LLM-Driven Triton Kernel Generation with Iterative Compiler Feedback

Hannah Shu
Computer Science and Engineering
University of Michigan
Ann Arbor, MI
hyshu@umich.edu

Nikhil Gunaratnam
Computer Science and Engineering
University of Michigan
Ann Arbor, MI
nikhilg@umich.edu

Pranav Rakasi
Computer Science and Engineering
University of Michigan
Ann Arbor, MI
prakasi@umich.edu

***Abstract*—Hand-tuned GPU kernels remain critical for high-performance deep learning systems, yet they are costly to develop and maintain, while existing autotuning approaches are often slow, brittle, and limited to predefined search spaces. We explore an alternative approach that leverages LLMs to generate Triton GPU kernels directly, assisted by a compiler-in-the-loop feedback mechanism to iteratively improve the outputted kernels. Individual operation kernels have largely already been hand-tuned and optimized to the max, so our project focuses on fused operations where traditional libraries have limited optimization flexibility and where LLM-driven code synthesis can introduce novel kernel structures. We evaluate this approach by measuring compilation success rates, functional correctness, and runtime performance across a range of fused workloads. Our results demonstrate that this approach of fusing the operations achieves speedup depending on the tensor size and operation complexity, with speedup emerging for larger tensors such that the computation volume exceeds the overhead of launching the kernel.**

***Index Terms*—GPU kernels, Triton, large language models, code generation, kernel fusion, autotuning**

I. Introduction

Deep learning workloads demand high-performance GPU kernels to achieve acceptable training and inference speeds. While frameworks like PyTorch provide optimized implementations for common operations through libraries such as cuDNN, achieving peak performance often requires hand-tuned kernels tailored to specific model architectures, tensor shapes, and hardware configurations. Writing such kernels traditionally requires deep expertise in GPU architecture, memory hierarchies, and parallel programming paradigms - skills that remain scarce even among experienced software engineers.

The emergence of Triton [1] has lowered the barrier to GPU kernel development by providing a Python-based domain-specific language that abstracts away many low-level details such as shared memory management, thread synchronization, and memory coalescing. However, writing efficient Triton kernels still requires understanding of tiling strategies, memory access patterns, and hardware-specific optimizations. Moreover, optimal kernel configurations vary significantly across different GPU architectures, tensor dimensions, and runtime contexts, making static optimization insufficient for the diverse deployment scenarios encountered in modern machine learning systems.

Traditional autotuning frameworks address this variability by searching over large configuration spaces to find optimal parameters for specific hardware and workload combinations. However, these approaches suffer from three fundamental limitations: they are slow to adapt when conditions change, expensive to rerun for new configurations, and brittle when faced with workloads outside their training distribution. As models evolve rapidly and deployment targets diversify, the need for more adaptive optimization strategies has become increasingly apparent.

Recent advances in large language models (LLMs) have demonstrated remarkable capabilities in code generation tasks, raising the question of whether these models can be leveraged for automated GPU kernel development. Rather than searching through predefined configuration spaces, LLMs could potentially generate novel kernel implementations on demand, adapting to specific requirements through natural language specifications. However, GPU kernel generation presents unique challenges that distinguish it from general-purpose code generation. Kernels must not only be syntactically correct but also produce numerically accurate results while achieving competitive performance - requirements that demand both semantic understanding and hardware awareness.

In this work, we explore the feasibility of using LLMs to generate Triton GPU kernels, with a focus on fused operations where the creative potential of LLMs is most beneficial. We make three key contributions:

First, we present a systematic study of LLM reliability for Triton kernel generation, examining compilation success rates, numerical correctness, and performance characteristics across multiple state-of-the-art models including GPT-4o-mini, Llama 3.3 70B, and Llama 3.1 8B.

Second, we develop an iterative feedback system that provides compiler error messages and runtime diagnostics back to the LLM, enabling automatic refinement of generated kernels. Our experiments demonstrate that this feedback loop significantly improves both compilation success and numerical correctness compared to single-shot generation.

Third, we identify fused operations - such as bias addition combined with activation functions, or normalization combined with elementwise transforms - as a particularly promising target for LLM-based kernel generation. Unlike single operations where PyTorch and cuDNN already provide highly optimized implementations, fused operations offer opportunities for novel implementations that reduce kernel launch overhead and improve data locality.

Our results indicate that while LLMs are not yet reliable kernel generators without iteration, the combination of LLM generation with compiler-guided feedback represents a viable path toward more adaptive GPU optimization. We discuss the implications of these findings for future systems that could leverage LLMs to generate optimized kernels on demand, adapting to changing hardware and workload characteristics without the overhead of traditional autotuning.

II. Background and Motivation

A. GPU Kernel Optimization Challenges

Modern GPUs achieve high throughput through massive parallelism, but realizing this potential requires careful attention to the architectural characteristics of the hardware. The memory hierarchy of a typical GPU includes off-chip DRAM (global memory) with high bandwidth but significant latency, on-chip shared memory (SRAM) within each streaming multiprocessor that provides fast access but limited capacity, and registers that offer the fastest access but are scarce resources. Efficient GPU code must orchestrate data movement across these levels while keeping thousands of threads productively occupied.

Writing GPU kernels requires managing multiple interrelated concerns simultaneously. Memory accesses must be coalesced to maximize bandwidth utilization, with threads in a warp accessing contiguous memory addresses. Shared memory must be used judiciously to cache frequently accessed data while avoiding bank conflicts that serialize access. Thread block sizes and grid dimensions must be chosen to balance occupancy against register pressure. These optimizations interact in complex ways: a change that improves memory access patterns may increase register usage, potentially reducing occupancy and overall throughput.

Furthermore, optimal configurations are highly dependent on context. The same logical operation may require different implementations depending on tensor shapes, data types, batch sizes, and the specific GPU architecture being targeted. A kernel tuned for large batch inference on an A100 may perform poorly for small batch training on a consumer GPU. This variability makes static optimization insufficient - what works well in one setting may be suboptimal or even counterproductive in another.

B. The Triton Programming Model

Triton addresses some of these challenges by providing a higher-level abstraction for GPU programming. Rather than managing individual threads, Triton programmers work with blocks of data, and the compiler automatically handles shared memory allocation, thread synchronization, and many memory access optimizations. A kernel that would require hundreds of lines of CUDA can often be expressed in tens of lines of Triton, making GPU programming accessible to a broader audience.

The Triton compiler translates Python-like kernel code through multiple intermediate representations, applying optimizations at each stage before generating PTX instructions for NVIDIA GPUs or equivalent code for other platforms. This compilation process includes automatic tiling, memory coalescing, and instruction scheduling. However, the programmer must still make key algorithmic decisions: how to partition work across thread blocks, what tile sizes to use, and how to structure the computation to expose parallelism.

Despite its advantages, Triton introduces its own challenges. The abstraction sometimes obscures performance-critical details, making it difficult to reason about why a particular implementation is slow. The API evolves rapidly, and code written for one version may not compile with another. Perhaps most significantly for our purposes, generating correct Triton code requires understanding both the Python-level syntax and the underlying execution model—a combination that proves challenging for current LLMs.

C. Limitations of Existing Autotuning Approaches

Traditional autotuning frameworks approach kernel optimization as a search problem over a predefined configuration space. Given a parameterized kernel template, these systems evaluate many configurations to find one that performs well on the target hardware. While effective in many scenarios, this approach has significant limitations.

The search process is inherently slow. Evaluating each configuration requires compiling the kernel and running it on actual hardware, with each evaluation taking milliseconds to seconds. For large configuration spaces, the search may require hours or days to converge, making it impractical to retune whenever conditions change.

Autotuning is also expensive in terms of computational resources. The search process consumes GPU cycles that could otherwise be used for productive work. In cloud environments where GPU time is billed by the hour, extensive autotuning can represent a significant cost.

Perhaps most critically, traditional autotuning is brittle. The tuned configurations are specific to the exact conditions under which tuning was performed. When tensor shapes change, when the model is deployed on different hardware, or when the surrounding code is modified in ways that affect caching behavior, the tuned configuration may no longer be optimal. This brittleness limits the practical value of autotuning in dynamic environments where conditions change frequently.

D. The Case for Fused Operations

Our initial experiments focused on generating kernels for individual operations such as vector addition and softmax. However, we quickly discovered that this approach faces a fundamental challenge: PyTorch and cuDNN already provide highly optimized implementations for common single operations. An LLM-generated kernel for matrix multiplication is unlikely to outperform cuBLAS, which represents decades of hand-tuning by expert engineers with intimate knowledge of GPU architecture.

The real opportunity for LLM-based kernel generation lies in fused operations—combinations of multiple operations executed in a single kernel. Consider a common pattern in neural networks: applying a linear transformation followed by adding a bias and then applying an activation function. In a naive implementation, each operation launches a separate kernel, reading inputs from and writing outputs to global memory. A fused implementation performs all three operations in a single kernel, keeping intermediate results in registers or shared memory and avoiding redundant memory traffic.

Fused operations offer several advantages as targets for LLM generation. First, the space of possible fusions is combinatorially large, making it impractical to hand-optimize every combination that might be useful. Second, the optimal fusion strategy depends on context—fusing operations that are memory-bound may help, while fusing compute-bound operations may not. Third, fusion requires understanding the semantics of each operation well enough to combine them correctly, which is precisely the kind of reasoning that LLMs excel at.

Common fusion opportunities in deep learning include combining bias addition with activation functions (ReLU, GELU, etc.), merging layer normalization with subsequent elementwise operations, fusing the query-key-value projections in transformer attention with the subsequent softmax computation, and combining multiple pointwise operations that would otherwise require separate kernel launches. These fusions can yield significant speedups by reducing memory bandwidth requirements and kernel launch overhead.

III. System Design

A. System Architecture Overview

Our system employs an AI-driven pipeline to automatically generate, validate, and benchmark optimized Triton kernels for tensor operations. The architecture consists of seven interconnected components operating through an iterative feedback loop.

Environment Setup: The system initializes the execution environment by installing required packages (PyTorch, Triton, API clients), verifying GPU availability for kernel execution, and configuring API credentials for remote code generation.

Model Loading: Instead of loading models locally, the system uses API-based code generation via cloud-hosted LLMs (e.g., OpenAI GPT-4, HuggingFace Inference API). This approach eliminates the need for local GPU memory during generation and enables code generation on any machine with internet access.

Prompt Engineering: The system constructs structured prompts that specify the target operation (e.g., RMS Normalization + Bias fusion), include Triton-specific syntax guidelines, and incorporate conversation history from previous generation attempts.

Code Generation: Triton kernel code is generated through API calls to remote LLMs. The generated code is automatically extracted from markdown blocks and cleaned, with retry logic handling transient API failures.

Code Execution: Generated kernels are executed within a safe wrapper that isolates execution, captures errors, and validates outputs against reference implementations. Test data is automatically generated to match expected input shapes and dtypes.

Performance Benchmark: The system measures kernel performance using CUDA events, comparing Triton implementations against PyTorch baselines across multiple tensor shapes, dtypes, and operation variants. Metrics include latency (mean, p50, p95), memory usage, and speedup ratios.

Correctness Analysis: Outputs are validated through numerical comparison with PyTorch references using appropriate tolerances, shape validation, and operation-specific semantic checks (e.g., softmax normalization properties).

Feedback Loop: The system implements an iterative refinement process: (1) generate code from prompt, (2) execute and capture errors, (3) construct error-specific feedback prompts with targeted guidance, (4) regenerate code with feedback, repeating up to 15 iterations until successful execution and correctness validation. This loop automatically fixes common issues like pointer type conversions, dtype mismatches, and syntax errors without manual intervention.

B. Code Generation Pipeline

The system employs API-based code generation through cloud-hosted large language models, primarily using OpenAI GPT-4 and HuggingFace Inference API. This approach eliminates the need for local model deployment and enables code generation on any machine with internet access. API calls are wrapped with robust retry logic featuring exponential backoff to handle rate limits and transient network failures. The system adaptively adjusts generation parameters such as temperature and maximum token count based on error repetition patterns, increasing creativity when the same errors persist across iterations.

Prompt engineering follows a structured multi-turn conversation format. The system message establishes the model's role as an expert GPU programmer specializing in Triton kernel optimization, with explicit emphasis on understanding pointer types, dtype conversions, and Triton compilation requirements. Conversation history is selectively incorporated, including only the last two generation attempts to maintain context within token limits. Error messages are prioritized over full code snippets in the history, ensuring the model focuses on critical failure points. When errors occur, structured feedback prompts are constructed with targeted guidance specific to the error type. For instance, pointer type errors trigger explicit instructions to convert tensors to float32 or float16 before calling `.data_ptr()` in launcher functions. These feedback prompts include the error type, truncated error message, relevant code snippets from the failed attempt, and iteration context, followed by the original task specification to maintain operational context.

Generated responses are processed through extraction and cleaning stages. Code extraction uses regular expressions to locate Python code within markdown-formatted responses. The system first searches for code blocks marked with the Python language specifier (...“‘‘), falling back to generic code blocks (“‘‘...“‘‘) if none are found. When multiple blocks exist, the longest is selected as the complete implementation. For responses without code block delimiters, extraction identifies code by detecting lines starting with Python keywords like import statements or Triton decorators. The cleaning stage fixes common generation artifacts: truncated imports are corrected (e.g., ”t torch” → ”import torch”), missing required imports are added (torch, triton, triton.language), and markdown formatting is removed to produce executable code.

C. Iterative Feedback Loop

The feedback loop implements iterative refinement through up to 15 iterations, automatically correcting errors in generated Triton kernels. Each iteration is informed by execution results, error analysis, and conversation history.

Compiler output from Triton provides the primary error signals. Execution captures Triton compilation errors, Python runtime exceptions, and tracebacks, which are classified into categories: SyntaxError, CompilationError (pointer type mismatches, unsupported operations), AttributeError (missing Triton APIs like tl.tanh), TypeError, RuntimeError, and CorrectnessError (numerical mismatches). Error messages are truncated to 300 characters to keep feedback concise and focused.

Performance metrics are not used for error correction, but execution results include correctness validation through numerical comparison with PyTorch references. The system verifies output shapes match expected dimensions and values are within acceptable tolerances using torch.allclose(). These correctness checks determine success.

Conversation history maintains context across iterations, storing each attempt's code, error information, and metadata. However, the simplified feedback approach includes only the last two attempts in prompts to balance context with token efficiency. The history enables pattern recognition of repeated failures, triggering adaptive parameter adjustments (increased temperature and token limits) to encourage more creative solutions.

Error analysis and feedback generation use a simplified approach that prioritizes brevity and clarity. Rather than including full code snippets and extensive error details, the feedback prompt consists of the original clean task specification, a short error summary (truncated to 300 characters), and minimal targeted guidance only for specific error types. For correctness errors, additional guidance is provided about common causes (wrong RMS calculation, incorrect strides, missing masks, missing scale multiplication). For other error types, the feedback simply presents the error type and message, allowing GPT-4 to analyze and fix the issue without overwhelming detail. This simplified approach reduces token usage and focuses the model on the essential error information.

Proactive code fixes are applied automatically before execution, even without prior errors. The system scans for known problematic patterns (non-existent Triton APIs like tl.tanh, tl.libdevice, tl.select). Direct transformations are applied: problematic activations are replaced with ReLU, pointer issues are fixed by modifying launcher functions to pass tensors directly, and kernel launch syntax is corrected. These fixes run on every iteration, preventing common errors from reaching execution. When the same error type appears multiple times in history, corresponding direct fixes are applied, reducing iterations needed.

Success criteria terminate the loop when the execution result indicates success, requiring code execution without exceptions, correct output shapes, and numerical correctness validation. The system returns successful code with iteration count and conversation history. If the maximum limit of 15 iterations is reached without success, it returns failure status with the last error encountered. The success check occurs immediately after execution, allowing early exit to minimize API calls.

D. Triton Kernel Implementation

The generated Triton kernels implement fused tensor operations (vector addition with activation, RMS normalization, softmax) optimized for GPU execution. Operations are fused into single kernels to minimize memory traffic, reading each input element once and writing results directly without intermediate global memory writes.

Kernels handle arbitrary tensor sizes through dynamic stride computation (supporting both 3D [B, L, H] and 2D [B*L, H] tensors), 2D grid layouts when hidden dimensions exceed block size (typically 256 elements per block), and masking for boundary cases where dimensions are not divisible by block size. Masking uses col_id ¡ H conditions applied to both load and store operations, ensuring correct behavior without padding. Stride-aware pointer arithmetic (offset = row_id * stride + col_id) maintains coalesced memory access patterns.

Testing validates kernels across multiple tensor shapes and data types (float32, float16). For each configuration, the system executes both Triton kernels and PyTorch reference implementations, comparing results using torch.allclose() with dtype-specific tolerances (float32: rtol=1e-4, atol=1e-5; float16: rtol=1e-2, atol=1e-3). The framework verifies shape correctness, numerical accuracy, and captures compilation errors, runtime exceptions, and correctness failures for feedback loop processing.

IV. Experimental Setup

A. Hardware and Environment

Experiments were conducted on Google Colab using T4 and A100 GPUs (A100 available with Colab Pro+). Code generation uses API-based cloud services (OpenAI GPT-4), requiring no local GPU memory, while kernel execution and benchmarking require GPU access for Triton compilation and CUDA operations. The environment uses PyTorch with CUDA support, Triton for kernel compilation, and standard Python libraries. GPU verification checks CUDA availability and reports device name and memory capacity before execution, ensuring kernels run on appropriate hardware for performance measurements.

B. LLM Models Evaluated

We evaluated multiple models across different deployment strategies, evolving from local execution to API-based generation. Initial experiments used Llama 3.2 3B Instruct loaded locally on Google Colab, which avoided API costs but encountered significant limitations: context window constraints prevented including full error tracebacks and conversation history in feedback prompts, model loading required 5-10 minutes and substantial GPU memory (6GB+), and generation was slow (30-90 seconds per iteration), making the feedback loop impractical. To address these issues, we migrated to API-based generation, first testing Llama 3.3 70B via HuggingFace Inference API, which provided better code quality but suffered from availability issues (frequent 503 errors during model loading) and inconsistent endpoint behavior. We then experimented with Llama 3.1 8B via Groq API, which offered very fast generation (2-5 seconds) but encountered a critical problem: the verbose, detailed error feedback from the feedback loop (including full tracebacks, previous code snippets, and extensive error analysis) overwhelmed the model's context and led to confusion, causing the model to fix non-existent issues or introduce new errors while attempting to address the extensive feedback. This experience revealed that more feedback was not necessarily better. We ultimately settled on GPT-4 via OpenAI API, which provides reliable availability, fast generation (5-15 seconds), good error understanding, and robust handling of conversational context. Crucially, this led us to simplify the feedback loop: instead of including full error tracebacks, previous code snippets, and extensive error analysis, we now provide only the original clean prompt, a short error summary (300 characters), and minimal targeted guidance. This simplified approach reduces token usage, focuses the model on essential error information, and prevents the confusion that arose from overly verbose feedback, resulting in more reliable code generation and faster convergence.

C. Benchmarking Methodology

Performance evaluation uses PyTorch CUDA event timing to measure GPU execution time with high precision. The benchmarking framework tests kernels across multiple tensor shapes (e.g., [4, 128, 768], [8, 256, 1024], [16, 512, 2048]) and data types (float32 and float16) to evaluate performance across different configurations. For each configuration, the system performs 10 warmup runs to ensure GPU kernels are compiled and caches are warmed, followed by 200 measurement iterations. Each iteration uses torch.cuda.Event objects to record precise GPU timestamps: start_event.record() is called before kernel execution, end_event.record() after execution, and torch.cuda.synchronize() ensures GPU operations complete before measuring elapsed time. This approach captures pure GPU execution time (excluding CPU-GPU transfer overhead) by measuring the interval between CUDA events. Statistics are computed from the 200 measurements: mean latency, median (p50), and 95th percentile (p95) provide insights into both average performance and tail latency. The framework compares fused Triton kernels against unfused PyTorch reference implementations, calculating speedup as the ratio of unfused to fused mean execution time. All measurements exclude warmup runs to ensure stable, representative performance metrics.

D. Baseline Comparisons

Performance comparisons use PyTorch native implementations as baselines, which represent unfused, sequential execution of operations. For bias and activation fusion, the baseline uses torch.nn.functional.relu() and torch.nn.functional.gelu() applied to the result of element-wise addition (x + bias), requiring separate kernel launches for addition and activation. For RMS normalization, the baseline computes operations sequentially: element-wise bias addition, mean of squares computation using torch.mean(), square root using torch.sqrt(), normalization, and scale multiplication, each potentially triggering separate kernel launches. These PyTorch operations leverage CUDNN-optimized kernels when available, providing highly optimized implementations that serve as strong performance targets. CUDNN kernels are automatically selected by PyTorch based on tensor shapes, data types, and operation parameters, representing state-of-the-art vendor-optimized implementations. The baseline measurements capture the total execution time of these unfused sequences, including any intermediate memory allocations and kernel launch overhead, providing a fair comparison against fused Triton kernels that combine multiple operations into single kernel launches.

V. Results

Initial code generation attempts faced low compilation success rates due to Triton API compatibility issues. Generated kernels frequently failed with errors related to unsupported pointer types, non-existent Triton functions (e.g., tl.tanh, tl.select), incorrect kernel launch syntax, and dtype conversion problems. The iterative feedback loop improved success rates by enabling automatic error correction through targeted feedback and proactive code fixes. However, success rates remained variable and required extensive architectural tuning. The primary challenge was designing a flexible framework adaptable to different operation fusions (bias+activation, RMS normalization, softmax), each with unique tensor layouts, reduction patterns, and memory access requirements. The benchmarking phase revealed additional issues, as successfully compiled kernels often failed correctness validation or exhibited performance regressions. The system's effectiveness ultimately depended on careful prompt engineering, selective error feedback, and operation-specific adaptations rather than a universal approach.

Correctness validation verifies both shape matching and numerical accuracy. Shape validation ensures Triton kernel outputs match reference implementation dimensions, with mismatches immediately flagged as errors. Numerical accuracy is assessed using torch.allclose() with dtype-specific tolerances:

float32 requires rtol=1e-4 and atol=1e-5, while float16 uses looser tolerances (rtol=1e-2, atol=1e-3) to account for reduced precision. When mismatches occur, the system reports maximum difference values to aid debugging.

A common correctness issue encountered in bias+activation kernels involved activation path divergence. Generated kernels frequently implemented conditional activation logic (e.g., if activation_type == 0: apply ReLU) but omitted else clauses for non-ReLU activation types. This caused kernels to produce incorrect outputs when activation_type != 0, as the activation was not applied, leading to values that should have been zeroed (for ReLU) or transformed (for other activations) remaining unchanged. The feedback loop addressed this by detecting correctness errors and providing targeted guidance to ensure all activation paths are properly handled, either through explicit else clauses or by applying appropriate transformations for all activation types.

A. Performance Benchmarks

Table I and Table II present performance results for RMS normalization with bias fusion and bias with activation fusion, respectively. All measurements are averaged over 200 iterations after 10 warmup runs.

TABLE I
PERFORMANCE RESULTS: RMS NORMALIZATION + BIAS FUSION

Shape	Dtype	Fused (ms)	Unfused (ms)	Speedup
(4, 128, 768)	float32	0.16	0.11	0.64×
(4, 128, 768)	float16	0.21	0.12	0.55×
(8, 256, 1024)	float32	0.21	0.38	1.79×
(8, 256, 1024)	float16	0.18	0.25	1.40×

TABLE II
PERFORMANCE RESULTS: BIAS + ACTIVATION FUSION

Shape	Dtype	Fused (ms)	Unfused (ms)	Speedup
(4, 128, 768)	float32	0.05	0.04	0.83×
(4, 128, 768)	float16	0.05	0.03	0.66×
(8, 256, 1024)	float32	0.11	0.16	1.47×
(8, 256, 1024)	float16	0.08	0.11	1.46×

The results demonstrate that fusion benefits are highly dependent on tensor size and operation complexity. For RMS normalization with bias fusion, the generated Triton kernels show a clear size-dependent performance pattern. For smaller tensors (4, 128, 768), the fused implementation is 1.56-1.83× slower than the unfused PyTorch baseline, as kernel launch overhead and multi-kernel coordination costs dominate execution time. However, for larger tensors (8, 256, 1024), the fused implementation achieves 1.40-1.79× speedup, demonstrating that fusion benefits emerge when computation volume exceeds launch overhead. The improved performance on larger tensors indicates that the two-pass reduction approach (computing RMS statistics, then normalizing) becomes more efficient relative to PyTorch's sequential operations as problem size increases.

Bias with activation fusion shows a similar pattern but with more consistent results. For smaller tensors, the fused kernel is slightly slower (0.66-0.83× speedup), while larger tensors achieve 1.46-1.47× speedup. The simpler element-wise operation benefits more consistently from fusion, as it requires only a single kernel launch and eliminates intermediate memory writes present in the unfused implementation.

B. Impact of Feedback Loop

The iterative feedback loop improved compilation success rates and numerical correctness. Initial attempts frequently failed due to Triton API incompatibilities, but the loop enabled automatic error correction through targeted guidance and proactive code fixes, addressing issues like pointer type mismatches and non-existent API functions. Numerical correctness improved through iterative refinement, with the loop detecting errors via shape validation and numerical comparison, then providing targeted fixes for issues such as missing activation paths or incorrect reductions.

Iteration requirements varied by operation complexity. Bias with activation fusion, a simpler element-wise operation, converged in approximately 2 iterations. RMS normalization with bias fusion, involving complex two-pass reductions, required 8-9 iterations to achieve correctness, reflecting the increased complexity of reduction operations and multi-kernel coordination.

The feedback loop's effectiveness stems from key design choices: proactive code fixes address known problematic patterns before execution, simplified feedback provides concise error summaries rather than overwhelming tracebacks, and operation-specific adaptations enable targeted guidance for different fusion patterns. These mechanisms transform initial compilation failures into working, correct implementations through systematic iterative refinement.

VI. DISCUSSION

Our experiments reveal several important insights for AI-generated GPU kernel development. LLMs are not reliable for generating correct Triton kernels without iterative refinement—initial attempts consistently fail due to API incompatibilities. However, compiler-error feedback dramatically increases success rates by enabling targeted corrections, though this required extensive architectural tuning to balance context and avoid information overload. Performance results show that single operations lag behind PyTorch's CUDNN-optimized implementations, particularly for smaller tensor sizes, but fused operations achieve competitive performance on larger tensors by eliminating intermediate memory writes. Benchmarking-driven feedback proved essential, as successfully compiled kernels often failed correctness validation or exhibited performance regressions. Model choice significantly impacts development velocity: API-based generation with fast throughput enables rapid iteration cycles, whereas slower or unreliable APIs create bottlenecks. Throughout implementation, we encountered challenges in designing a flexible architecture adaptable to different operation fusions, each with unique requirements for tensor layouts, reduction patterns, and memory access patterns.

The system design evolved through several iterations in response to implementation challenges. Initially focused on multi-operation kernels, we encountered Triton API incompatibilities, compilation failures, and multi-kernel coordination difficulties, leading to a pivot toward fused operations that combine related operations into single kernels. We built a compiler-error feedback loop to iteratively refine generated code and switched to OpenAI API-based generation for faster iteration cycles. This approach addresses a significant industry need: fused kernels are typically hand-written even in optimized libraries like CUDNN, and optimized fused kernels do not always exist for custom operation combinations. AI-generated Triton code provides an accessible path to achieve performance benefits that would otherwise require expert GPU programming knowledge and significant development time.

A. Limitations

The system faces several limitations that impact its practical deployment. Context window constraints limit the amount of conversation history and error feedback that can be included in prompts, requiring careful selection of the most relevant information. Generation speed depends on API response times, with each iteration taking 5-15 seconds, making multi-iteration feedback loops time-consuming for complex operations requiring 8-9 iterations. Memory requirements are minimal for API-based generation but GPU memory is still needed for kernel execution and benchmarking. Token limits and API rate limits can restrict the number of iterations or the detail level of feedback, potentially requiring fallback strategies or subscription upgrades for production use.

VII. Related Work

Our work builds on several threads of research in GPU kernel optimization, LLM-based code generation, and dynamic code transformation systems.

A. LLM-Based GPU Kernel Generation

The application of LLMs to GPU kernel generation has received increasing attention as model capabilities have improved. KernelBench [4] established a benchmark for evaluating LLMs' ability to generate efficient GPU kernels, framing the task as transpiling PyTorch operations to CUDA. Their evaluation revealed that while LLMs can generate syntactically correct code, achieving both correctness and competitive performance remains challenging. The benchmark uses a metric called `fast_p` that captures the fraction of generated kernels that are both correct and faster than a baseline threshold.

GEAK (Generating Efficient AI-centric Kernels) [3], developed by AMD, presents an agentic framework for Triton kernel generation targeting AMD GPUs. GEAK employs a multi-agent architecture with specialized modules for generation, reflection, evaluation, and optimization. By leveraging inference-time compute scaling and Reflexion-style feedback mechanisms, GEAK achieves up to 63% execution accuracy and 2.59× speedup over direct LLM prompting. Our work shares GEAK's insight that iterative refinement is essential for reliable kernel generation, though we focus on a simpler single-agent architecture with compiler feedback.

B. Dynamic Code Transformation

The idea of dynamically transforming running code to adapt to changing conditions has a rich history in systems research. Protean Code [2] introduced a mechanism for enacting arbitrary compiler transformations at runtime with negligible overhead. The key insight was that instead of maintaining full control throughout execution (as in traditional dynamic optimizers), the system allows the original binary to run continuously and diverts control flow only at virtualized points, enabling rapid switching to new code variants. Protean Code demonstrated the verify-promote-rollback discipline that enables safe deployment of dynamically generated code: new variants are first verified for correctness, then promoted to production use, with the ability to roll back if problems are detected.

While Protean Code focused on transformations like inserting non-temporal memory hints to manage cache contention in datacenters, the underlying principle - that code can be safely transformed at runtime in response to changing conditions - directly informs our vision for LLM-based kernel generation. A future system might generate kernel variants on demand as workload characteristics change, verify them against reference implementations, and promote successful variants to production use.

C. Autotuning and Search-Based Optimization

Traditional autotuning systems like Halide [7] and TVM [8] frame kernel optimization as search over configuration spaces. These systems have achieved impressive results by combining domain-specific scheduling languages with learned cost models and search algorithms. However, they operate within predefined search spaces and cannot generate fundamentally new implementations. Recent work has explored using machine learning to guide the search process, but the fundamental limitation remains: the system can only find configurations within its predefined space.

Our approach differs by leveraging LLMs to generate novel implementations rather than selecting from predefined options. This enables exploration of optimizations that might not be captured in traditional search spaces, though it introduces new challenges around ensuring correctness and performance of the generated code.

VIII. Future Work

Several directions would enhance the system's practical applicability. Extended fused operation coverage would support a broader range of operation combinations, requiring discovery and optimization to create a unified template that works across all operation types rather than operation-specific adaptations. A production-level application would enable users to select operations and generate optimized fused Triton code through a manually tunable interface adaptable to various use cases. Reintegration into running models would implement a verify-promote-rollback workflow, allowing generated kernels to be tested in production environments with automatic rollback if performance degrades. More sophisticated feedback mechanisms could incorporate performance metrics directly into the feedback loop, enabling optimization for both correctness and speed. Larger scale evaluation across diverse operation types, tensor sizes, and hardware configurations would validate the system's generalizability and identify patterns for improving the unified template approach.

IX. Conclusion

We present a system for automatically generating optimized Triton kernels through iterative AI-driven code generation with compiler-error feedback. Our approach successfully generates fused kernels that achieve competitive or superior performance compared to PyTorch's CUDNN-optimized implementations, with bias+activation fusion achieving 1.46-1.47× speedup and RMS normalization achieving 1.40-1.79× speedup on larger tensors. The iterative feedback loop enables automatic error correction, improving compilation success rates and numerical correctness through targeted guidance and proactive code fixes. Key contributions include the simplified feedback mechanism that balances context with clarity, operation-specific adaptations for different fusion patterns, and demonstration that AI-generated kernels can compete with hand-optimized vendor implementations. This work demonstrates that generating optimized fused kernels—traditionally requiring expert GPU programming knowledge—can be achieved with just a few clicks, democratizing access to high-performance GPU kernels and enabling rapid development of custom operation fusions that may not exist in standard libraries.

Acknowledgment

References

[1] P. Tillet, H. T. Kung, and D. Cox, "Triton: An intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019.

[2] M. A. Laurenzano, Y. Zhang, L. Tang, and J. Mars, "Protean code: Achieving near-free online code transformations for warehouse scale computers," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[3] J. Wang, V. Joshi, S. Majumder, X. Chao, B. Ding, Z. Liu, P. P. Brahma, D. Li, Z. Liu, and E. Barsoum, "GEAK: Introducing Triton kernel AI agent & evaluation benchmarks," *arXiv preprint arXiv:2507.23194*, 2025.

[4] A. Ouyang, S. Guo, S. Arora, A. L. Zhang, W. Shao, A. Fang, and C. Ré, "KernelBench: Can LLMs write efficient GPU kernels?" *arXiv preprint*, 2024.

[5] Sakana AI, "The AI CUDA Engineer: Agentic CUDA kernel discovery, optimization and composition," 2024.

[6] W. Chen et al., "CUDA-LLM: LLMs can write efficient CUDA kernels," *arXiv preprint arXiv:2506.09092*, 2025.

[7] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

[8] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.