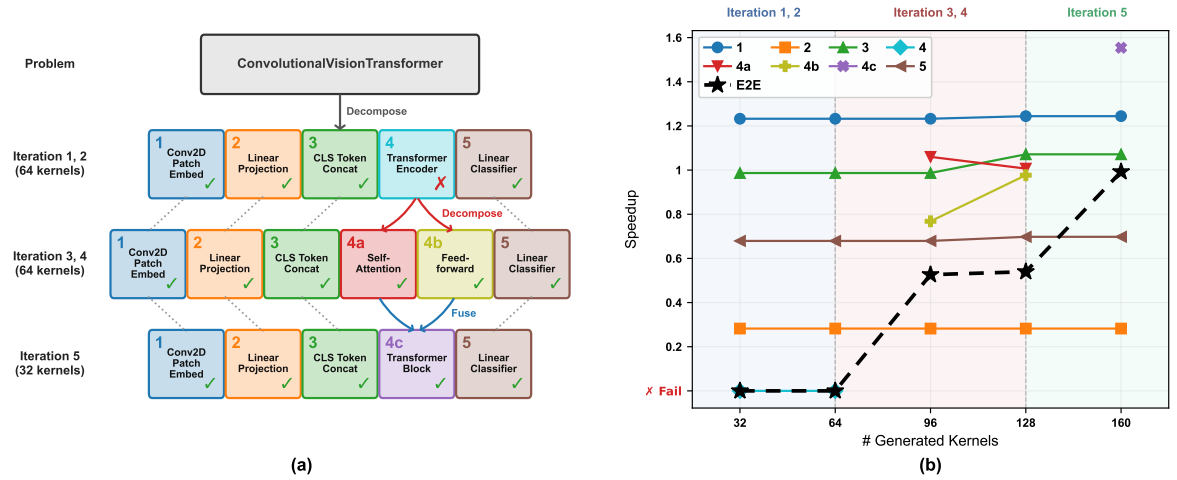

Figure 1. MKEVOLVE kernel synthesis on the KernelBench L3 ConvolutionalVisionTransformer task for 5 outer loop iterations. **Left**: Subproblem structure evolution during refinement, with checkmarks and crosses respectively denoting correct and invalid subkernels. MKEVOLVE iteratively decomposes the subproblem structure until all subproblems admit correct subkernels, then considers subproblem fusion. **Right**: Subkernel (Lines 1-5) and end-to-end (**E2E**) kernel speedup relative to *torch.compile* during refinement. End-to-end kernel speedup increases from failure to 0.53 after the Transformer Encoder subproblem (4) is decomposed into Self-Attention (4a) and Feed-forward (4b), then increases significantly again when (4a) and (4b) are merged into the Transformer Block subproblem (4c).

iteratively co-evolves a modular decomposition and independently optimized subkernels, targeting correctness while achieving speedup; (ii) we show that this compositional structure yields kernels that are interpretable, configurable, and readily transferable to architecturally related workloads; and (iii) we evaluate MKEVOLVE with Triton on KernelBench L2 and L3, yielding improvements in both correctness and runtime over direct synthesis baselines while reducing LLM token usage by up to 35%. Taken together, our results establish modularity as a key principle for LLM-driven kernel generation.[1]

2. Related Work

Post-training Recent works improve kernel generation by fine-tuning LLMs on kernel code datasets. KernelLLM (Fisches et al., 2025) and Concur (Kong et al., 2025) use supervised tuning on curated datasets, while Kevin (Baronio et al., 2025) applies reinforcement learning (RL) for correctness and performance. CUDA Agent (Dai et al., 2026) further advances this idea by training LLMs via large-scale agentic RL, combining data synthesis and execution feedback in a skill-augmented environment. AutoTriton (Li et al., 2025) and TritonRL (Woo et al., 2026) combine both; TritonRL also adds cheating detection, which we adopt in our pipeline. However, these require direct access to model weights, making them unusable with closed models that only expose API access. Moreover, MKEVOLVE is complementary, operating purely at inference time and is able to leverage stronger post-trained LLMs.

[1]Code will be released at https://github.com/amazon-science/ModularKernelEvolution.

Inference scaling A parallel line of work focuses on improving kernel quality at inference time by scaling LLM computation. At the simplest end, MultiKernelBench (Wen et al., 2025) studies one-shot prompting across platforms, while Lange et al. (2025) explore parallel sampling of independent candidates. More structured search strategies include beam search in AccelOpt (Zhang et al., 2025) and AutoComp (Hong et al., 2025), and evolutionary methods in AlphaEvolve (Novikov et al., 2025) and Avo (Chen et al., 2026), which iteratively refine candidate kernels. Multi-agent systems such as Astra (Wei et al., 2025), KernelFalcon (Wang et al., 2025), and KernelEvolve (Liao et al., 2026) partitioning the generation process across specialized agents. MKEVOLVE falls in this category but is distinguished by its iterative co-evolution of the decomposition structure and subkernel implementations, with beam search applied independently at the submodule level.

A closely related work is KernelFalcon (Wang et al., 2025), a multi-agent framework that constructs an LLM-generated JSON representation approximating the PyTorch compute graph, synthesizes subkernels for each subgraph using LLM-generated correctness tests, and composes them into an end-to-end kernel via an LLM. In contrast, MKEVOLVE introduces a key design distinction: it programmatically composes subkernels directly within the end-to-end kernel throughout the synthesis process in a verifiable manner. This enables both reliable reuse and flexible substitution of optimized subkernels across tasks.

Furthermore, MKEVOLVE adopts an iterative optimization strategy, evolving subkernels using beam search rather than stopping at the first correct solution and refining the sub-

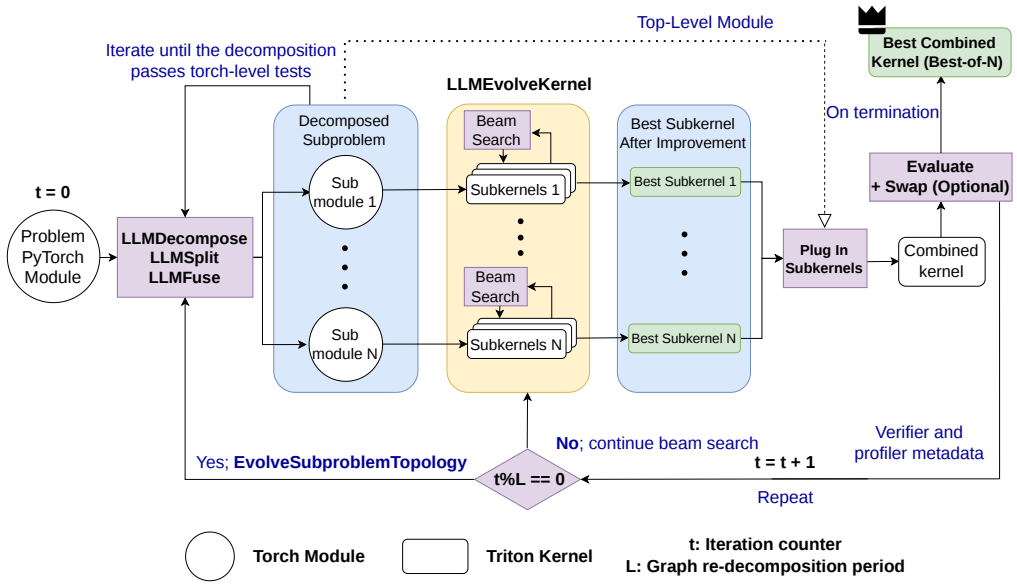

Figure 2. MKEvolve Algorithm 1 Visualization.

problem decomposition itself, resulting in more efficient kernel implementations. Finally, MKEVOLVE relies on a programmatic evaluation pipeline with cheating detection and strict correctness criteria, instead of LLM-generated tests, improving robustness and reliability.

3. Methodology

Algorithm 1 outlines the core MKEVOLVE workflow. MKEVOLVE produces a Python *codebase* that implements the given PyTorch module in a modular, kernelized form. This codebase consists of a set of files $\mathcal{K}$, each implementing a kernel for a subproblem derived from the original module M, along with a single top-level module file S that serves as the entry point and orchestrates the subkernels in $\mathcal{K}$. A representative MKEVOLVE codebase is provided in Appendix G. We now describe each algorithm component in detail.

LLMDecompose agent decomposes the initial problem into subproblems $\mathcal{P}$ and a top-level module S. The agent is instructed to decompose the original problem into subproblems when fully fused kernels are difficult to implement correctly, while maximizing intra-subproblem fusion opportunities and delegating all mathematical computation to the subproblems. Its input is a Python file containing the problem PyTorch module along with input generation functions for module's `__init__` and `forward` methods, and its output is a Python codebase comprised of subproblem PyTorch module files and a top-level module file that chain them into a functionally identical implementation of the original PyTorch module. During execution, the agent launches workers that attempt decomposition in parallel via sequential scaling, and returns as soon as any worker produces a solution that passes programmatic checks using PyTorch hooks, including but not limited to:

- Functional correctness: Verifying that the original module and the decomposed codebase produce identical outputs on the same input.
- `forward` shape coverage: Ensuring each subproblem's `forward` input generator covers all input shapes used by the top-level module.
- `__init__` input coverage: Ensuring each subproblem's `__init__` input generator covers all `__init__` initialization inputs used by the top-level module.

These checks ensure that the generated subproblems fully and exclusively cover the initialization and forward inputs required by the generated top-level module. Otherwise, a fallback top-level module is returned that wraps the original module as the sole submodule.

LLMEvolveKernel agent generates, debugs, and optimizes Triton kernels for the given (sub)problem for b inner loop iterations. We use beam search, where the reward is kernel speedup vs *torch.compile*, and incorrect kernels have a speedup of 0. LLMEvolveKernel agent's verifier comprises of TritonRL-based cheating detector (Woo et al., 2026) and the KernelBench repository's CUDA-stream evaluation pipeline.

Algorithm 1 MKEvolve

Require: PyTorch module M, Verifier $\mathcal{V}$, Iterations T, Refinement period L, Threshold τ
Ensure: Best subkernels $\mathcal{K}^*$, Best top-level module S^*, Best speedup r^*

```
P, S ← LLMDecompose(M)                                  ▷ Initial subproblems and top-level module
K ← [K_i]_{i=1}^{|P|},   K_i = (∅, 0)                   ▷ Initial subkernels and scores list
for t = 1 ... T do
    P, K, S ← EvolveSubproblemTopology(P, K, t, L)      ▷ See Algorithm 2
    [b_1, ..., b_{|P|}] ← AllocateLLMBudget(K)          ▷ Prioritize based on runtime/error
    for i = 1 ... |P| do
        K_i ← LLMEvolveKernel(P_i, K_i, b_i, V)         ▷ Local subkernel improvement
    end for
    r ← Evaluate(S, K, V)                               ▷ End-to-end kernel verification and profiling
    if r ≥ r* then
        K*, S*, r* ← K, S, r
    end if
end for
if swapping is enabled then
    K* ← Swap(K*, τ)                                    ▷ Substitute poor subkernels with alternatives
end if
return K*, S*
```

Algorithm 2 EvolveSubproblemTopology $(\mathcal{P}, \mathcal{K}, t, L)$

```
if t > 0 and t mod L = 0 then   ▷ Periodic refinement
    if there are failing subkernels K_fail ⊂ K then
        P, S ← LLMSplit(P, K_fail)
    else if fusion is enabled then
        P, S ← LLMFuse(P, K)
    end if
end if
if local and global correctness checks do not agree then
    P, S ← TotalFuse(P)          ▷ Monolithic problem
end if
K ← FilterKernels(P, K)   ▷ For deleted subproblems
K ← InitEmptyKernels(P, K) ▷ For new subproblems
return P, K, S
```

LLMSplit agent attempts to split a list of provided subproblems into exactly 2 PyTorch subproblems each, and restructure the top-level module to account for this change. If this fails, the previous subproblem and top-level module are returned.

LLMFuse agent attempts to fuse any number of subproblems PyTorch modules into larger modules at the LLM's discretion, and restructure the top-level module to account for this change. If this fails, the previous subproblem and top-level module are returned. We denote MKEVOLVE with LLMFuse enabled as MKEVOLVE (FUSE).

AllocateLLMBudget allocates the iteration's LLM call budget assigned to LLMEvolveKernel agents tasked with solving different subproblem kernels. Specifically, if there are any subproblem(s) that are new or has incorrect Triton kernel implementation, it evenly distributes the LLM call budget between them. Otherwise, it allocates the LLM call budget to each subproblem agent proportionally to the subproblem kernel's runtime observed during the last end-to-end evaluation.

Evaluate verifies and profiles the end-to-end kernel implemented by the top-level module against the original problem code, with the verifier used by LLMEvolveKernel. The output reward is speedup against *torch.compile*, with incorrect kernels receiving 0 reward.

Swap iterates through all subkernels and programmatically replaces subkernels whose speedup relative to *torch.compile* is below the swap threshold τ with alternative implementations (ex. PyTorch, subkernels from a more powerful LLM). Swap is not triggered if there are fewer than two subkernels or if all subkernels are slower than *torch.compile*. We denote MKEVOLVE with Swap enabled as MKEVOLVE (SWAP).

Additional Remarks MKEVOLVE produces interpretable and modular kernels that enable engineers to quickly localize which subproblem is responsible for end-to-end errors (see Figure 1), rather than relying solely on aggregate failure messages that only report overall numerical deviation of the kernel output from the ground truth reference implementation. MKEVOLVE also enables flexible selection among subproblem kernels and direct observation of each subker-

(a) Claude 4.5 Opus metrics on 100 KernelBench level 2 problems.

METHOD	Correct (↑)	$\text{Fast}_{0.5}$ (↑)	Fast_1 (↑)	Fast_2 (↑)	# Tokens
Parallel Scaling	0.68	0.46	0.03	0.01	1.3×10^6
Beam Search	0.96	0.72	0.36	0.07	2.0×10^6
MKEvolve	**0.99**	**0.77**	0.49	**0.09**	1.7×10^6
MKEvolve (Fuse)	0.98	0.76	**0.54**	**0.09**	1.9×10^6
KernelFalcon	0.93	0.22	0.00	0.00	0.2×10^6
MKEvolve (PostFuse)	0.99	0.80	0.55	**0.10**	1.8×10^6
MKEvolve (Swap)	**1.00**	**0.87**	**0.58**	0.09	1.7×10^6

(b) Claude 4.5 Opus metrics on 50 KernelBench level 3 problems.

METHOD	Correct (↑)	$\text{Fast}_{0.5}$ (↑)	Fast_1 (↑)	Fast_2 (↑)	# Tokens
Parallel Scaling	0.72	0.32	0.14	0.06	1.8×10^6
Beam Search	0.88	0.64	0.26	**0.10**	3.0×10^6
MKEvolve	**0.94**	**0.74**	**0.34**	0.08	2.1×10^6
MKEvolve (Fuse)	0.92	0.68	0.32	0.08	2.9×10^6
KernelFalcon	0.76	0.14	0.04	0.00	1.4×10^6
MKEvolve (PostFuse)	**0.94**	0.74	0.34	0.08	2.4×10^6
MKEvolve (Swap)	**0.94**	**0.80**	**0.60**	0.06	2.1×10^6

Table 1. Claude 4.5 Opus experiment results on baseline correctness and speedup. For each metric, the best value among the first four baselines synthesizing 160 kernels is shown in bold. The best value across all methods is shown in bold and underlined.

nel's impact on overall speedup (see Figure 1). Notably, even without invoking the LLMFuse agent, MKEVOLVE supports kernel fusion–based optimizations at the subproblem level. Finally, under certain assumptions, the global model error can be bounded by the local errors of individual subkernels, ensuring correctness of both the overall kernel and its components (see Section E).

4. Experiments

Setup We assess the correctness and speedup of LLM-synthesized FP32 Triton kernels on 150 KernelBench (Ouyang et al., 2025) L2 and L3 benchmark problems. The KernelBench L2 benchmark comprises 100 PyTorch problems that include multiple primitive operator sequences (e.g., a combination of convolution, ReLU, and bias). The KernelBench L3 benchmark comprises 50 PyTorch problems that include full machine learning models (e.g., AlexNet, MiniGPT). All experiments are conducted on AWS P4d nodes with A100 GPUs and are conducted twice using Claude 4.5 Opus and GPT-OSS 120B as the base LLM.

We assess a baseline's performance using the following metrics: (1) the proportion of benchmark problems for which it generates a correct solution (Correct), (2) the proportion of which it produces a kernel that is at least p-times as fast as *torch.compile* (Fast_p where $p \in \{0.5, 1, 2\}$) (Ouyang et al., 2025), and (3) the proportion of which it produces a kernel that is faster than all other baselines. These metrics are computed after the baseline finishes and returns the fastest discovered kernel. All kernels are evaluated with the KernelBench repository's CUDA-stream evaluation pipeline. The pipeline measures correctness by running each kernel 5 times with a random input of same shape and comparing the outputs to the PyTorch output with atol/rtol of 10^{-4}. Speedup is measured against *torch.compile* using 10 warmup and 100 profiling runs, with correctness check failure defined as 0 speedup. We also augment our evaluation pipeline with a TritonRL-based (Woo et al., 2026) cheating detector and impose a 5-minute subprocess-level time limit.

We benchmark Parallel Scaling, Beam Search, MKEvolve, and MKEvolve (Fuse) that generate 160 kernels for each problem with the same set of prompts and evaluators. This isolates the effects of subkernel optimization vs end-to-end kernel optimization on the solution correctness and speedup. We also benchmark additional baselines that generate a variable number of kernels: KernelFalcon (Wang et al., 2025) [2] modified to generate FP32 kernels that are evaluated with our pipeline, MKEvolve (PostFuse) that attempts to fuse MKEvolve's solution into an end-to-end kernel 24 times, and MKEvolve (Swap), which programmatically replaces MKEvolve's solution subkernels with less than 0.9x speedup of subproblem *torch.compile* code with PyTorch implementations. Please refer to Appendix A for additional details.

Claude 4.5 Opus Results Table 1 reports correctness and speedup metrics for each baseline, along with the number of LLM tokens used during program execution. We observe that MKEvolve outperforms Parallel Scaling and Beam Search on all but one metric across both KernelBench L2 and L3 benchmarks. This result is achieved while us-

[2] We baseline the March 3rd, 2026 version of KernelFalcon.

(a) GPT-OSS 120B metrics on 100 KernelBench level 2 problems.

METHOD	Correct (↑)	$\text{Fast}_{0.5}$ (↑)	Fast_1 (↑)	Fast_2 (↑)	# Tokens
Parallel Scaling	0.59	0.32	0.13	0.05	1.6×10^6
Beam Search	0.84	0.48	0.23	**0.16**	2.5×10^6
MKEvolve	0.94	**0.54**	0.22	0.05	1.9×10^6
MKEvolve (Fuse)	**0.95**	0.53	**0.24**	0.15	2.2×10^6
KernelFalcon	0.83	0.25	0.01	0.00	0.3×10^6
MKEvolve (PostFuse)	0.97	0.54	0.27	0.08	2.3×10^6
MKEvolve (Swap)	**0.99**	**0.77**	**0.35**	0.05	1.9×10^6

(b) GPT-OSS 120B metrics on 50 KernelBench level 3 problems.

METHOD	Correct (↑)	$\text{Fast}_{0.5}$ (↑)	Fast_1 (↑)	Fast_2 (↑)	# Tokens
Parallel Scaling	0.26	0.12	0.02	0.00	2.0×10^6
Beam Search	0.52	0.14	0.04	**0.02**	3.8×10^6
MKEvolve	0.70	0.18	0.06	**0.02**	2.4×10^6
MKEvolve (Fuse)	**0.72**	**0.32**	**0.12**	0.00	2.7×10^6
KernelFalcon	0.38	0.10	0.02	0.00	1.0×10^6
MKEvolve (PostFuse)	0.72	0.20	0.10	**0.02**	3.0×10^6
MKEvolve (Swap)	**0.88**	**0.58**	**0.36**	0.02	2.4×10^6

Table 2. GPT-OSS 120B experiment results on baseline correctness and speedup. For each metric, the best value among the first four baselines synthesizing 160 kernels is shown in bold. The best value across all methods is shown in bold and underlined.

ing 15 to 30% less tokens than beam search in spite of MKEvolve employing beam search under the hood. We attribute this token efficiency to MKEvolve refining subkernels rather than optimizing end-to-end kernels, as done by the other baselines. MKEvolve (Fuse) performs comparably to MKEvolve for the most part, though its attempt at fusing across subproblems and working with larger subproblems brings its token usage closer to Beam Search. We note that MKEvolve is capable of producing fully fused kernels at the subproblem level.

When expanding the comparison to methods that synthesize a variable number of kernels (KernelFalcon) or build on MKEvolve's solutions, we find that MKEvolve (Swap) dominates most metrics. While this success originates from selectively swapping subproblem solutions with PyTorch implementations, we note that the swapping behavior is entirely controllable and can be useful in practice when writing kernels for complex models, for which it is easy to design performant kernels for some parts but not others. We find that KernelFalcon's emphasis on synthesizing correct kernels, combined with its use of an LLM-determined correctness threshold and the absence of cheating detectors in parts of the codebase, could result in suboptimal performance. The synthesized kernels were generally slow and sometimes failed our evaluation pipeline's correctness checks from exceeding the evaluation timeout, failing stricter correctness thresholds, or being flagged by the cheating detector.

Our findings are further supported by Figure 4, which shows that MKEvolve and its variants produce faster kernels more frequently than Beam Search's kernels across benchmarks. Overall, MKEvolve, through its modular subkernel synthesis strategy, outperforms the other baselines while using fewer LLM tokens on both the simpler operator-sequence tasks in KernelBench L2 and the more complex full-model tasks in KernelBench L3.

GPT-OSS 120B Results Table 2 reports correctness and speedup metrics for each baseline, along with the number of LLM tokens used during program execution. We observe a similar trend to that of the Claude 4.5 Opus results, where MKEvolve generally outperforms Beam Search on correctness and performance while using 25 to 35% fewer tokens. While the speedup metrics are closer compared to the Claude 4.5 Opus results, Figure 4 suggests that MKEvolve still holds an edge over Beam Search while being much more token efficient from solving simpler subproblems when synthesizing new kernels. MKEvolve (Fuse) performs marginally better than MKEvolve at the expense of higher token usage. MKEvolve (Swap) again performs best out of all baselines. Overall, we find that the benefit of MKEvolve's modular kernel synthesis strategy can hold for both frontier and open-source base LLMs.

Further Analysis Figure 5 and Figure 6 in Appendix B depict the baselines' metric improvements over time as they generate 160 kernels versus tokens usage. We observe that MKEvolve achieves the highest metrics across different token usage levels for most metrics across benchmarks. We also ablate MKEvolve to perform a single problem decom-

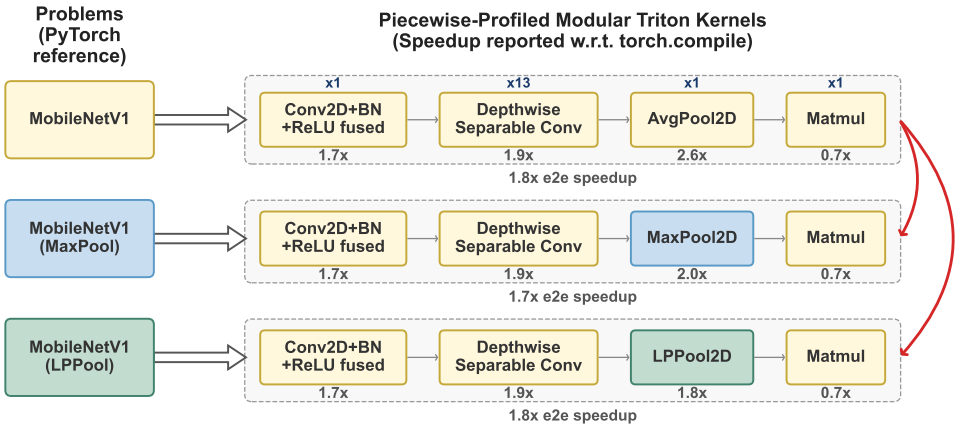

Figure 3. Transferability of MKEvolve-generated Triton kernels across MobileNetV1 variants. **Top**: The base kernel invokes four subkernels sequentially, achieving 1.8× speedup over torch.compile. **Middle & Bottom**: Replacing AvgPool2D with MaxPool2D or LPPool2D yields adapted kernels achieving 1.7× and 1.8× speedups, requiring only 4 LLM calls to produce the new pooling subkernels.

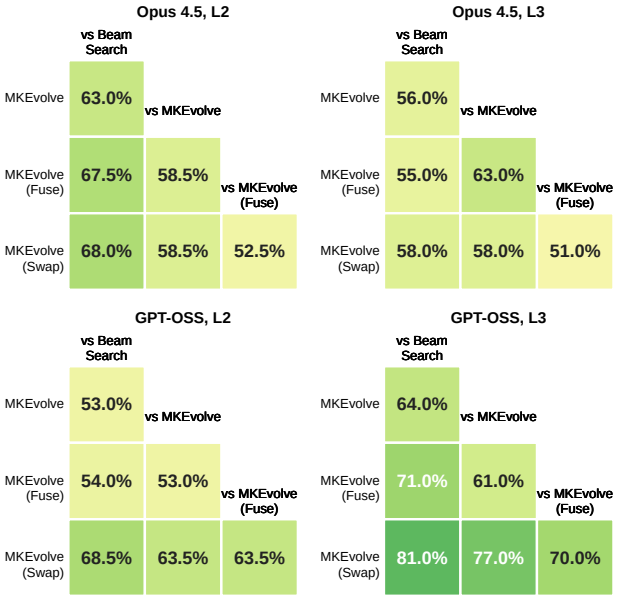

Figure 4. Win-rate (proportion of tasks for which the row method achieved higher speedup than the column method) heatmaps comparing various MKEvolve variants and beam search, using both Claude 4.5 Opus and gpt-oss as backbone LLM to solve L2 and L3 tasks sampled from KernelBench.

position in Appendix F, and find that this slightly hurts the correctness metric compared to multiple subproblem splits.

We also rerun our main experiment for Beam Search and MKEvolve for another seed in Appendix C to assess the effect of random seed on our results. We observe that MKEvolve generally outperforms Beam Search while using substantially fewer tokens, and on the Claude 4.5 Opus experiments, outperforms Beam Search on every metric and benchmark.

We emphasize that MKEvolve's ability to operate on a decomposed problem representation enables it to generate kernels that are readily adaptable to related tasks, a desirable property in machine learning where practitioners often work with models that differ only slightly. Figure 3 showcases this for the KernelBench L3 MobileNetV1 problem. By replacing the MobileNetV1 solution's AvgPool2D subkernel with MaxPool2D and LPPool2D subkernels synthesized after just 4 LLM invocations, MKEvolve can rapidly produce kernels that obtain 1.7x and 1.8x speedup over *torch.compile* for the MobileNetV1 architectural variants.

Lastly, while we employed atol/rtol of 10^{-4} for the FP32 kernel correctness verification to be consistent with the official KernelBench evaluation, we empirically show how modular kernel generation with local subkernel verification paves the way for a more robust correctness check mechanism capable of detecting common correctness errors in Appendix D.

5. Conclusion

We introduced MKEVOLVE, a modular kernel code generation framework that iteratively decomposes complex PyTorch modules into subproblems, independently solves them, and composes the resulting subkernels into a complete, end-to-end solution. We demonstrated that MKEvolve improves both kernel correctness and performance speedup, while reducing LLM token usage by up to 35% compared to a beam search baseline that operates directly on end-to-end kernels. We conclude by highlighting several promising directions for future work, including adaptive correctness threshold selection for subproblems, investigating more advanced subkernel optimization strategies such as Monte Carlo Tree Search, and extending MKEVOLVE to additional kernel programming frameworks and settings.

Impact Statement

This work has the potential to accelerate AI training and inference on GPUs and NPUs. Such improvements could enable more efficient AI systems, including systems that process and respond to user queries more quickly. At the same time, the methods developed in this work could also be used in applications with negative societal consequences, such as more efficient surveillance systems. As with many advances in AI infrastructure, the broader impact depends on how these methods are deployed.

References

Baronio, C., Marsella, P., Pan, B., Guo, S., and Alberti, S. Kevin: Multi-turn rl for generating cuda kernels, 2025. URL https://arxiv.org/abs/2507.11948.

Beuzeville, T., Buttari, A., Gratton, S., and Mary, T. Deterministic and probabilistic rounding error analysis of neural networks in floating-point arithmetic. *IMA Journal of Numerical Analysis*, 2025. URL https://hal.science/hal-04663142.

Castin, V., Ablin, P., and Peyré, G. How smooth is attention?, 2024. URL https://arxiv.org/abs/2312.14820.

Chen, T., Ye, Z., Xu, B., Ye, Z., Liu, T., Hassani, A., Chen, T., Kerr, A., Wu, H., Xu, Y., Chen, Y.-J., Chen, H., Kane, A., Krashinsky, R., Liu, M.-Y., Grover, V., Ceze, L., Bringmann, R., Tran, J., Liu, W., Xie, F., Lightstone, M., and Shi, H. Avo: Agentic variation operators for autonomous evolutionary search, 2026. URL https://arxiv.org/abs/2603.24517.

Dai, W., Wu, H., Yu, Q., ang Gao, H., Li, J., Jiang, C., Lou, W., Song, Y., Yu, H., Chen, J., Ma, W.-Y., Zhang, Y.-Q., Liu, J., Wang, M., Liu, X., and Zhou, H. Cuda agent: Large-scale agentic rl for high-performance cuda kernel generation, 2026. URL https://arxiv.org/abs/2602.24286.

Fisches, Z. V., Paliskara, S., Guo, S., Zhang, A., Spisak, J., Cummins, C., Leather, H., Synnaeve, G., Isaacson, J., Markosyan, A., and Saroufim, M. KernelLLM: Making Kernel Development More Accessible, 6 2025. URL https://huggingface.co/facebook/KernelLLM.

Higham, N. J. and Mary, T. Sharper probabilistic backward error analysis for basic linear algebra kernels with random data. *SIAM Journal on Scientific Computing*, 42(5): A3427–A3446, 2020. doi: 10.1137/20M1314355. URL https://doi.org/10.1137/20M1314355.

Hong, C., Bhatia, S., Cheung, A., and Shao, Y. S. Autocomp: A powerful and portable code optimizer for tensor accelerators, 2025. URL https://arxiv.org/abs/2505.18574.

Kim, H., Papamakarios, G., and Mnih, A. The lipschitz constant of self-attention, 2021. URL https://arxiv.org/abs/2006.04710.

Kong, L., Wei, J., Shen, H., and Wang, H. Concur: Conciseness makes state-of-the-art kernel generation, 2025. URL https://arxiv.org/abs/2510.07356.

Lange, R. T., Sun, Q., Prasad, A., Faldor, M., Tang, Y., and Ha, D. Towards robust agentic cuda kernel benchmarking, verification, and optimization, 2025. URL https://arxiv.org/abs/2509.14279.

Li, S., Wang, Z., He, Y., Li, Y., Shi, Q., Li, J., Hu, Y., Che, W., Han, X., Liu, Z., and Sun, M. Autotriton: Automatic triton programming with reinforcement learning in llms, 2025. URL https://arxiv.org/abs/2507.05687.

Liao, G., Qin, H., Wang, Y., Golden, A., Kuchnik, M., Yetim, Y., Ang, J. J., Fu, C., He, Y., Hsia, S., Jiang, Z., Li, D., Pashkevich, U., Puvvada, V., Shi, F., Steiner, M., Xiao, R., Yan, N., Yu, X., Fang, Z., Levenstein, R., Ho, K., Zhu, H., Hammond, A., Li, R., Mathews, A., Gondkar, K., Zainul-Abedin, A., Singh, K., Yu, H., Chi, W., Huang, B., Zhang, S., Weller, N., Marine, Z., Cook, W., Wu, C.-J., and Liu, G. Kernelevolve: Scaling agentic kernel coding for heterogeneous ai accelerators at meta, 2026. URL https://arxiv.org/abs/2512.23236.

Novikov, A., Vũ, N., Eisenberger, M., Dupont, E., Huang, P.-S., Wagner, A. Z., Shirobokov, S., Kozlovskii, B., Ruiz, F. J. R., Mehrabian, A., Kumar, M. P., See, A., Chaudhuri, S., Holland, G., Davies, A., Nowozin, S., Kohli, P., and Balog, M. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

Ouyang, A., Guo, S., Arora, S., Zhang, A. L., Hu, W., Re, C., and Mirhoseini, A. Kernelbench: Can LLMs write efficient GPU kernels? In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=yeoN1iQT1x.

Qi, X., Wang, J., and Zhang, L. Understanding Optimization of Deep Learning via Jacobian Matrix and Lipschitz Constant, November 2023. URL http://arxiv.org/abs/2306.09338. arXiv:2306.09338 [cs].

Wang, L., Chen, S., Maher, B., Isaacson, J., Fang, L., Chi, W., Liu, J., Hammond, A., Fisches, Z., Saroufim, M., Hunt, W., Li, R., Kahn, J., El-Haraty, E., and Mathews,

A. KernelFalcon: Autonomous GPU Kernel Generation via Deep Agents. PyTorch Blog, 2025. URL https://pytorch.org/blog/kernelfalcon-autonomous-gpu-kernel-generation-via-deep-agents/.

Wei, A., Sun, T., Seenichamy, Y., Song, H., Ouyang, A., Mirhoseini, A., Wang, K., and Aiken, A. Astra: A multi-agent system for gpu kernel performance optimization, 2025. URL https://arxiv.org/abs/2509.07506.

Wen, Z., Zhang, Y., Li, Z., Liu, Z., Xie, L., and Zhang, T. MultiKernelBench: A Multi-Platform Benchmark for Kernel Generation, 2025. URL https://arxiv.org/abs/2507.17773.

Woo, J., Zhu, S., Nie, A., Jia, Z., Wang, Y., and Park, Y. Tritonrl: Training llms to think and code triton without cheating, 2026. URL https://arxiv.org/abs/2510.17891.

Zhang, G., Zhu, S., Wei, A., Song, Z., Nie, A., Jia, Z., Vijaykumar, N., Wang, Y., and Olukotun, K. Accelopt: A self-improving llm agentic system for ai accelerator kernel optimization, 2025. URL https://arxiv.org/abs/2511.15915.

A. Additional Experiment Details

Shared Setup All methods share the initial kernel generation prompt. Beam Search and MKEvolve variants use a beam width of 4, a beam expansion size of 1, and share the kernel debugging and refinement prompts. MKEvolve updates its subgraph structure once every 2 iterations, runs for 5 outer loops with a per-iteration LLM call budget of 32, and does not use fusion. We disable TF32 for both PyTorch and Triton as it affected correctness results for the default KernelBench threshold and ensure that the model tensor is contiguous. For parallel scaling and beam search, only the solutions that pass the cheating detector one more time at the end are returned. For MKEvolve, only subkernels that pass the cheating detector one more time after subkernel beam search are accepted. Claude 4.5 Opus experiments employed Claude 4.5 Sonnet as the cheating detector model, whereas GPT-OSS 120B experiment employed GPT-OSS 120B as the cheating detector model. We note that the typical wall clock time was around 2-4 hours per job.

KernelFalcon Setup We baseline the March 3rd, 2026 version of the official KernelFalcon repository. To match our evaluation setup, we changed KernelFalcon's self-test generation prompt to test FP32 kernels from FP16 kernels and set the default torch.allclose threshold default targets to 1e-4 atol/rtol for all problems (we note that KernelFalcon LLM-generated self-test allows the LLM to change the atol/rtol threshold). KernelFalcon was informed that it is to design kernels that will run on A100 GPU and that tl.dot's allow_tf32 flag can be set to False if there are persistent correctness errors. We evaluated KernelFalcon with the same code that we used to evaluate all other approaches. Because KernelFalcon's LLM-generated self-test outputs code is incompatible with KernelBench's evaluation script, we use Claude 4.5 Opus to convert the KernelFalcon solution wrapper to KernelBench evaluation-compatible format. Specifically, we attempted conversion 5 times per kernel and considered the task as a success if any of the 5 conversion attempts succeeded. We note that we further robustified the KernelFalcon repository to improve its reliability. For example, we added retry logic to the subproblem structure JSON file creation that fixed KernelFalcon from aborting the subproblem decomposition attempt and attempting end-to-end sequential refinements. In addition, we ensured that only a single subproblem kernel refinement happens at a time, since in some instances, KernelFalcon's AutoAgent attempted to launch 47 workers in parallel that attempt to evaluate 47 different subproblems on a single GPU, which led to GPU memory errors. KernelFalcon was run with sequential scaling `--max-iters=10` and `--compose-max-iters=10`, which corresponds to a maximum of 40 refinement iterations for every subproblem kernel synthesis, 10 refinement iterations for LLM that combines the subkernels by including them in the prompt, and 40 refinement iterations for the backup sequential scaling agent that tries to build an end-to-end kernel from scratch. KernelFalcon received the same verifier and profiler feedback from our evaluation pipeline as all other baselines.

B. Timeline Plots

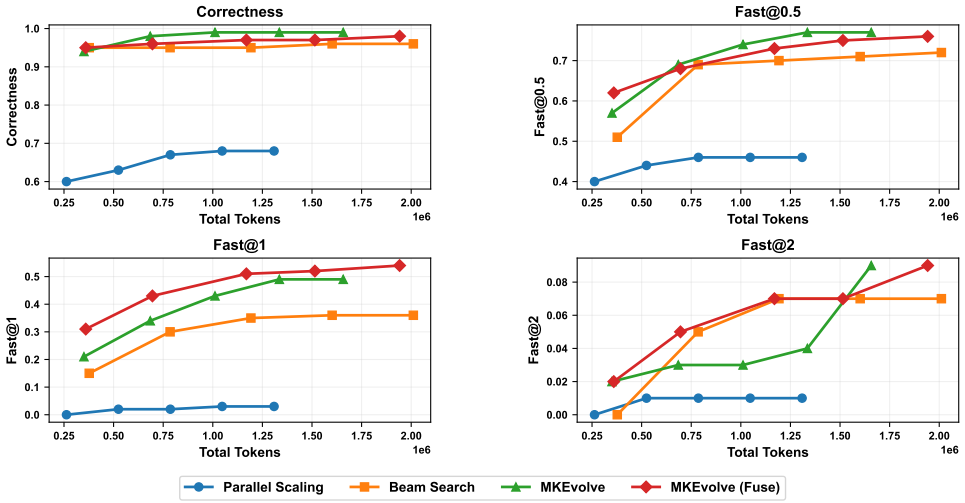

Figure 5. Claude 4.5 Opus KernelBench L2 experiment improvement over time plot for all baselines. We note that MKEvolve and its variants achieve the best final performance on all metrics while consuming fewer tokens per outer iteration.

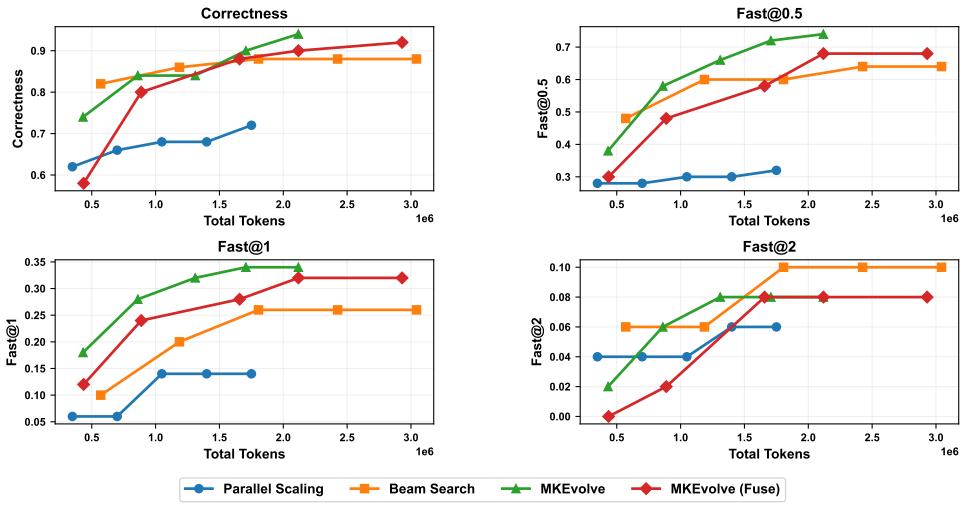

Figure 6. Claude 4.5 Opus KernelBench L3 experiment improvement over time plot for all baselines. We note that MKEvolve and its variants achieve the best final performance on all but one metric while consuming fewer tokens per outer iteration.

C. Example Second Seed Results

METHOD	Correct (↑)	$\text{Fast}_{0.5}$ (↑)	Fast_1 (↑)	Fast_2 (↑)	# Tokens
Beam Search	0.95	0.66	0.37	0.03	2.0×10^6
MKEvolve	**0.99**	**0.75**	**0.49**	**0.04**	1.7×10^6
MKEvolve (Swap)	1.00	0.87	0.61	0.04	1.7×10^6

Table 3. Claude 4.5 Opus Metrics on 100 KernelBench level 2 problems (Second Seed)

METHOD	Correct (↑)	$\text{Fast}_{0.5}$ (↑)	Fast_1 (↑)	Fast_2 (↑)	# Tokens
Beam Search	0.88	0.64	0.26	0.06	3.0×10^6
MKEvolve	**0.94**	**0.70**	**0.42**	**0.08**	2.1×10^6
MKEvolve (Swap)	0.96	0.78	0.58	0.04	2.1×10^6

Table 4. Claude 4.5 Opus Metrics on 50 KernelBench level 3 problems (Second Seed)

METHOD	Correct (↑)	$\text{Fast}_{0.5}$ (↑)	Fast_1 (↑)	Fast_2 (↑)	# Tokens
Beam Search	0.87	0.47	**0.29**	**0.17**	2.5×10^6
MKEvolve	**0.97**	**0.52**	0.21	0.09	1.9×10^6
MKEvolve (Swap)	0.99	0.76	0.27	0.08	1.9×10^6

Table 5. GPT-OSS 120B Metrics on 100 KernelBench level 2 problems (Second Seed)

METHOD	Correct (↑)	$\text{Fast}_{0.5}$ (↑)	Fast_1 (↑)	Fast_2 (↑)	# Tokens
Beam Search	0.56	**0.28**	0.06	**0.02**	3.7×10^6
MKEvolve	**0.68**	0.24	**0.10**	**0.02**	2.4×10^6
MKEvolve (Swap)	0.84	0.66	0.38	0.00	2.4×10^6

Table 6. GPT-OSS 120B Metrics on 50 KernelBench level 3 problems (Second Seed)

Table 7. Maximum absolute error (GPU vs. FP32 CPU reference) across operations. Errors collected using 30 runs with random $\mathcal{N}(0, 1)$ inputs.

	GPU FP32		GPU BF16	
Operation	**Max Abs**	**Mean Abs**	**Max Abs**	**Mean Abs**
Matmul 512^2	5×10^{-5}–7×10^{-5}	$\sim 6 \times 10^{-6}$	3×10^{-1}–5×10^{-1}	$\sim 1 \times 10^{-5}$
Softmax 128×1024	4×10^{-9}–7×10^{-9}	$\sim 6 \times 10^{-11}$	0–3×10^{-5}	$\sim 2 \times 10^{-10}$
LayerNorm	$\sim 1 \times 10^{-6}$	$\sim 4 \times 10^{-8}$	8×10^{-3}–2×10^{-2}	$\sim 5 \times 10^{-8}$
CrossEntropy 128×1000	$\sim 1 \times 10^{-6}$	$\sim 5 \times 10^{-8}$	3×10^{-2}–3×10^{-2}	$\sim 1 \times 10^{-2}$
Conv2d	8×10^{-4}–1×10^{-3}	$\sim 1 \times 10^{-4}$	1×10^{-2}–2×10^{-2}	$\sim 9 \times 10^{-4}$

Table 8. Input-dependent error for matmul 1024×1024 (max absolute error, 30 runs).

Input Distribution	**FP32 Max Abs**	**BF16 Max Abs**
$\mathcal{N}(0, 0.01^2)$	2.0×10^{-8}–2.6×10^{-8}	3.1×10^{-5}–6.1×10^{-5}
$\mathcal{N}(0, 1)$	2.0×10^{-4}–2.7×10^{-4}	4.7×10^{-1}–5.0×10^{-1}
$\mathcal{N}(0, 10^2)$	1.9×10^{-2}–2.8×10^{-2}	3.2×10^{1}–6.1×10^{1}
Uniform$(0, 1)$	5.2×10^{-4}–6.7×10^{-4}	~ 1.0
Uniform$(-100, 100)$	6.3×10^{-1}–8.8×10^{-1}	1.0×10^{3}–2.0×10^{3}
Sparse (5%)	1.4×10^{-6}–1.9×10^{-6}	3.1×10^{-2}–6.0×10^{-2}

D. Modular Kernel Generation Enables Better Correctness Checking

KernelBench's approach for verifying kernel correctness is to compare output tensors against CPU references using `torch.allclose(output, reference, atol, rtol)` and pre-determined thresholds. This function checks whether $|\text{output} - \text{reference}| \leq \texttt{atol} + \texttt{rtol} \times |\text{reference}|$ holds element-wise. While the criteria itself seems to be simple, we argue that choosing appropriate tolerance values is fundamentally difficult in practice, especially for lower-precision beyond FP32. The tolerance needs to depend on various factors like datatype, operator, operand size, input data distribution, etc. For more complex kernels that involve multiple operators, this difficulty is further amplified. Through this section, we argue that our decomposition technique makes the correctness checking problem more tractable to ensure the correctness of LLM-generated kernel implementations in practice.

First, we demonstrate that the floating-point errors inherent to operators can make it complicated to determine the right threshold, even when considering operators in isolation. Table 7 shows the maximum absolute error between correct GPU and FP32 CPU implementations: in FP32, errors range from $\sim 4 \times 10^{-9}$ (softmax) to $\sim 1 \times 10^{-3}$ (conv2d); In BF16 from $\sim 3 \times 10^{-5}$ (softmax) to $\sim 5 \times 10^{-1}$ (matmul). Table 8 shows that the same matmul produces errors spanning many orders of magnitude depending on input scale. Table 9 shows error also grows with matrix dimension. These observations offer some intuition for why checking correctness of complex kernels using the resulting tensors can be difficult: the inherent "error profile" for different operators can differ dramatically, and can even depend on the output of preceding operators; also numerical deviations caused by incorrect components may be "partially absorbed" by later components, due to their differences in sensitivity to numerical perturbations.

We now show that decomposing composed operators into sub-operations makes per-operator tolerance selection more reliable through case studies. We use two realistic cases: (1) MinGPT Causal Attention (KernelBench `L3#43`: QKV projection → reshape → scaled dot-product → causal mask → softmax → attention → output projection), and (2) a Transformer FFN block (LayerNorm → Linear → GELU → Linear → residual). We inject bugs that reflect common mistakes when writing Triton kernels, which include the following attention bugs:

- **Unsafe softmax:** `exp(x)` without subtracting the max—overflows for large scores.
- **Causal mask off-by-one:** diagonal is masked (`tril(diagonal=-1)` vs `tril(diagonal=0)`).
- **Tile boundary (K):** last partial tile of K's head dimension is zeroed (unloaded), simulating missing boundary handling when dimensions don't divide evenly by `BLOCK_SIZE`.
- **Wrong accumulator dtype:** $QK^\top$ matmul accumulates in BF16 instead of FP32.

Table 9. Error growth with matrix dimension for matmul ($\mathcal{N}(0,1)$ inputs, 30 runs). BF16 errors are $\sim$1000$\times$ larger than FP32 and both grow with dimension.

	FP32		BF16	
N	**Avg Max Abs**	**Avg Mean Abs**	**Avg Max Abs**	**Avg Mean Abs**
64	0	0	6.9×10^{-2}	9.0×10^{-3}
128	0	0	1.2×10^{-1}	1.3×10^{-2}
256	4.6×10^{-5}	3.6×10^{-6}	2.0×10^{-1}	1.8×10^{-2}
512	6.1×10^{-5}	5.8×10^{-6}	2.5×10^{-1}	2.5×10^{-2}
1024	2.1×10^{-4}	1.3×10^{-5}	4.9×10^{-1}	3.6×10^{-2}
2048	5.2×10^{-4}	2.7×10^{-5}	5.0×10^{-1}	5.1×10^{-2}
4096	1.0×10^{-3}	5.4×10^{-5}	1.0	7.2×10^{-2}

Table 10. BF16 threshold dilemma for **Causal Attention**: false negative rate (correct outputs rejected) and bug catch rate at three fixed E2E thresholds (out of 20 eval runs).

	`atol`=10^{-3}	`atol`=10^{-2}	`atol`=10^{-1}
Correct rejected	**20/20**	0/20	0/20
Unsafe softmax	20/20	**0/20**	**0/20**
Mask off-by-one	20/20	20/20	20/20
Tile boundary (K)	20/20	20/20	20/20
Wrong acc dtype (QK)	20/20	**0/20**	**0/20**
Softmax dim=-2	20/20	20/20	20/20

- **Softmax wrong dimension:** softmax computed over `dim=-2` instead of `dim=-1`.

and FFN bugs:

- **Unsafe LayerNorm:** divides by σ without subtracting μ first.
- **GELU sigmoid approximation:** uses $x \cdot \sigma(1.702x)$ instead of the exact GELU.
- **Tile boundary (fc1):** last partial block of `fc1.weight` is zeroed (unloaded).
- **Wrong accumulator dtype (fc2):** fc2 matmul accumulates in BF16 instead of FP32.
- **Missing residual connection:** output $=$ fc2(GELU(fc1(LN(x)))) without adding x.
- **ReLU instead of GELU:** wrong activation function.

We compare two verification strategies. **End-to-end (E2E):** check only the final output using a fixed tolerance, as in KernelBench. **Decomposed:** check each intermediate tensor using a calibrated per-operator tolerance (max $+ 3\sigma$ of the correct error observed over 20 calibration runs, with `rtol`=0); a bug is caught if any stage fails. Tables 10 and 11 show the core problem with fixed thresholds for BF16 composed operators: every choice is either too tight (rejecting correct outputs) or too loose (missing bugs). For the attention block, `atol` $= 10^{-3}$ rejects 100% of correct outputs; `atol` $= 10^{-2}$ is usable but misses 2 of 5 bugs. For the FFN block, `atol` $= 10^{-2}$ rejects 100% of correct outputs, while `atol` $= 10^{-1}$—the only usable option—misses 2 of 6 bugs.

Tables 12 and 13 compare E2E at the best usable fixed threshold against the decomposed approach. In FP32, the decomposed approach matches or exceeds E2E on every bug, but the gap is very small. In BF16, the gap between the E2E and the decomposed approach becomes much more obvious. The decomposed method catches 4/5 attention bugs (vs. 3/5 for E2E and 5/6 FFN bugs (vs. 4/6 for E2E).

We empirically define the *feasibility gap* of an operator s to quantify the robustness of detection:

$$\text{gap}(s) = \min_{\text{trials}} \text{err}_{\text{buggy}}(s) \;-\; \max_{\text{trials}} \text{err}_{\text{correct}}(s)$$

Table 11. BF16 threshold dilemma for **FFN block**: false negative rate and bug catch rate at three fixed E2E thresholds (out of 20 eval runs).

	atol=10^{-3}	atol=10^{-2}	atol=10^{-1}
Correct rejected	**20/20**	**20/20**	0/20
Unsafe LayerNorm	20/20	20/20	**5/20**
GELU sigmoid approx	20/20	20/20	**0/20**
Tile boundary (fc1)	20/20	20/20	20/20
Wrong acc dtype (fc2)	20/20	20/20	**0/20**
Missing residual	20/20	20/20	20/20
ReLU not GELU	20/20	20/20	20/20

Table 12. Bug detection for MinGPT Causal Attention (20 eval runs). E2E uses the best usable fixed threshold (FP32: 10^{-5}; BF16: 10^{-2}). Decomposed uses max$+3\sigma$ per-stage tolerances with rtol=0. †Undetectable by both.

	FP32		**BF16**	
Bug	**E2E**	**Dec**	**E2E**	**Dec**
Unsafe softmax	$0^{\dagger}$	1	0	**12**
Mask off-by-one	20	20	20	20
Tile boundary (K)	20	20	20	20
Wrong acc dtype (QK)	20	20	$0^{\dagger}$	$0^{\dagger}$
Softmax dim=-2	20	20	20	20
Total	80/100	**81/100**	60/100	**72/100**
Correct rejected	0/20	0/20	0/20	0/20

A positive gap means a tolerance exists that perfectly separates correct from buggy at the output of operator s; a larger gap (bigger when both gaps are positive, or closer to zero when both gaps are negative) means the detection is more robust to input randomness and hardware nondeterminism. A negative gap means no tolerance can separate correct from buggy. Table 14 shows that for bugs caught by both methods, the decomposed feasibility gap is consistently wider. For particular bugs like GELU sigmoid approximation and unsafe softmax in BF16, the E2E gap is **negative**, while the gap is positive for the decomposed method. This illustrates that the decomposed method can enable detection of certain subtle bugs that are impossible to be caught using the E2E checking method.

In the current paper, we conduct all our correctness checking using the criteria of `atol=rtol=1e-4` for per-module checking and end-to-end result tensor checking, which agrees with what KernelBench provides, to ensure a fair comparison with existing techniques building on KernelBench. Our correctness analysis above also supports this decision, due to the similar observed effects of using fine-grained accuracy criteria versus end-to-end criteria for FP32 kernels in KernelBench. However, we point out that our modular kernel generation and accuracy checking framework paves the way for more robust accuracy checking when it comes to generating kernels that use lower-precision datatypes in practice.

E. Local Smoothness Implies Global Smoothness

There exist various work studying smoothness properties of neural network building blocks (Kim et al., 2021; Castin et al., 2024; Qi et al., 2023) as well as techniques for estimating floating- point round-off errors (Beuzeville et al., 2025; Higham & Mary, 2020). Following standard procedures of probabilistic backward error analysis (see e.g., Beuzeville et al. (2025)) on the L_∞ norm of tensors, it is possible to get a bound on the element-wise maximum difference for the output tensor of a sequence of composed operators, which is expressed in terms of the condition number and other properties of the individual operators and their floating-point implementations. We note that this observation offers a guarantee that the tolerance for the composed operators can be guaranteed given the tolerance for individual operators, while the converse is not necessarily true—we demonstrated several such cases where numerical deviation are "dampened" by subsequent operators in Appendix D. This offers some empirical evidence on why our compositional generation method is more robust in ensuring that all operators are implemented correctly.

Table 13. Bug detection for Transformer FFN block (20 eval runs). E2E uses fixed threshold (FP32: 10^{-5}; BF16: 10^{-1}, the only usable option). †Undetectable by both.

	FP32		BF16	
Bug	**E2E**	**Dec**	**E2E**	**Dec**
Unsafe LayerNorm	20	20	5	**20**
GELU sigmoid approx	20	20	0	**20**
Tile boundary (fc1)	20	20	20	20
Wrong acc dtype (fc2)	20	20	0†	0†
Missing residual	20	20	20	20
ReLU not GELU	20	20	20	20
Total	120/120	**120/120**	65/120	**100/120**
Correct rejected	0/20	1/20	0/20	0/20

Table 14. Feasibility gap: E2E (best usable fixed threshold) vs. decomposed (best stage). A positive gap means the bug is separable from correct outputs; negative means inseparable. "Ratio" = decomposed gap / E2E gap; ∞: E2E gap negative, decomposed gap positive.

	FP32			BF16		
Bug	**E2E**	**Dec (stage)**	**Ratio**	**E2E**	**Dec (stage)**	**Ratio**
Causal Attention						
Unsafe softmax	-9×10^{-7}	-1×10^{-7} (smax)	n/a	-6×10^{-4}	$+2 \times 10^{-4}$ (smax)	$\boldsymbol{\infty}$
Mask off-by-one	$+7.6 \times 10^{-1}$	$+1.5$ (attn)	$2\times$	$+7.7 \times 10^{-1}$	$+1.5$ (attn)	$2\times$
Tile boundary	$+1.1 \times 10^{-1}$	$+11$ (QK)	$\mathbf{99}\times$	$+1.1 \times 10^{-1}$	$+11$ (QK)	$\mathbf{101}\times$
Wrong acc dtype	$+5.6 \times 10^{-4}$	$+6.3 \times 10^{-2}$ (QK)	$\mathbf{112}\times$	-1.1×10^{-3}	-1×10^{-5} (proj)	n/a
Softmax dim=-2	$+1.4$	$+2.4$ (attn)	$2\times$	$+1.4$	$+2.5$ (attn)	$2\times$
FFN Block						
Unsafe LN	$+6.0 \times 10^{-2}$	$+1.8 \times 10^{-1}$ (lin1)	$3\times$	$+5.9 \times 10^{-2}$	$+2.6 \times 10^{-1}$ (lin1)	$4\times$
GELU sigmoid	$+9.9 \times 10^{-3}$	$+2.0 \times 10^{-2}$ (gelu)	$2\times$	-2.2×10^{-3}	$+1.6 \times 10^{-2}$ (gelu)	$\boldsymbol{\infty}$
Tile boundary	$+1.0 \times 10^{-1}$	$+2.1$ (lin1)	$\mathbf{21}\times$	$+8.4 \times 10^{-2}$	$+2.1$ (lin1)	$\mathbf{25}\times$
Wrong acc dtype	$+3.1 \times 10^{-3}$	$+3.1 \times 10^{-3}$ (lin2)	$1\times$	-3.6×10^{-3}	-5.7×10^{-4} (lin2)	n/a
Missing residual	$+4.3$	$+4.3$ (res)	$1\times$	$+4.4$	$+4.4$ (res)	$1\times$
ReLU not GELU	$+3.3 \times 10^{-1}$	$+3.3 \times 10^{-1}$ (lin2)	$1\times$	$+2.8 \times 10^{-1}$	$+2.9 \times 10^{-1}$ (lin2)	$1\times$

F. Single Split Results

METHOD	Correct (↑)	$\text{Fast}_{0.5}$ (↑)	Fast_1 (↑)	Fast_2 (↑)	# Tokens
MKEvolve	**0.99**	0.77	0.49	**0.09**	1.7×10^6
MKEvolve (Single Split)	0.98	**0.80**	**0.53**	0.07	1.7×10^6

(a) Claude Opus 4.5 Metrics on 100 KernelBench level 2 problems

METHOD	Correct (↑)	$\text{Fast}_{0.5}$ (↑)	Fast_1 (↑)	Fast_2 (↑)	# Tokens
MKEvolve	**0.94**	**0.74**	0.34	**0.08**	2.1×10^6
MKEvolve (Single Split)	0.90	0.62	**0.36**	**0.08**	2.1×10^6

(b) Claude Opus 4.5 Metrics on 50 KernelBench level 3 problems

Table 15. Comparison of MKEvolve that iteratively refines the subproblem structure against MKEvolve (Single Split) that decomposes the main problem into subproblems once.

G. Example Solution for ConvolutionalVisionTransformer

This section contains a sample MKEvolve output for the KernelBench L3 benchmark's ConvolutionalVisionTransformer problem. It first lists *decomposed_solution.py*, which is the end-to-end kernel implementation of the ConvolutionalVisionTransformer module. Then, it lists all the subkernels in the *subkernel_implementations* folder imported by *decomposed_solution.py*. We highlight that this modular structure allows easy replacement of subkernel implementations in the end-to-end kernel by placing different self-contained subkernel implementations in the *subkernel_implementations* folder.

G.1. Top-Level Module

decomposed_solution.py

```python
import sys
import torch.nn as nn
from subkernel_implementations.conv_patch_embed import ModelNew as ConvPatchEmbed
from subkernel_implementations.linear_proj import ModelNew as LinearProj
from subkernel_implementations.cls_token_concat import ModelNew as ClsTokenConcat
from subkernel_implementations.self_attention_block import ModelNew as SelfAttentionBlock
from subkernel_implementations.feedforward_block import ModelNew as FeedforwardBlock
from subkernel_implementations.classification_head import ModelNew as ClassificationHead


class ModelNew(nn.Module):
    def __init__(self, num_classes, embed_dim=512, num_heads=8, num_layers=6,
                 mlp_ratio=4.0, patch_size=4, in_channels=3, image_size=32):
        super(ModelNew, self).__init__()

        self.patch_size = patch_size
        self.image_size = image_size
        self.embed_dim = embed_dim

        num_patches = (image_size // patch_size) ** 2

        # Match original initialization order exactly
        self.conv_patch_embed = ConvPatchEmbed(in_channels, embed_dim, patch_size)
        self.linear_proj = LinearProj(embed_dim * num_patches, embed_dim)

        # TransformerEncoderLayers - 6 instances, each split into self_attention + feedforward
        # Order must match: each layer's self_attn linear layers, then feedforward linear layers
        self.self_attention_0 = SelfAttentionBlock(embed_dim, num_heads)
        self.feedforward_0 = FeedforwardBlock(embed_dim, int(embed_dim * mlp_ratio))
        self.self_attention_1 = SelfAttentionBlock(embed_dim, num_heads)
        self.feedforward_1 = FeedforwardBlock(embed_dim, int(embed_dim * mlp_ratio))
        self.self_attention_2 = SelfAttentionBlock(embed_dim, num_heads)
        self.feedforward_2 = FeedforwardBlock(embed_dim, int(embed_dim * mlp_ratio))
        self.self_attention_3 = SelfAttentionBlock(embed_dim, num_heads)
        self.feedforward_3 = FeedforwardBlock(embed_dim, int(embed_dim * mlp_ratio))
        self.self_attention_4 = SelfAttentionBlock(embed_dim, num_heads)
        self.feedforward_4 = FeedforwardBlock(embed_dim, int(embed_dim * mlp_ratio))
        self.self_attention_5 = SelfAttentionBlock(embed_dim, num_heads)
        self.feedforward_5 = FeedforwardBlock(embed_dim, int(embed_dim * mlp_ratio))

        # CLS token (Parameter created after transformer layers)
        self.cls_token_concat = ClsTokenConcat(embed_dim)

        self.classification_head = ClassificationHead(embed_dim, num_classes)

    def forward(self, x):
        B = x.size(0)

        x = self.conv_patch_embed(x)
        x = self.linear_proj(x)
        x = self.cls_token_concat(x, B)

        x = self.self_attention_0(x)
        x = self.feedforward_0(x)
        x = self.self_attention_1(x)
        x = self.feedforward_1(x)
        x = self.self_attention_2(x)
        x = self.feedforward_2(x)
        x = self.self_attention_3(x)
```

```
        x = self.feedforward_3(x)
        x = self.self_attention_4(x)
        x = self.feedforward_4(x)
        x = self.self_attention_5(x)
        x = self.feedforward_5(x)

        return self.classification_head(x)
```

G.2. Subkernels

subkernel_implementations/conv_patch_embed.py

```
import triton
import triton.language as tl
import torch
import torch.nn as nn
from torch.nn import init


@triton.autotune(
    configs=[
        triton.Config({'BLOCK_M': 16, 'BLOCK_N': 32, 'BLOCK_K': 16}, num_warps=2),
        triton.Config({'BLOCK_M': 16, 'BLOCK_N': 64, 'BLOCK_K': 16}, num_warps=2),
        triton.Config({'BLOCK_M': 32, 'BLOCK_N': 32, 'BLOCK_K': 16}, num_warps=2),
        triton.Config({'BLOCK_M': 32, 'BLOCK_N': 64, 'BLOCK_K': 16}, num_warps=4),
        triton.Config({'BLOCK_M': 32, 'BLOCK_N': 64, 'BLOCK_K': 32}, num_warps=4),
        triton.Config({'BLOCK_M': 64, 'BLOCK_N': 64, 'BLOCK_K': 32}, num_warps=4),
        triton.Config({'BLOCK_M': 64, 'BLOCK_N': 128, 'BLOCK_K': 32}, num_warps=4),
    ],
    key=['M', 'N', 'K'],
)
@triton.jit
def patchify_conv_kernel(
    x_ptr, weight_ptr, bias_ptr, out_ptr,
    M, N, K,
    B, num_patches, C_in, patch_size, H_out, W_out, H, W,
    stride_xb, stride_xc, stride_xh, stride_xw,
    out_stride_b, out_stride_f,
    BLOCK_M: tl.constexpr, BLOCK_N: tl.constexpr, BLOCK_K: tl.constexpr,
):
    pid_m = tl.program_id(0)
    pid_n = tl.program_id(1)

    offs_m = pid_m * BLOCK_M + tl.arange(0, BLOCK_M)
    offs_n = pid_n * BLOCK_N + tl.arange(0, BLOCK_N)

    # Decompose M index into batch and patch indices
    batch_idx = offs_m // num_patches
    patch_idx = offs_m % num_patches
    ph = patch_idx // W_out
    pw = patch_idx % W_out

    h_base = ph * patch_size
    w_base = pw * patch_size

    acc = tl.zeros((BLOCK_M, BLOCK_N), dtype=tl.float32)

    patch_area = patch_size * patch_size

    for k_start in range(0, K, BLOCK_K):
        offs_k = k_start + tl.arange(0, BLOCK_K)

        ic = offs_k // patch_area
        k_spatial = offs_k % patch_area
        kh = k_spatial // patch_size
```

```
        kw = k_spatial % patch_size

        # x[batch, ic, h_base + kh, w_base + kw]
        h_idx = h_base[:, None] + kh[None, :]
        w_idx = w_base[:, None] + kw[None, :]

        x_ptrs = (x_ptr +
                  batch_idx[:, None] * stride_xb +
                  ic[None, :] * stride_xc +
                  h_idx * stride_xh +
                  w_idx * stride_xw)

        mask_m = offs_m[:, None] < M
        mask_k = offs_k[None, :] < K
        x_vals = tl.load(x_ptrs, mask=mask_m & mask_k, other=0.0)

        # weight[oc, k] where k = ic * patch_area + kh * patch_size + kw
        w_ptrs = weight_ptr + offs_n[:, None] * K + offs_k[None, :]
        mask_n = offs_n[:, None] < N
        w_vals = tl.load(w_ptrs, mask=mask_n & (offs_k[None, :] < K), other=0.0)

        acc += tl.dot(x_vals, tl.trans(w_vals), allow_tf32=False)

    # Add bias
    bias_vals = tl.load(bias_ptr + offs_n, mask=offs_n < N, other=0.0)
    acc += bias_vals[None, :]

    # Store with flattened layout: out[b, oc * num_patches + patch_idx]
    # This matches flatten(start_dim=1) on (B, embed_dim, H_out, W_out)
    out_batch = offs_m[:, None] // num_patches
    out_patch = offs_m[:, None] % num_patches
    out_idx = offs_n[None, :] * num_patches + out_patch

    out_ptrs = out_ptr + out_batch * out_stride_b + out_idx
    mask_out = (offs_m[:, None] < M) & (offs_n[None, :] < N)

    tl.store(out_ptrs, acc, mask=mask_out)


class ModelNew(nn.Module):
    def __init__(self, in_channels, embed_dim, patch_size):
        super().__init__()
        self.in_channels = in_channels
        self.embed_dim = embed_dim
        self.patch_size = patch_size

        weight = torch.empty(embed_dim, in_channels, patch_size, patch_size)
        init.kaiming_uniform_(weight, a=5**0.5)

        bias = torch.empty(embed_dim)
        fan_in, _ = init._calculate_fan_in_and_fan_out(weight)
        bound = 1 / fan_in**0.5 if fan_in > 0 else 0
        init.uniform_(bias, -bound, bound)

        self.weight = nn.Parameter(weight)
        self.bias = nn.Parameter(bias)
        self._weight_flat = None

    def forward(self, x):
        B, C_in, H, W = x.shape
        patch_size = self.patch_size
        embed_dim = self.embed_dim

        H_out = H // patch_size
        W_out = W // patch_size
```

```
        num_patches = H_out * W_out

        M = B * num_patches
        N = embed_dim
        K = C_in * patch_size * patch_size

        out = torch.empty((B, embed_dim * num_patches), device=x.device, dtype=x.dtype)

        # Cache flattened weight
        if self._weight_flat is None or self._weight_flat.data_ptr() != self.weight.
            data_ptr():
            self._weight_flat = self.weight.reshape(embed_dim, -1).contiguous()
        weight_flat = self._weight_flat

        grid = lambda META: (
            triton.cdiv(M, META['BLOCK_M']),
            triton.cdiv(N, META['BLOCK_N']),
        )

        patchify_conv_kernel[grid](
            x, weight_flat, self.bias, out,
            M, N, K,
            B, num_patches, C_in, patch_size, H_out, W_out, H, W,
            x.stride(0), x.stride(1), x.stride(2), x.stride(3),
            out.stride(0), out.stride(1),
        )

        return out
```

subkernel_implementations/linear_proj.py

```
import triton
import triton.language as tl
import torch
import torch.nn as nn
from torch.nn import init
import math


@triton.autotune(
    configs=[
        # For small M (batch=10), we need configs that handle this well
        triton.Config({'BLOCK_SIZE_M': 16, 'BLOCK_SIZE_N': 32, 'BLOCK_SIZE_K': 128, '
            GROUP_SIZE_M': 1}, num_stages=4, num_warps=4),
        triton.Config({'BLOCK_SIZE_M': 16, 'BLOCK_SIZE_N': 64, 'BLOCK_SIZE_K': 128, '
            GROUP_SIZE_M': 1}, num_stages=4, num_warps=4),
        triton.Config({'BLOCK_SIZE_M': 16, 'BLOCK_SIZE_N': 128, 'BLOCK_SIZE_K': 128, '
            GROUP_SIZE_M': 1}, num_stages=4, num_warps=4),
        triton.Config({'BLOCK_SIZE_M': 16, 'BLOCK_SIZE_N': 64, 'BLOCK_SIZE_K': 256, '
            GROUP_SIZE_M': 1}, num_stages=3, num_warps=4),
        triton.Config({'BLOCK_SIZE_M': 16, 'BLOCK_SIZE_N': 128, 'BLOCK_SIZE_K': 256, '
            GROUP_SIZE_M': 1}, num_stages=3, num_warps=4),
        triton.Config({'BLOCK_SIZE_M': 32, 'BLOCK_SIZE_N': 64, 'BLOCK_SIZE_K': 128, '
            GROUP_SIZE_M': 1}, num_stages=4, num_warps=4),
        triton.Config({'BLOCK_SIZE_M': 32, 'BLOCK_SIZE_N': 128, 'BLOCK_SIZE_K': 128, '
            GROUP_SIZE_M': 1}, num_stages=4, num_warps=4),
    ],
    key=['M', 'N', 'K'],
)
@triton.jit
def linear_kernel(
    x_ptr, weight_t_ptr, bias_ptr, output_ptr,
    M, N, K,
    stride_xm, stride_xk,
    stride_wtk, stride_wtn,
```

```
    stride_om, stride_on,
    BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_K: tl.constexpr,
    GROUP_SIZE_M: tl.constexpr,
):
    """
    Fused linear layer kernel: output = x @ weight_t + bias
    - x: (M, K)
    - weight_t: (K, N) - pre-transposed weight
    - bias: (N,)
    - output: (M, N)
    """
    pid = tl.program_id(axis=0)
    num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
    num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)

    pid_m = pid // num_pid_n
    pid_n = pid % num_pid_n

    offs_m = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    offs_n = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
    offs_k = tl.arange(0, BLOCK_SIZE_K)

    # Pointers for x (M, K) and weight_t (K, N)
    x_ptrs = x_ptr + offs_m[:, None] * stride_xm + offs_k[None, :] * stride_xk
    wt_ptrs = weight_t_ptr + offs_k[:, None] * stride_wtk + offs_n[None, :] * stride_wtn

    accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)

    for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
        k_offs = k * BLOCK_SIZE_K + offs_k

        x_mask = (offs_m[:, None] < M) & (k_offs[None, :] < K)
        wt_mask = (k_offs[:, None] < K) & (offs_n[None, :] < N)

        x = tl.load(x_ptrs, mask=x_mask, other=0.0)
        wt = tl.load(wt_ptrs, mask=wt_mask, other=0.0)

        # x: (BLOCK_SIZE_M, BLOCK_SIZE_K), wt: (BLOCK_SIZE_K, BLOCK_SIZE_N)
        # Result: (BLOCK_SIZE_M, BLOCK_SIZE_N)
        accumulator = tl.dot(x, wt, accumulator, allow_tf32=False)

        x_ptrs += BLOCK_SIZE_K * stride_xk
        wt_ptrs += BLOCK_SIZE_K * stride_wtk

    # Add bias - load once and broadcast
    bias_mask = offs_n < N
    bias = tl.load(bias_ptr + offs_n, mask=bias_mask, other=0.0)
    accumulator = accumulator + bias[None, :]

    # Store output
    output_ptrs = output_ptr + offs_m[:, None] * stride_om + offs_n[None, :] * stride_on
    output_mask = (offs_m[:, None] < M) & (offs_n[None, :] < N)
    tl.store(output_ptrs, accumulator.to(tl.float32), mask=output_mask)


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

        # Initialize weight: (out_features, in_features)
        weight = torch.empty((out_features, in_features))
        init.kaiming_uniform_(weight, a=math.sqrt(5))

```

```
        # Store transposed weight for efficient matmul: (in_features, out_features)
        self.weight_t = nn.Parameter(weight.t().contiguous())

        # Initialize bias: (out_features,)
        bias = torch.empty(out_features)
        fan_in, _ = init._calculate_fan_in_and_fan_out(weight)
        bound = 1 / math.sqrt(fan_in) if fan_in > 0 else 0
        init.uniform_(bias, -bound, bound)
        self.bias = nn.Parameter(bias)

    def forward(self, x):
        assert x.is_cuda, "Input must be on CUDA"
        x = x.contiguous()

        M, K = x.shape
        N = self.out_features

        # Allocate output
        output = torch.empty((M, N), device=x.device, dtype=x.dtype)

        # Grid calculation
        grid = lambda META: (
            triton.cdiv(M, META['BLOCK_SIZE_M']) * triton.cdiv(N, META['BLOCK_SIZE_N']),
        )

        # Launch kernel with pre-transposed weight
        linear_kernel[grid](
            x, self.weight_t, self.bias, output,
            M, N, K,
            x.stride(0), x.stride(1),
            self.weight_t.stride(0), self.weight_t.stride(1),
            output.stride(0), output.stride(1),
        )

        return output
```

subkernel_implementations/cls_token_concat.py

```
import triton
import triton.language as tl
import torch
import torch.nn as nn


@triton.autotune(
    configs=[
        triton.Config({'BLOCK_SIZE': 32}, num_warps=1),
        triton.Config({'BLOCK_SIZE': 64}, num_warps=2),
        triton.Config({'BLOCK_SIZE': 128}, num_warps=2),
        triton.Config({'BLOCK_SIZE': 256}, num_warps=4),
    ],
    key=['total_elements'],
)
@triton.jit
def cls_token_concat_kernel(
    x_ptr,            # Input tensor (B, embed_dim)
    cls_token_ptr,    # CLS token flattened (embed_dim,)
    output_ptr,       # Output tensor (B, 2, embed_dim)
    batch_size,       # B
    embed_dim,        # embed_dim
    total_elements,   # B * embed_dim
    BLOCK_SIZE: tl.constexpr,
):
    """
    Flattened kernel: each program handles BLOCK_SIZE elements.
```

```
    We process both cls_token copy and x copy in a unified manner.
    """
    pid = tl.program_id(0)

    # Each program handles BLOCK_SIZE elements from the flattened (B, embed_dim) space
    offs = pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)
    mask = offs < total_elements

    # Compute batch index and embed index
    batch_idx = offs // embed_dim
    embed_idx = offs % embed_dim

    # Load cls_token values (indexed by embed_idx)
    cls_vals = tl.load(cls_token_ptr + embed_idx, mask=mask, other=0.0)

    # Load x values
    x_vals = tl.load(x_ptr + offs, mask=mask, other=0.0)

    # Output layout: (B, 2, embed_dim)
    # output[b, 0, e] = cls_token[e] at offset: b * 2 * embed_dim + e
    # output[b, 1, e] = x[b, e] at offset: b * 2 * embed_dim + embed_dim + e

    out_cls_offs = batch_idx * 2 * embed_dim + embed_idx
    out_x_offs = batch_idx * 2 * embed_dim + embed_dim + embed_idx

    # Store cls_token at position [:, 0, :]
    tl.store(output_ptr + out_cls_offs, cls_vals, mask=mask)

    # Store x at position [:, 1, :]
    tl.store(output_ptr + out_x_offs, x_vals, mask=mask)


class ModelNew(nn.Module):
    """
    Optimized Triton implementation of CLS token concatenation.
    """

    def __init__(self, embed_dim):
        super().__init__()
        # Initialize cls_token exactly as nn.Parameter(torch.zeros(1, 1, embed_dim))
        self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))

    def forward(self, x, batch_size):
        """
        Args:
            x: Input tensor of shape (B, embed_dim)
            batch_size: Number of batch elements (B)

        Returns:
            Output tensor of shape (B, 2, embed_dim)
        """
        embed_dim = x.shape[1]
        total_elements = batch_size * embed_dim

        # Allocate output tensor
        output = torch.empty((batch_size, 2, embed_dim), device=x.device, dtype=x.dtype)

        # Flatten cls_token for kernel access (contiguous)
        cls_token_flat = self.cls_token.view(-1)

        # 1D grid based on total elements
        def grid(META):
            return (triton.cdiv(total_elements, META['BLOCK_SIZE']),)

        cls_token_concat_kernel[grid](
```

```
            x,
            cls_token_flat,
            output,
            batch_size,
            embed_dim,
            total_elements,
        )

        return output
```

subkernel_implementations/self_attention_block.py

```
import triton
import triton.language as tl
import torch
import torch.nn as nn
from torch.nn import init
import math

@triton.autotune(
    configs=[
        triton.Config({'BLOCK_M': 32, 'BLOCK_K': 64, 'BLOCK_N': 64}, num_warps=4),
        triton.Config({'BLOCK_M': 32, 'BLOCK_K': 128, 'BLOCK_N': 128}, num_warps=4),
        triton.Config({'BLOCK_M': 64, 'BLOCK_K': 64, 'BLOCK_N': 64}, num_warps=4),
    ],
    key=['M', 'K', 'N'],
)
@triton.jit
def matmul_bias_kernel(
    x_ptr, weight_ptr, bias_ptr, out_ptr,
    M, K, N,
    stride_xm, stride_xk,
    stride_wn, stride_wk,
    stride_om, stride_on,
    BLOCK_M: tl.constexpr, BLOCK_K: tl.constexpr, BLOCK_N: tl.constexpr,
):
    pid_m = tl.program_id(0)
    pid_n = tl.program_id(1)

    offs_m = pid_m * BLOCK_M + tl.arange(0, BLOCK_M)
    offs_n = pid_n * BLOCK_N + tl.arange(0, BLOCK_N)

    acc = tl.zeros((BLOCK_M, BLOCK_N), dtype=tl.float32)

    for k in range(0, K, BLOCK_K):
        offs_k = k + tl.arange(0, BLOCK_K)
        mask_k = offs_k < K
        mask_m = offs_m < M
        mask_n = offs_n < N

        x = tl.load(x_ptr + offs_m[:, None] * stride_xm + offs_k[None, :] * stride_xk,
                    mask=mask_m[:, None] & mask_k[None, :], other=0.0)
        w = tl.load(weight_ptr + offs_n[:, None] * stride_wn + offs_k[None, :] * stride_wk
            ,
                    mask=mask_n[:, None] & mask_k[None, :], other=0.0)

        acc += tl.dot(x, tl.trans(w), allow_tf32=False)

    mask_m = offs_m < M
    mask_n = offs_n < N
    bias = tl.load(bias_ptr + offs_n, mask=mask_n, other=0.0)
    acc += bias[None, :]

    tl.store(out_ptr + offs_m[:, None] * stride_om + offs_n[None, :] * stride_on,
             acc, mask=mask_m[:, None] & mask_n[None, :])
```

```python


@triton.jit
def attention_kernel_v2(
    qkv_ptr, out_ptr,
    batch_size, seq_len, num_heads, head_dim, embed_dim,
    scale,
    HEAD_DIM: tl.constexpr,
):
    pid_batch = tl.program_id(0)
    pid_head = tl.program_id(1)
    pid_seq = tl.program_id(2)

    d_offs = tl.arange(0, HEAD_DIM)
    d_mask = d_offs < head_dim

    # Q location: [batch, seq, 0:embed_dim], head offset
    q_base = pid_batch * seq_len * 3 * embed_dim + pid_seq * 3 * embed_dim + pid_head *
        head_dim
    q_vec = tl.load(qkv_ptr + q_base + d_offs, mask=d_mask, other=0.0)

    max_score = -1e9
    sum_exp = 0.0
    out_acc = tl.zeros((HEAD_DIM,), dtype=tl.float32)

    for j in range(seq_len):
        kv_base = pid_batch * seq_len * 3 * embed_dim + j * 3 * embed_dim + pid_head *
            head_dim
        k_vec = tl.load(qkv_ptr + kv_base + embed_dim + d_offs, mask=d_mask, other=0.0)

        score = tl.sum(q_vec * k_vec, axis=0) * scale

        new_max = tl.maximum(max_score, score)
        correction = tl.exp(max_score - new_max)
        exp_score = tl.exp(score - new_max)

        out_acc = out_acc * correction
        sum_exp = sum_exp * correction + exp_score

        v_vec = tl.load(qkv_ptr + kv_base + 2 * embed_dim + d_offs, mask=d_mask, other
            =0.0)
        out_acc += exp_score * v_vec
        max_score = new_max

    out_acc = out_acc / (sum_exp + 1e-9)

    out_offset = pid_batch * seq_len * embed_dim + pid_seq * embed_dim + pid_head *
        head_dim
    tl.store(out_ptr + out_offset + d_offs, out_acc, mask=d_mask)


@triton.jit
def fused_proj_residual_ln_kernel(
    attn_ptr, x_ptr, w_ptr, b_ptr, ln_w_ptr, ln_b_ptr, out_ptr,
    M, K, eps,
    BLOCK_K: tl.constexpr,
):
    pid = tl.program_id(0)
    offs_k = tl.arange(0, BLOCK_K)
    mask_k = offs_k < K

    # Load input row
    attn_row = tl.load(attn_ptr + pid * K + offs_k, mask=mask_k, other=0.0)
    x_row = tl.load(x_ptr + pid * K + offs_k, mask=mask_k, other=0.0)

```

```
    # Output projection: proj_out[k] = sum_j(attn[j] * W[k, j])
    proj_out = tl.zeros((BLOCK_K,), dtype=tl.float32)

    for j in range(0, K, BLOCK_K):
        j_offs = j + tl.arange(0, BLOCK_K)
        j_mask = j_offs < K

        attn_chunk = tl.load(attn_ptr + pid * K + j_offs, mask=j_mask, other=0.0)
        # W[offs_k, j_offs] - load BLOCK_K x BLOCK_K tile
        w_tile = tl.load(w_ptr + offs_k[:, None] * K + j_offs[None, :],
                         mask=mask_k[:, None] & j_mask[None, :], other=0.0)
        proj_out += tl.sum(w_tile * attn_chunk[None, :], axis=1)

    # Add bias and residual
    bias = tl.load(b_ptr + offs_k, mask=mask_k, other=0.0)
    residual = proj_out + bias + x_row

    # Layer norm
    mean = tl.sum(residual, axis=0) / K
    centered = residual - mean
    var = tl.sum(centered * centered, axis=0) / K
    inv_std = 1.0 / tl.sqrt(var + eps)

    normalized = centered * inv_std
    gamma = tl.load(ln_w_ptr + offs_k, mask=mask_k, other=1.0)
    beta = tl.load(ln_b_ptr + offs_k, mask=mask_k, other=0.0)

    out = normalized * gamma + beta
    tl.store(out_ptr + pid * K + offs_k, out, mask=mask_k)


class ModelNew(nn.Module):
    def __init__(self, embed_dim, num_heads):
        super().__init__()
        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.head_dim = embed_dim // num_heads

        in_proj_weight = torch.empty(3 * embed_dim, embed_dim)
        in_proj_bias = torch.empty(3 * embed_dim)
        out_proj_weight = torch.empty(embed_dim, embed_dim)
        init.kaiming_uniform_(out_proj_weight, a=math.sqrt(5))
        out_proj_bias = torch.empty(embed_dim)
        fan_in, _ = init._calculate_fan_in_and_fan_out(out_proj_weight)
        bound = 1 / math.sqrt(fan_in) if fan_in > 0 else 0
        init.uniform_(out_proj_bias, -bound, bound)

        init.xavier_uniform_(in_proj_weight)
        init.constant_(in_proj_bias, 0.0)
        init.constant_(out_proj_bias, 0.0)

        ln_weight = torch.ones(embed_dim)
        ln_bias = torch.zeros(embed_dim)

        self.in_proj_weight = nn.Parameter(in_proj_weight)
        self.in_proj_bias = nn.Parameter(in_proj_bias)
        self.out_proj_weight = nn.Parameter(out_proj_weight)
        self.out_proj_bias = nn.Parameter(out_proj_bias)
        self.ln_weight = nn.Parameter(ln_weight)
        self.ln_bias = nn.Parameter(ln_bias)
        self.eps = 1e-5

    def forward(self, x):
        batch_size, seq_len, embed_dim = x.shape
        batch_seq = batch_size * seq_len
```

```
        x_flat = x.reshape(batch_seq, embed_dim)

        qkv = torch.empty(batch_seq, 3 * embed_dim, device=x.device, dtype=x.dtype)

        grid_qkv = lambda META: (triton.cdiv(batch_seq, META['BLOCK_M']), triton.cdiv(3 *
            embed_dim, META['BLOCK_N']))
        matmul_bias_kernel[grid_qkv](
            x_flat, self.in_proj_weight, self.in_proj_bias, qkv,
            batch_seq, embed_dim, 3 * embed_dim,
            x_flat.stride(0), x_flat.stride(1),
            self.in_proj_weight.stride(0), self.in_proj_weight.stride(1),
            qkv.stride(0), qkv.stride(1),
        )

        qkv_reshaped = qkv.view(batch_size, seq_len, 3 * embed_dim)
        attn_out = torch.empty(batch_size, seq_len, embed_dim, device=x.device, dtype=x.
            dtype)
        scale = 1.0 / math.sqrt(self.head_dim)
        HEAD_DIM = triton.next_power_of_2(self.head_dim)

        attention_kernel_v2[(batch_size, self.num_heads, seq_len)](
            qkv_reshaped, attn_out,
            batch_size, seq_len, self.num_heads, self.head_dim, embed_dim,
            scale,
            HEAD_DIM=HEAD_DIM,
        )

        attn_flat = attn_out.reshape(batch_seq, embed_dim)
        output = torch.empty_like(attn_flat)

        BLOCK_K = triton.next_power_of_2(embed_dim)
        fused_proj_residual_ln_kernel[(batch_seq,)](
            attn_flat, x_flat, self.out_proj_weight, self.out_proj_bias,
            self.ln_weight, self.ln_bias, output,
            batch_seq, embed_dim, self.eps,
            BLOCK_K=BLOCK_K,
        )

        return output.view(batch_size, seq_len, embed_dim).contiguous()
```

subkernel_implementations/feedforward_block.py

```
import triton
import triton.language as tl
import torch
import torch.nn as nn
from torch.nn import init
import math


@triton.autotune(
    configs=[
        triton.Config({'BLOCK_M': 16, 'BLOCK_N': 64, 'BLOCK_K': 32}, num_warps=4),
        triton.Config({'BLOCK_M': 16, 'BLOCK_N': 128, 'BLOCK_K': 32}, num_warps=4),
        triton.Config({'BLOCK_M': 32, 'BLOCK_N': 64, 'BLOCK_K': 32}, num_warps=4),
        triton.Config({'BLOCK_M': 32, 'BLOCK_N': 128, 'BLOCK_K': 32}, num_warps=4),
        triton.Config({'BLOCK_M': 16, 'BLOCK_N': 256, 'BLOCK_K': 32}, num_warps=4),
        triton.Config({'BLOCK_M': 32, 'BLOCK_N': 256, 'BLOCK_K': 32}, num_warps=8),
    ],
    key=['M', 'K', 'N'],
)
@triton.jit
def fused_linear_relu_kernel(
    x_ptr, w1_ptr, b1_ptr, out_ptr,
    M, K, N,
```

```
    stride_xm, stride_xk,
    stride_w1n, stride_w1k,
    stride_outm, stride_outn,
    BLOCK_M: tl.constexpr, BLOCK_N: tl.constexpr, BLOCK_K: tl.constexpr,
):
    pid_m = tl.program_id(0)
    pid_n = tl.program_id(1)

    offs_m = pid_m * BLOCK_M + tl.arange(0, BLOCK_M)
    offs_n = pid_n * BLOCK_N + tl.arange(0, BLOCK_N)

    acc = tl.zeros((BLOCK_M, BLOCK_N), dtype=tl.float32)

    for k in range(0, K, BLOCK_K):
        offs_k = k + tl.arange(0, BLOCK_K)

        x_ptrs = x_ptr + offs_m[:, None] * stride_xm + offs_k[None, :] * stride_xk
        w_ptrs = w1_ptr + offs_n[:, None] * stride_w1n + offs_k[None, :] * stride_w1k

        x_mask = (offs_m[:, None] < M) & (offs_k[None, :] < K)
        w_mask = (offs_n[:, None] < N) & (offs_k[None, :] < K)

        x = tl.load(x_ptrs, mask=x_mask, other=0.0)
        w = tl.load(w_ptrs, mask=w_mask, other=0.0)

        acc += tl.dot(x, tl.trans(w), allow_tf32=False)

    bias = tl.load(b1_ptr + offs_n, mask=offs_n < N, other=0.0)
    acc = acc + bias[None, :]
    acc = tl.maximum(acc, 0.0)

    out_ptrs = out_ptr + offs_m[:, None] * stride_outm + offs_n[None, :] * stride_outn
    out_mask = (offs_m[:, None] < M) & (offs_n[None, :] < N)
    tl.store(out_ptrs, acc, mask=out_mask)


@triton.jit
def fused_linear2_residual_ln_kernel(
    hidden_ptr, w2_ptr, b2_ptr, x_ptr, gamma_ptr, beta_ptr, out_ptr,
    M, K, N, eps,
    stride_hm, stride_hk,
    stride_w2n, stride_w2k,
    stride_xm, stride_xn,
    stride_outm, stride_outn,
    BLOCK_K: tl.constexpr, BLOCK_N: tl.constexpr,
):
    pid_m = tl.program_id(0)
    offs_n = tl.arange(0, BLOCK_N)
    n_mask = offs_n < N

    acc = tl.zeros((BLOCK_N,), dtype=tl.float32)

    # Load hidden row once
    for k in range(0, K, BLOCK_K):
        offs_k = k + tl.arange(0, BLOCK_K)
        k_mask = offs_k < K

        h = tl.load(hidden_ptr + pid_m * stride_hm + offs_k, mask=k_mask, other=0.0)

        w_ptrs = w2_ptr + offs_n[:, None] * stride_w2n + offs_k[None, :] * stride_w2k
        w_mask = n_mask[:, None] & k_mask[None, :]
        w = tl.load(w_ptrs, mask=w_mask, other=0.0)

        acc += tl.sum(w * h[None, :], axis=1)

```

```
    # Add bias
    bias = tl.load(b2_ptr + offs_n, mask=n_mask, other=0.0)
    acc = acc + bias

    # Residual connection
    x = tl.load(x_ptr + pid_m * stride_xm + offs_n, mask=n_mask, other=0.0)
    y = acc + x

    # Layer norm
    mean = tl.sum(y, axis=0) / N
    diff = y - mean
    var = tl.sum(diff * diff, axis=0) / N
    rstd = 1.0 / tl.sqrt(var + eps)

    gamma = tl.load(gamma_ptr + offs_n, mask=n_mask, other=1.0)
    beta = tl.load(beta_ptr + offs_n, mask=n_mask, other=0.0)

    out = diff * rstd * gamma + beta
    tl.store(out_ptr + pid_m * stride_outm + offs_n, out, mask=n_mask)


class ModelNew(nn.Module):
    def __init__(self, embed_dim, dim_feedforward):
        super().__init__()
        self.embed_dim = embed_dim
        self.dim_feedforward = dim_feedforward

        # Linear1 initialization
        self.weight1 = torch.empty(dim_feedforward, embed_dim)
        init.kaiming_uniform_(self.weight1, a=math.sqrt(5))
        fan_in1, _ = init._calculate_fan_in_and_fan_out(self.weight1)
        bound1 = 1 / math.sqrt(fan_in1) if fan_in1 > 0 else 0
        self.bias1 = torch.empty(dim_feedforward)
        init.uniform_(self.bias1, -bound1, bound1)

        # Linear2 initialization
        self.weight2 = torch.empty(embed_dim, dim_feedforward)
        init.kaiming_uniform_(self.weight2, a=math.sqrt(5))
        fan_in2, _ = init._calculate_fan_in_and_fan_out(self.weight2)
        bound2 = 1 / math.sqrt(fan_in2) if fan_in2 > 0 else 0
        self.bias2 = torch.empty(embed_dim)
        init.uniform_(self.bias2, -bound2, bound2)

        # LayerNorm initialization
        self.gamma = torch.ones(embed_dim)
        self.beta = torch.zeros(embed_dim)

        # Register as parameters
        self.weight1 = nn.Parameter(self.weight1)
        self.bias1 = nn.Parameter(self.bias1)
        self.weight2 = nn.Parameter(self.weight2)
        self.bias2 = nn.Parameter(self.bias2)
        self.gamma = nn.Parameter(self.gamma)
        self.beta = nn.Parameter(self.beta)

        self.eps = 1e-5

    def forward(self, x):
        orig_shape = x.shape
        x_flat = x.view(-1, self.embed_dim).contiguous()
        M = x_flat.shape[0]
        K1 = self.embed_dim
        N1 = self.dim_feedforward

        hidden = torch.empty(M, N1, device=x.device, dtype=x.dtype)
```

```python

        grid1 = lambda META: (triton.cdiv(M, META['BLOCK_M']), triton.cdiv(N1, META['
            BLOCK_N']))
        fused_linear_relu_kernel[grid1](
            x_flat, self.weight1, self.bias1, hidden,
            M, K1, N1,
            x_flat.stride(0), x_flat.stride(1),
            self.weight1.stride(0), self.weight1.stride(1),
            hidden.stride(0), hidden.stride(1),
        )

        out = torch.empty(M, self.embed_dim, device=x.device, dtype=x.dtype)
        BLOCK_N2 = triton.next_power_of_2(self.embed_dim)
        BLOCK_K2 = min(128, triton.next_power_of_2(N1))

        fused_linear2_residual_ln_kernel[(M,)](
            hidden, self.weight2, self.bias2, x_flat, self.gamma, self.beta, out,
            M, N1, self.embed_dim, self.eps,
            hidden.stride(0), hidden.stride(1),
            self.weight2.stride(0), self.weight2.stride(1),
            x_flat.stride(0), x_flat.stride(1),
            out.stride(0), out.stride(1),
            BLOCK_K=BLOCK_K2, BLOCK_N=BLOCK_N2,
        )

        return out.view(orig_shape).contiguous()
```

subkernel_implementations/classification_head.py

```python
import triton
import triton.language as tl
import torch
import torch.nn as nn
from torch.nn import init
import math


@triton.autotune(
    configs=[
        triton.Config({'BLOCK_N': 64, 'BLOCK_K': 32}, num_warps=2),
        triton.Config({'BLOCK_N': 128, 'BLOCK_K': 32}, num_warps=4),
        triton.Config({'BLOCK_N': 64, 'BLOCK_K': 64}, num_warps=2),
        triton.Config({'BLOCK_N': 128, 'BLOCK_K': 64}, num_warps=4),
        triton.Config({'BLOCK_N': 256, 'BLOCK_K': 32}, num_warps=4),
        triton.Config({'BLOCK_N': 256, 'BLOCK_K': 64}, num_warps=4),
    ],
    key=['N', 'K'],
)
@triton.jit
def fused_cls_linear_kernel(
    x_ptr,          # Input: (B, 2, embed_dim)
    weight_ptr,     # Weight: (num_classes, embed_dim)
    bias_ptr,       # Bias: (num_classes,)
    output_ptr,     # Output: (B, num_classes)
    B,              # Batch size
    N,              # num_classes
    K,              # embed_dim
    stride_x_b,     # Stride for batch dimension in x
    stride_x_k,     # Stride for embed_dim in x
    stride_w_n,     # Stride for output features in weight
    stride_w_k,     # Stride for input features in weight
    stride_o_b,     # Stride for batch in output
    stride_o_n,     # Stride for classes in output
    BLOCK_N: tl.constexpr,
    BLOCK_K: tl.constexpr,
```

```
):
    """
    Each program handles one batch element and a block of output classes.
    pid_m = batch index, pid_n = output class block
    """
    pid_m = tl.program_id(0)  # Batch index
    pid_n = tl.program_id(1)  # Output class block

    # Output class offsets for this block
    offs_n = pid_n * BLOCK_N + tl.arange(0, BLOCK_N)
    mask_n = offs_n < N

    # Initialize accumulator
    acc = tl.zeros((BLOCK_N,), dtype=tl.float32)

    # Base pointer to x[pid_m, 0, :] - CLS token for this batch element
    x_base = x_ptr + pid_m * stride_x_b

    # Loop over K dimension
    for k_start in range(0, K, BLOCK_K):
        offs_k = k_start + tl.arange(0, BLOCK_K)
        k_mask = offs_k < K

        # Load x vector: (BLOCK_K,)
        x_vec = tl.load(x_base + offs_k * stride_x_k, mask=k_mask, other=0.0)

        # Load weight block: (BLOCK_N, BLOCK_K) -> we need (BLOCK_K, BLOCK_N) for dot
        # weight[n, k] -> weight_ptr + n * stride_w_n + k * stride_w_k
        w_ptrs = weight_ptr + offs_n[:, None] * stride_w_n + offs_k[None, :] * stride_w_k
        w_block = tl.load(w_ptrs, mask=mask_n[:, None] & k_mask[None, :], other=0.0)

        # Compute: sum over k of x[k] * w[n, k] for each n
        # This is equivalent to w_block @ x_vec for each output class
        acc += tl.sum(w_block * x_vec[None, :], axis=1)

    # Add bias
    bias = tl.load(bias_ptr + offs_n, mask=mask_n, other=0.0)
    acc = acc + bias

    # Store output
    out_ptrs = output_ptr + pid_m * stride_o_b + offs_n * stride_o_n
    tl.store(out_ptrs, acc, mask=mask_n)


class ModelNew(nn.Module):
    def __init__(self, embed_dim, num_classes):
        super().__init__()
        self.embed_dim = embed_dim
        self.num_classes = num_classes

        # Initialize weight: (num_classes, embed_dim)
        self.weight = nn.Parameter(torch.empty(num_classes, embed_dim))
        init.kaiming_uniform_(self.weight, a=math.sqrt(5))

        # Initialize bias: (num_classes,)
        self.bias = nn.Parameter(torch.empty(num_classes))
        fan_in, _ = init._calculate_fan_in_and_fan_out(self.weight)
        bound = 1 / math.sqrt(fan_in) if fan_in > 0 else 0
        init.uniform_(self.bias, -bound, bound)

    def forward(self, x):
        B = x.shape[0]

        # Ensure contiguous for efficient access
        x = x.contiguous()
```

```
        # Allocate output
        output = torch.empty((B, self.num_classes), device=x.device, dtype=x.dtype)

        # Grid: one program per (batch element, output class block)
        grid = lambda META: (B, triton.cdiv(self.num_classes, META['BLOCK_N']))

        # Launch kernel
        fused_cls_linear_kernel[grid](
            x, self.weight, self.bias, output,
            B, self.num_classes, self.embed_dim,
            x.stride(0), x.stride(2),
            self.weight.stride(0), self.weight.stride(1),
            output.stride(0), output.stride(1),
        )

        return output
```

H. Prompts

H.1. Subproblem Decomposition

```
Decompose the PyTorch module under the "PROBLEM TO DECOMPOSE" section into
 ↪  submodules optimized for kernel fusion.

## TASK OBJECTIVE

Create a `ModelNew` nn.Module in `decomposed_problem.py` that is **functionally
 ↪  equivalent** to the `Model` class in the problem description below. The
 ↪  `ModelNew` module should:
- Accept the same inputs as the original `Model`
- Produce numerically identical outputs
- Be composed of submodules from `subkernel_problems/` that can each be
 ↪  independently optimized as fused kernels
- Not use any other torch.nn modules in its code than submodules from
 ↪  `subkernel_problems/`, including modules such as `nn.ModuleList` and
 ↪  `nn.Parameters`
- Do all mathematical computations in the files within `subkernel_problems/` and
 ↪  have no mathematical computations directly in the file
 ↪  `decomposed_problem.py`
- Avoid creating submodules that only perform highly trivial operations such as
 ↪  no-op, identity function, copy function, contiguous operation, flatten
 ↪  operation, packing and unpacking data, etc
- Avoid creating submodules that pass one torch.nn Module as an argument to
 ↪  another torch.nn Module, the only exceptions applying to nn.Sequential,
 ↪  nn.ModuleList, and nn.ModuleDict

Note on Numerical Equivalence: PyTorch uses a global random number generator for
 ↪  weight initialization. When `ModelNew.__init__` instantiates subkernel
 ↪  modules (from `subkernel_problems/`), those subkernel modules internally
 ↪  create `torch.nn` modules. The order in which `ModelNew.__init__`
 ↪  instantiates subkernel modules must ensure the underlying `torch.nn` modules
 ↪  are created in the EXACT SAME ORDER as they appear in the original
 ↪  `Model.__init__`. For example, if the original model has:
```python
self.conv1 = nn.Conv2d(3, 64, 3)   # consumes RNG state first
self.bn1 = nn.BatchNorm2d(64)      # consumes RNG state second
self.conv2 = nn.Conv2d(64, 128, 3) # consumes RNG state third
```
```

```
Then `ModelNew.__init__` must instantiate its subkernel modules in an order that
↪  causes Conv2d(3,64,3) to be created first, then BatchNorm2d(64), then
↪  Conv2d(64,128,3). Misordering causes different random values to be assigned
↪  to weights, resulting in numerical mismatch.
This also applies when decomposing a submodule containing a single torch.nn layer
↪  that contains multiple torch.nn layers such as nn.MultiheadAttention. For
↪  reference, its parameter initialization method is
```python
def _reset_parameters(self):
    if self._qkv_same_embed_dim:
        xavier_uniform_(self.in_proj_weight)
    else:
        xavier_uniform_(self.q_proj_weight)
        xavier_uniform_(self.k_proj_weight)
        xavier_uniform_(self.v_proj_weight)
    if self.in_proj_bias is not None:
        constant_(self.in_proj_bias, 0.0)
        constant_(self.out_proj.bias, 0.0)
    if self.bias_k is not None:
        xavier_normal_(self.bias_k)
    if self.bias_v is not None:
        xavier_normal_(self.bias_v)
```

VERIFICATION CRITERIA

Your decomposition will be validated against these checks:

1. **Functional Equivalence**: `ModelNew(*get_inputs())` must produce the same
↪ output as `Model(*get_inputs())` from the original problem file (numerical
↪ equivalence check)
2. **Subkernel Standalone Execution**: Each subkernel in `subkernel_problems/`
↪ must execute successfully with its own `get_inputs()` and `get_init_inputs()`
↪ (or `get_input_configs()`):
   - `model = Model(*get_init_inputs())` must instantiate without error
   - `output = model(*get_inputs())` must execute without error
3. **Shape Consistency**: The `get_inputs()` in each subkernel file must return
↪ tensors with shapes matching what `ModelNew.forward()` will actually pass to
↪ that submodule
4. **Init Function Consistency**: The `get_init_inputs()` in each subkernel file
↪ must return arguments that `ModelNew` uses to initialize the corresponding
↪ `Model` module
5. **Input Function Coverage**: If `ModelNew` nn.Module at
↪ `decomposed_problem.py` contain multiple instantiation of subkernel PyTorch
↪ modules in `subkernel_problems`, the corresponding subkernel PyTorch module
↪ file must have the function `get_input_configs()` that returns a list of
↪ dicts with functions that return `get_init_inputs()` and `get_inputs()`
↪ shapes present in `ModelNew`'s forward pass
6. **Cheating Detection**: The only PyTorch modules initialized at
↪ `decomposed_problem.py` should be modules from `subkernel_problems/`

FUSION OBJECTIVE
```

```
Maximize fusion opportunities by grouping operations that can be efficiently
 ↪  executed in a single kernel. Only split into separate submodules when fusion
 ↪  is blocked by correctness or hardware constraints.

**Splitting Constraints (only split when these apply):**
- Data dependencies crossing incompatible tiling axes (e.g., reduction reuse
 ↪  across incompatible blocks)
- Operations requiring global synchronization (e.g., full tensor reductions
 ↪  before the result is used)
- Branching control flow that cannot be predicated

**When splitting is necessary:**
- Emit the minimal number of submodules
- Add a comment explaining the blocking constraint (e.g., "reduction requires
 ↪  global synchronization")

## FUSION-FRIENDLY GROUPINGS

**Operations that SHOULD be fused together (single submodule):**
- Matmul/GEMM followed by element-wise ops (e.g., matmul → relu, linear → gelu)
- Chains of element-wise operations (relu, sigmoid, tanh, add, mul, scale)
- Reshape/view/permute operations (free at kernel level)
- Dropout, masking, scaling applied to computation results
- Bias addition after matmul

**Common Fusible Patterns:**
- `Linear + Activation`: nn.Linear → ReLU/GELU/SiLU (single kernel)
- `Attention QKV`: Q@K^T → scale → mask → softmax → @V (may need split at softmax
 ↪  due to reduction)
- `MLP Block`: Linear → Activation → Linear (split between the two linears if
 ↪  needed)
- `Residual Add`: computation + skip connection (fuse the add)

## MEMORY & DATAFLOW OPTIMIZATION

- Treat global memory traffic as the primary cost
- Never materialize intermediates unless unavoidable
- Fuse producers → consumers when data reuse exists
- Keep intermediates in registers/SRAM where possible

## DECOMPOSITION STRATEGY

1. Analyze the forward pass as a dataflow graph
2. Identify operations requiring global synchronization (fusion blockers)
3. Group all other operations between blockers into single submodules
4. Each submodule = one kernel opportunity
5. Ensure exact output semantics, dtype behavior, and broadcasting rules are
 ↪  preserved

## SUBKERNEL INPUT FUNCTION REQUIREMENTS

**CRITICAL DECISION RULE:**
For each subkernel file, count how many times `ModelNew.__init__` instantiates
 ↪  that subkernel's Model class:
- **1 instantiation** → define `get_inputs()` and `get_init_inputs()`
```

```
- **N instantiations (N > 1)** → define `get_input_configs()` returning exactly N
 ↪  dicts

**Function Definitions:**

When Model is instantiated ONCE:
- **get_init_inputs()**: Returns tuple of arguments for
 ↪  `Model(*get_init_inputs())`
- **get_inputs()**: Returns tuple of input tensors for `model(*get_inputs())`

When Model is instantiated MULTIPLE TIMES:
- **get_input_configs()**: Returns a list of N dicts (one per instantiation),
 ↪  each containing:
  - `"get_init_inputs"`: function returning tuple for that instance's
   ↪  initialization
  - `"get_inputs"`: function returning tuple for that instance's forward pass

**Shape Requirements:**
- `get_init_inputs()` must return arguments matching what `ModelNew.__init__`
 ↪  passes when instantiating
- `get_inputs()` must return tensors with shapes matching what
 ↪  `ModelNew.forward()` passes to that submodule

This ensures each subkernel can be independently tested and optimized with
 ↪  realistic input shapes.

## SUBMODULE REUSE REQUIREMENT

**Do NOT create duplicate subkernel files for the same operation type, even if
 ↪  they have different shapes.**
Each subkernel file should define a **shape-parameterized** `Model` class that
 ↪  can be instantiated multiple times with different dimension arguments. A
 ↪  single kernel implementation can handle different tensor shapes through
 ↪  parameterization.

**Rules:**

1. **One file per operation pattern**: If the same fused operation pattern (e.g.,
 ↪  Linear+ReLU, Conv2d+ReLU+MaxPool) appears multiple times in the original
 ↪  model, create ONE subkernel file and instantiate it multiple times in
 ↪  `ModelNew.__init__` with different parameters.
2. **Parameterize by dimensions**: The subkernel `Model.__init__` should accept
 ↪  dimension arguments (e.g., `in_features`, `out_features`, `in_channels`,
 ↪  `out_channels`) that configure the operation for different shapes.
3. **Identify operation patterns, not instances**: When analyzing the forward
 ↪  pass, group by operation TYPE (e.g., "Linear+ReLU", "Conv2d+ReLU"), not by
 ↪  layer index or specific dimension values.
4. **get_inputs() shows ONE example shape**: The `get_inputs()` function should
 ↪  return tensors for one valid input configuration.
5. **get_input_configs() shows MULTIPLE example shapes**: If the same fused
 ↪  operation pattern appears multiple times in the original model, the file
 ↪  corresponding to the pattern must have  `get_input_configs()` function that
 ↪  returns a list of dictionaries with functions `get_inputs` and
 ↪  `get_init_inputs` for all shapes
```

```
**EXAMPLE - Applying the Reuse Principle:**

Consider a 4-layer MLP: `Linear(784,256) → GELU → Linear(256,128) → GELU →
 ↪  Linear(128,64) → GELU → Linear(64,10)`
This model has the pattern `[Linear+GELU] × 3` followed by `[Linear] × 1`.

**WRONG approach (creates duplicate files for the same operation):**
```
subkernel_problems/
    linear_gelu_1.py # Linear(784, 256) + GELU
    linear_gelu_2.py # Linear(256, 128) + GELU ← DUPLICATE PATTERN!
    linear_gelu_3.py # Linear(128, 64) + GELU   ← DUPLICATE PATTERN!
    linear_out.py   # Linear(64, 10)
```

**CORRECT approach (reuses the same class for identical operation patterns):**
```
subkernel_problems/
    linear_gelu.py   # Model(in_features, out_features) - ONE file, instantiated
     ↪ 3 times
    linear.py   # Model(in_features, out_features) - for final layer
     ↪ without activation
```

**Apply this principle to ANY model you decompose**: identify the unique
 ↪  operation patterns, create one subkernel file per pattern, and instantiate
 ↪  each pattern as many times as needed with appropriate dimension parameters.

## FUSION EXAMPLES

**Example 1: GELU + GEMM**
Input: `gelu(A) @ B`
→ Single submodule: GELU and matmul can be fused (gelu feeds directly into
 ↪  matmul)

**Example 2: Two-layer MLP**
Input: `relu(linear1(x)) → linear2`
→ Two submodules:
  - Submodule 1: linear1 + relu (fused)
  - Submodule 2: linear2
  - Reason: Output of linear1+relu must be materialized before linear2

**Example 3: Attention Block**
Input: `softmax(Q @ K^T / sqrt(d)) @ V`
→ Depends on implementation:
  - If online softmax possible: single submodule
  - If standard softmax: split at softmax (requires full row before
   ↪  normalization)

## PROBLEM TO DECOMPOSE
```python
{{ problem_descriptor }}
```
```

H.2. Subkernel Generation

```
**TASK: Generate a complete kernel implementation for the problem described in
 ↪  the "PYTORCH TO KERNEL PROBLEM" section.**


## PYTORCH TO KERNEL PROBLEM

You are given the following Triton kernel writing problem that contains
1. PyTorch module named `Model` that extends `torch.nn.Module`
2. Function that returns a sample argument for the module's `__init__` method
 ↪  `get_init_inputs`
3. Function that returns a sample argument for the module's `forward` method
 ↪  `get_inputs`


{{ problem_descriptor }}


Your task is to produce a PyTorch module named `ModelNew` that is a Triton
 ↪  implementation of the `Model` module satisfying the following requirements.
1. The `ModelNew` module contains `__init__` and `forward` methods with the same
 ↪  input and output signatures as the `Model` module
2. The `ModelNew` module's `forward` method must
  - Correctly and efficiently implement the `Model` module's `forward` method in
   ↪  Triton
  - Invoke a separately written Triton kernel function decorated with @triton.jit
    - The Triton kernel can be named anything (e.g., _kernel)
    - The Triton kernel function must explicitly accept all necessary tensors as
     ↪  function inputs rather than internally initializing random tensors
  - Handle grid calculation and kernel launch within the method, outside the
   ↪  Triton kernel function
  - Return contiguous tensors if returning tensors
3. If the `Model`'s `__init__` method implicitly initializes parameters/tensors,
 ↪  explicitly initialize these values in the exact same order in the `__init__`
 ↪  method
  - Example 1: `Model` contains `nn.Linear`/`nn.Conv3D` and its weights and bias
   ↪  values need to be explicitly passed to the Triton kernel function
    ```python
    # This is how nn.Linear initializes its weights and bias
    from torch.nn import init

    weight = <INITIALIZE APPROPRIATELY SHAPED EMPTY WEIGHT TENSOR ON THE CPU>
    init.kaiming_uniform_(weight, a=5**0.5)
    bias = <INITIALIZE APPROPRIATELY SHAPED EMPTY BIAS TENSOR ON THE CPU>
    fan_in, _ = init._calculate_fan_in_and_fan_out(weight)
    bound = 1 / fan_in**0.5 if fan_in > 0 else 0
    init.uniform_(bias, -bound, bound)
    ```
  - Example 2: `Model` contains `nn.MultiheadAttention` and its various
   ↪ parameters need to be explicitly passed to the Triton kernel function
    ```python
    from torch.nn import init

    <ASSUME APPROPRIATELY SHAPED TENSORS AND BOOLEAN FLAGS ARE INITIALIZED>
```
```

```
    if _qkv_same_embed_dim:
        init.xavier_uniform_(in_proj_weight)
    else:
        init.xavier_uniform_(q_proj_weight)
        init.xavier_uniform_(k_proj_weight)
        init.xavier_uniform_(v_proj_weight)

    if in_proj_bias is not None:
        init.constant_(in_proj_bias, 0.0)
        init.constant_(out_proj.bias, 0.0)
    if bias_k is not None:
        init.xavier_normal_(bias_k)
    if bias_v is not None:
        init.xavier_normal_(bias_v)
    ```
4. The `ModelNew` module must not cheat by initializing torch.nn module or trying
 ↪ to replicate `Model`'s forward pass using PyTorch operations or shortcuts
5. Do not repeat the raw original `Model` code anywhere, even in the comments
6. Keep logic in the `forward` method and the kernel minimal; speed is important
 ↪ and the entire `forward` method execution time is measured
7. Assume that the modules will be in PyTorch `eval` mode (this is highly
 ↪ relevant for BatchNorm, etc)
{% if assert_single_launch %}
8. The `ModelNew` module must only launch exactly ONE fused Triton kernel
 ↪ (multiple launches of the same kernel allowed) that performs all meaningful
 ↪ computation of the module
{% endif %}

{% if show_example -%}
Example solution format for Triton persistent matmul kernel (wrapper can use
 ↪ PyTorch, but kernel must purely use Triton):
```python
import torch
import triton
import triton.language as tl

@triton.jit
def _compute_pid(tile_id, num_pid_in_group, num_pid_m, GROUP_SIZE_M, NUM_SMS):
    group_id = tile_id // num_pid_in_group
    first_pid_m = group_id * GROUP_SIZE_M
    group_size_m = min(num_pid_m - first_pid_m, GROUP_SIZE_M)
    pid_m = first_pid_m + (tile_id % group_size_m)
    pid_n = (tile_id % num_pid_in_group) // group_size_m
    return pid_m, pid_n


@triton.autotune(
    configs=matmul_get_configs(),
    key=["M", "N", "K"],
)
@triton.jit(launch_metadata=_matmul_launch_metadata)
def matmul_kernel_persistent(a_ptr, b_ptr, c_ptr,
                             M, N, K,
                             stride_am, stride_ak,
                             stride_bk, stride_bn,
```

```
                        stride_cm, stride_cn,
                        BLOCK_SIZE_M: tl.constexpr,
                        BLOCK_SIZE_N: tl.constexpr,
                        BLOCK_SIZE_K: tl.constexpr,
                        GROUP_SIZE_M: tl.constexpr,
                        NUM_SMS: tl.constexpr,
                        ):
start_pid = tl.program_id(axis=0)
num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)
k_tiles = tl.cdiv(K, BLOCK_SIZE_K)
num_tiles = num_pid_m * num_pid_n

# NOTE: There is currently a bug in blackwell pipelining that means it can't
 ↪  handle a value being
# used in both the prologue and epilogue, so we duplicate the counters as a
 ↪  work-around.
tile_id_c = start_pid - NUM_SMS

offs_k_for_mask = tl.arange(0, BLOCK_SIZE_K)
num_pid_in_group = GROUP_SIZE_M * num_pid_n

for tile_id in tl.range(start_pid, num_tiles, NUM_SMS, flatten=True):
    pid_m, pid_n = _compute_pid(tile_id, num_pid_in_group, num_pid_m,
     ↪  GROUP_SIZE_M, NUM_SMS)
    start_m = pid_m * BLOCK_SIZE_M
    start_n = pid_n * BLOCK_SIZE_N
    offs_am = start_m + tl.arange(0, BLOCK_SIZE_M)
    offs_bn = start_n + tl.arange(0, BLOCK_SIZE_N)
    offs_am = tl.where(offs_am < M, offs_am, 0)
    offs_bn = tl.where(offs_bn < N, offs_bn, 0)
    offs_am = tl.max_contiguous(tl.multiple_of(offs_am, BLOCK_SIZE_M),
     ↪  BLOCK_SIZE_M)
    offs_bn = tl.max_contiguous(tl.multiple_of(offs_bn, BLOCK_SIZE_N),
     ↪  BLOCK_SIZE_N)

    accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
    for ki in range(k_tiles):
        offs_k = ki * BLOCK_SIZE_K + tl.arange(0, BLOCK_SIZE_K)
        a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] *
         ↪  stride_ak)
        b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] *
         ↪  stride_bn)

        a = tl.load(a_ptrs, mask=offs_k_for_mask[None, :] < K - ki *
         ↪  BLOCK_SIZE_K, other=0.0)
        b = tl.load(b_ptrs, mask=offs_k_for_mask[:, None] < K - ki *
         ↪  BLOCK_SIZE_K, other=0.0)
        accumulator = tl.dot(a, b, accumulator)

    tile_id_c += NUM_SMS
    pid_m, pid_n = _compute_pid(tile_id_c, num_pid_in_group, num_pid_m,
     ↪  GROUP_SIZE_M, NUM_SMS)
    offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
```

```
        c_ptrs = c_ptr + stride_cm * offs_cm[:, None] + stride_cn * offs_cn[None,
        ↪  :]
        c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)
        if (c_ptr.dtype.element_ty == tl.float8e4nv):
            c = accumulator.to(tl.float8e4nv)
        else:
            c = accumulator.to(tl.float32)
        tl.store(c_ptrs, c, mask=c_mask)


class ModelNew(torch.nn.Module):
    def __init__(self, *args, **kwargs):
        """Any necessary module initialization logic."""
        ...

    def forward(self, a, b):
        """Wrapper function that handles kernel launch."""
        assert a.shape[1] == b.shape[0], "Incompatible dimensions"
        assert a.dtype == b.dtype, "Incompatible dtypes"
        NUM_SMS = torch.cuda.get_device_properties("cuda").multi_processor_count
        M, K = a.shape
        K, N = b.shape
        dtype = a.dtype
        # Allocates output. NOTE: PyTorch operations allowed here only for setup
        c = torch.empty((M, N), device=a.device, dtype=dtype)
        # 1D launch kernel where each block gets its own program.
        grid = lambda META: (min(NUM_SMS, triton.cdiv(M, META["BLOCK_SIZE_M"]) *
        ↪  triton.cdiv(N, META["BLOCK_SIZE_N"])), )
        matmul_kernel_persistent[grid](
            a, b, c,
            M, N, K,
            a.stride(0), a.stride(1),
            b.stride(0), b.stride(1),
            c.stride(0), c.stride(1),
            NUM_SMS=NUM_SMS,
        )
        return c
```
{%- endif %}

Your solution will be assessed in two ways.
1. Correctness: Are the outputs of `Model` and `ModelNew`'s `forward` methods
↪ sufficiently close?
2. Speedup: How fast is the `ModelNew`'s Triton-based `forward` method compared
↪ to `Model`'s PyTorch-based `forward` method?

General Triton Guidelines (Helpful):

1. KERNEL STRUCTURE:
  - Use @triton.jit decorator for kernel functions
  - Use tl.constexpr for compile-time constants (BLOCK_SIZE, etc.)
  - Include proper type hints and launch metadata when needed

2. MEMORY ACCESS PATTERNS:
  - Use tl.load and tl.store with proper masking

```
  - Coalesce memory accesses for optimal performance
  - Use tensor descriptors for advanced memory operations (TMA)
  - Handle boundary conditions with masks

3. INDEXING AND GRID:
  - Use tl.program_id(axis) for block indices
  - Calculate offsets: pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)
  - Use tl.cdiv for ceiling division
  - Always mask for out-of-bounds protection

4. OPTIMIZATION TECHNIQUES:
  - Use @triton.autotune for automatic configuration selection (Autotuned
   ↪  parameters must be declared as tl.constexpr meta-parameters and must not be
   ↪  passed as runtime kernel arguments)
  - Choose appropriate BLOCK_SIZE (powers of 2: 64, 128, 256, 512, 1024)
  - Leverage tensor cores with tl.dot for matrix operations
  - Use warp specialization for better scheduling
  - Consider epilogue subtiling to reduce shared memory usage
  - Aggressively fuse compatible operator stages to minimize memory traffic and
   ↪  kernel launch overhead; only keep stages separate if fusion is infeasible

5. COMMON PATTERNS:
  a) Elementwise operations: Load -> Compute -> Store
  b) Reductions: Use tl.reduce with proper axis (can also use tl.sum(), tl.max(),
   ↪  tl.min(), etc)
  c) Matrix multiplication: Use tl.dot with accumulator
  d) Softmax: Online normalization for numerical stability
  e) Fused operations: Combine multiple ops in single kernel and document the
   ↪  fused stages for reviewers
    - BN inside fused conv:
      - bn_mean_f32 = tl.load(mean_ptr + oc).to(tl.float32)
      - bn_var_f32 = tl.load(var_ptr + oc).to(tl.float32)
      - gamma_f32 = tl.load(gamma_ptr + oc).to(tl.float32)
      - beta_f32  = tl.load(beta_ptr + oc).to(tl.float32)
      - acc_f32   = acc.to(tl.float32)  # conv accumulator
      - x_norm    = (acc_f32 - bn_mean_f32) / tl.sqrt(bn_var_f32 + eps)
      - out_f32   = x_norm * gamma_f32 + beta_f32
      - tl.store(out_ptr + out_idx, out_f32.to(out_ptr.dtype.element_ty),
       ↪  mask=mask)
    - LayerNorm per row:
      - mean_f32 = tl.zeros((), dtype=tl.float32)
      - var_f32  = tl.zeros((), dtype=tl.float32)
      - accumulate in fp32; use tl.sqrt(var_f32 + eps); cast on store.

6. ADVANCED FEATURES:
  - Persistent kernels for better SM utilization
  - Tensor Memory Accelerator (TMA) descriptors
  - Multi-stage pipelines with num_stages
  - Warp specialization with warp_specialize parameter

7. RUNTIME CONSTRAINTS:
  - Wrappers: validate/allocate/launch only; no math
```

```
  - All compute runs in Triton kernels; no torch.nn, torch.nn.functional (e.g.,
   ↪  F.*), or other PyTorch compute ops, including general tensor-tensor math
   ↪  like torch.matmul/mm/bmm/einsum or their Tensor method forms, and no
   ↪  low-level torch.ops.aten.* calls (e.g., torch.ops.aten.conv2d,
   ↪  torch.ops.aten.layer_norm, torch.ops.aten.addmm, torch.ops.aten.mean)
  - PyTorch allowed only for allocation, dtype/device checks, and packaging
   ↪  results

8. TRITON MATH DOCS (from `triton.language` module or `tl`):
  - The following is a comprehensive list of math related `tl` ops; if a math op
   ↪  in `tl` is not in the following list, it does not exist
    - LinAlg Ops: tl.dot, tl.dot_scaled
    - Math Ops: tl.abs, tl.cdiv, tl.ceil, tl.clamp, tl.cos, tl.div_rn, tl.erf,
     ↪  tl.exp, tl.exp2, tl.fdiv, tl.floor, tl.fma, tl.log, tl.log2, tl.maximum,
     ↪  tl.minimum, tl.rsqrt, tl.sigmoid, tl.sin, tl.softmax, tl.sqrt,
     ↪  tl.sqrt_rn, tl.umulhi
    - Reduction Ops: tl.argmax, tl.argmin, tl.max, tl.min, tl.reduce, tl.sum,
     ↪  tl.xor_sum

## Basic Hardware Information

The Triton kernel should be optimized for NVIDIA A100 GPU with 80GB RAM and
 ↪  Ampere architecture.

GPU Details
- SMs: 108 (SXM) / 80 (PCIe)
- Warp size: 32 threads
- Max threads / block 1024 → `num_warps <= 32`
- Registers / SM: 256k 32-bit → large tiles easily become register-bound
- Shared memory / SM: up to 164 KB (configurable)
- L2 cache* ~40 MB (huge → favors block reuse / grouping)
- Tensor Cores**: FP16, BF16, TF32 (TF32 enabled by default in cuBLAS)

Important Suggestions
- Always set `tl.dot`'s `allow_tf32` flag to False; otherwise correctness tests
 ↪  will fail

**RUNTIME RESTRICTIONS:**
- The Python wrapper may only perform argument validation, tensor allocation, and
 ↪  launch configuration
- All math (convs, activations, pooling, reductions, etc.) must reside inside
 ↪  kernels
- Never import or instantiate `torch.nn` modules within the main kernel function,
 ↪  call `torch.nn.functional` (including aliases like `F.*`), or rely on PyTorch
 ↪  helpers such as `torch.conv*`, `torch.relu`, `torch.max_pool*`, etc
- PyTorch usage is limited to allocation helpers, dtype/device checks, and
 ↪  assertions needed to launch the main kernel function

**STRICTLY FORBIDDEN - DO NOT CHEAT:**
- DO NOT call PyTorch functions (torch.add, torch.mul, torch.sum, torch.matmul,
 ↪  torch.mm, torch.bmm, torch.einsum, etc.) inside the kernel computation
- DO NOT use PyTorch operations to perform the actual computation and just return
 ↪  the result (this includes tensor-tensor ops such as x.matmul(y), x.bmm(y),
 ↪  x.mm(y), or x.einsum(...))
```

```
- DO NOT call low-level operator APIs like torch.ops.aten.* (e.g.,
 ↪  torch.ops.aten.conv2d, torch.ops.aten.addmm, torch.ops.aten.layer_norm,
 ↪  torch.ops.aten.mean); these are PyTorch compute ops and using them is
 ↪  considered cheating
- DO NOT implement the logic using pure PyTorch and avoid writing kernel code
- DO NOT hide PyTorch compute (including torch.ops.aten.* calls or
 ↪  torch.nn/torch.nn.functional usage) inside helper functions that are then
 ↪  called from the wrapper
- The wrapper class/method can use PyTorch for tensor creation, memory
 ↪  allocation, and result formatting, but the core computation MUST happen in
 ↪  the main kernel function
- DO NOT import or instantiate torch.nn modules, call torch.nn.functional
 ↪  (including aliases like F.*), or use PyTorch activations/pooling helpers to
 ↪  satisfy the requirements


## FINAL CHECKLIST

**Runtime Constraints (CRITICAL: Evaluation will FAIL if these are not
 ↪  satisfied)**

* [ ] Wrapper PyTorch module `ModelNew` that validates, allocates, and launches
 ↪  kernel function(s) (no math) exists in the code
* [ ] No other PyTorch module class definitions exists in the code, including the
 ↪  problem PyTorch module `Model`, exists in the code
* [ ] Kernel function(s) use @triton.jit decorator
* [ ] All compute runs in Triton kernels
* [ ] No torch.nn, torch.nn.functional (F.*),torch.ops.aten.*,
 ↪  torch.matmul/mm/bmm/einsum (and their Tensor method equivalents), or PyTorch
 ↪  compute ops
* [ ] PyTorch used only for allocation, dtype/device checks, and packaging
 ↪  results
* [ ] Explicitly reason about what is being fused and not fused inside the
 ↪  wrapper docstring

**Kernel Structure**

* [ ] Compile-time constants use tl.constexpr (BLOCK_SIZE, etc.)
* [ ] Proper type hints and launch metadata included
* [ ] Kernel is self-contained and stateless

**Memory Access**

* [ ] tl.load and tl.store used with proper masking
* [ ] Memory accesses are coalesced (inner dimension maps to adjacent threads)
* [ ] Boundary conditions handled with masks
* [ ] Invariants loaded once and reused from registers

**Indexing & Grid**

* [ ] tl.program_id(axis) used for block indices
* [ ] Offsets calculated as: pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)
* [ ] Out-of-bounds protection with masks

**Parallelism & Launch**
```

```
* [ ] BLOCK_SIZE is power of 2 (64, 128, 256, 512, 1024)
* [ ] @triton.autotune used for automatic configuration selection

Fill out the following solution template and present it as a single Python code
 ↪  block:

```python
import triton
import triton.language as tl
import torch.nn as nn

# TODO: Write kernel function(s) wrapped in @triton.jit here


# TODO: Invoke the kernel function(s) from above *inside a PyTorch module named
 ↪  ModelNew* to solve the problem
class ModelNew(nn.Module):
    ...
```
```