AI AS A COMPILER: COMPILING TRITON KERNELS WITHOUT THE TRITON COMPILER

Anonymous authors
Paper under double-blind review

ABSTRACT

Compiler backends are expensive to build and maintain as programming models, workloads, and accelerators evolve. We investigate whether large language models can replace the conventional optimizing and lowering pipeline, a process that we call *AI lowering*. We study AI lowering from Triton to NVIDIA PTX: an LLM agent translates Triton kernels directly into PTX, bypassing Triton's intermediate representations and optimization passes while preserving the kernel ABI, and refines its output using randomized differential testing, benchmarking, and profiler feedback. Across twelve common kernels on Ada, Hopper, and Blackwell GPUs and ten kernels from recent ML papers, *AI lowering* achieves $0.83\times$-$3.34\times$ the performance of autotuned Triton. The largest gains come from transformations that Triton's lowering pipeline does not perform, such as decoding packed binary weights directly into Tensor Core operands ($3.34\times$ on BitDelta), assigning each thread a complete softmax row in tensor memory ($1.36\times$ on FlashAttention), and reusing overlapping convolution windows (up to $2.23\times$). Getting correctness guarantees is now the bottleneck. The state-of-the-art PTX verifier does not model the Hopper and Blackwell features these kernels rely on, and restricting generation to its supported fragment erases the gains on compute-heavy kernels. With best-effort extensions, the verifier checks 16 of 22 kernels; atomics, input-dependent control flow, and bit-level floating-point manipulation remain out of reach, making verification the central open challenge for trusted AI compilation.

1 INTRODUCTION

High-performance GPU kernels require substantial hardware-specific expertise, making them costly to develop, optimize, and maintain as workloads and architectures evolve. Domain-specific languages such as Triton, TileLang, CuTe DSL, ThunderKittens, HipKittens, and ParallelKittens reduce the burden on kernel programmers by exposing tile-oriented or hardware-aware abstractions (Tillet et al., 2019; Wang et al., 2025; NVIDIA Corporation, 2025; Spector et al., 2024; Hu et al., 2025; Sul et al., 2025). However, they largely shift complexity into the compiler: supporting new abstractions and hardware features still requires implementing lowering, scheduling, legality checks, and optimizations. Traditional compiler backends further amplify this burden by relying on multiple intermediate representations (IRs), whose semantics, transformations, and lowering paths must be continuously extended and maintained. Systems such as MLIR aim to reduce duplicated infrastructure, yet large production compilers such as LLVM and XLA still illustrate how difficult it is to maintain extensive optimization and code-generation pipelines without introducing correctness or performance regressions (Lattner et al., 2020; Mannarswamy & Das, 2022; Reifferscheid, 2024).

This problem is becoming more severe as accelerator architectures diversify and evolve rapidly. GPUs, TPUs, Trainium, wafer-scale systems, LPUs, in-memory-compute processors, and custom AI ASICs expose increasingly different execution, memory, and communication models, while even successive generations of the same accelerator family can introduce capabilities that existing IRs cannot naturally express. Consequently, new systems increasingly introduce specialized abstractions and compilation paths, such as Tawa's asynchronous references for warp specialization or ThunderKittens' hardware-specific tile and warp abstractions (Chen et al., 2025; Spector et al., 2024). This rapid pace of hardware innovation makes conventional IR-centric compiler development increasingly expensive and difficult to keep aligned with state-of-the-art performance. When existing compiler abstractions cannot expose new hardware efficiently, performance-critical implementations bypass the IR using

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

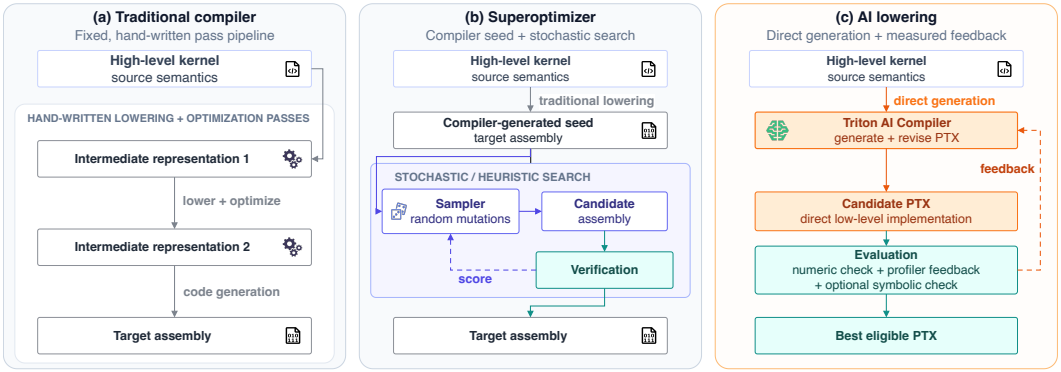

Figure 1: Traditional compilation, superoptimization, and AI lowering. (a) A traditional compiler applies hand-written lowering and optimization passes across intermediate representations. (b) A superoptimizer uses conventionally lowered target code as a seed, then iteratively mutates and scores candidate programs. (c) AI lowering bypasses the conventional optimizing/lowering backend: an LLM generates PTX directly and revises it using numerical, performance feedback, and optionally symbolic verification. In the search-based approaches, acceptance is guarded by numerical or formal checks.

target-specific intrinsics or assembly, as DeepGEMM does with Hopper-specific PTX instructions (Zhao et al., 2025; NVIDIA Corporation, 2026b).

We show that, for critical GPU kernels, **bypassing the conventional lowering pipeline is not only feasible but beneficial**. We introduce the TRITON AI COMPILER (TAIC), a direct Triton-to-PTX compiler that realizes AI lowering: an LLM synthesizes candidate PTX implementations directly from the Triton source. This contrasts with conventional compilers, which progressively lower high-level code through multiple hand-designed intermediate representations. We observe that AI lowering effectively optimizes PTX code in non-trivial ways that are inaccessible to the baseline Triton compiler such as combining Tensor Core instructions with warp-level shuffles, restructures data movement and computation directly at the PTX level, and selects target-specific instructions that are not exposed by Triton's evaluated lowering pipeline. Across Ada, Hopper, and Blackwell GPUs, AI lowering achieves between 0.83× and 3.34× the performance of optimized Triton-generated PTX. On B200, it also outperforms widely used optimized Triton kernels, including by 1.36× on FlashAttention and 1.10× on a Mamba-2 primitive.

While generation is inherently probabilistic, correctness should not be, yet existing verification tools cover only a fragment of modern GPU execution. Volta (Driscoll et al., 2026), the state of the art for PTX verification, symbolically executes every thread in a CTA and checks races, deadlocks, and output expressions, but primarily supports registers, shared memory, warp primitives, bulk-synchronous barriers, pre-Hopper instructions, and input-independent control flow. Modern kernels exceed this fragment: Blackwell's `tcgen05` interface adds tensor memory, descriptor-based layouts, asynchronous collective issue, `mbarrier` transaction accounting, and proxy fences (NVIDIA Corporation, 2026b), whose symbolic effects and ordering guarantees both need modeling. AI lowering uses more than 100 PTX instruction variants that do not appear in the original output of the Triton compiler. Restricting AI lowering to the verifier-supported subset erases its performance advantage, so we view this gap between rapidly evolving GPU ISAs and formally specified semantics as the new central challenge for AI kernel generation.

Our contributions are as follows:

- We introduce the TRITON AI COMPILER (TAIC) and the TRITON COMPILER ENVIRONMENT (TCENV), combining LLM-based PTX generation with compilation feedback, correctness testing, benchmarking, profiling, and iterative optimization, while preserving the kernel ABI[1], launch configuration, compile-time constants, and target-GPU contract.

[1]Application Binary Interface.

- We show that generated PTX achieves $0.83\times$-$2.23\times$ Triton performance on our twelve-kernel suite across Ada, Hopper, and Blackwell, uses architecture-specific primitives, and outperforms Triton on recent kernels, including BitDelta ($3.34\times$), FlashAttention ($1.36\times$), and Mamba-2 ($1.10\times$) on B200.
- Collectively, our results suggest a future in which AI compilers could replace custom-written intermediate representations with end-to-end, direct translation from high-level source to low-level code, lowering the complexity and time required to bring up software for new chips and architectures. We demonstrate the need for stronger verifiers to enable this full translation and list the current challenges to overcome.

2 Related Work

Tensor compilers, GPU DSLs, and kernel libraries. TVM and Ansor separate tensor computations from schedules and search over scheduling choices, while Mirage superoptimizes across kernel, thread-block, and thread levels (Chen et al., 2018; Zheng et al., 2020; Wu et al., 2025). Triton and Halide instead compile higher-level programming models, and CUTLASS and ThunderKittens provide expert-designed implementations and abstractions (Tillet et al., 2019; Ragan-Kelley et al., 2013; NVIDIA Corporation, 2026a; Spector et al., 2024). Rather than searching a hand-engineered schedule space or selecting a library implementation, AI lowering synthesizes hardware-specific PTX directly from a fixed Triton kernel.

LLM-generated GPU kernels. KernelBench and TritonBench evaluate CUDA or Triton generation from PyTorch-level tasks (Ouyang et al., 2025a; Li et al., 2025). AI CUDA Engineer adds evolutionary search (Lange et al., 2025), CUDA-L1 uses contrastive reinforcement learning, and Dr. Kernel studies multi-turn reinforcement learning with methods designed to limit reward hacking and biased gradients (Li et al., 2026; Liu et al., 2026). TritonGym provides a benchmark and tool-orchestration framework for agentic Triton generation (Guan et al., 2026), whereas PTXBench generates CUDA augmented with prescribed architecture-specific PTX instructions (Zhang et al., 2026a). Our task instead requires PTX-only output for a fixed, self-contained Triton program and leaves instruction selection unconstrained within the compilation contract.

Superoptimization and learned compilation. STOKE searches loop-free x86 binaries, while SuperCoder uses LLMs and reinforcement learning to optimize larger x86-64 assembly programs (Schkufza et al., 2012; Wei et al., 2026). Other work builds on explicit IRs: MLIR provides multi-level compiler infrastructure, CompilerGym exposes compiler decisions to agents, LLM Compiler learns LLVM IR and assembly representations, and CAKE co-designs an agent-facing GPU IR and compiler harness (Lattner et al., 2020; Cummins et al., 2021; 2024; Ye et al., 2026b). In contrast, AI lowering starts from Triton rather than target assembly and bypasses the conventional IR stack while jointly performing instruction selection, allocation, and scheduling in PTX.

Correctness of generated code. KernelBench and related systems rely on finite test suites (Ouyang et al., 2025a; Zhang et al., 2026b; Wu et al., 2025). Volta symbolically executes PTX and proves equivalence (Driscoll et al., 2026), but excludes the PTX instructions introduced by Hopper and Blackwell.

3 An Alternative Compilation Route

We choose Triton-to-PTX to explore AI lowering because it provides a clean kernel-level compilation boundary. Starting from PyTorch would introduce graph-level optimizations such as fusion, decomposition, and library dispatch, making kernel-level gains harder to isolate. CUDA, conversely, already exposes many low-level choices explicitly. Triton offers a useful middle ground: it specifies the computation and kernel interface while leaving hardware-specific decisions such as instruction selection, thread mapping, data movement, synchronization, and scheduling to the compiler.

3.1 PROBLEM FORMULATION

A Triton kernel is compiled under a fixed contract that determines both its specialization and execution. For each compilation instance, specialization corresponds to fixed values such as tensor dimensions, strides, and `constexpr` parameters (*the compile-time values*). The runtime launch interface (*the runtime interface*) specifies the grid dimensions, number of threads, and argument ABI. Finally, the generated code must target a specific GPU architecture (*the underlying hardware*). AI lowering must preserve this contract while replacing the conventional optimizing and lowering backend with model-generated PTX.

Formally, let K be a Triton kernel and let (c, I, h) denote its execution contract, where c denotes the fixed specialization parameters (*the compile-time values*), I denotes the launch configuration and argument ABI (*the runtime launch interface*), and h denotes the target GPU architecture (*the underlying hardware*). The contract induces a reference input-output semantics $[\![K]\!]_{c,I,h}$ over the legal runtime inputs $\mathcal{D}$; it does not designate a unique PTX program. A candidate PTX program P is correct when it exposes the required entry point and argument ABI and, under the same contract (c, I, h), satisfies

$$[\![P]\!]_{c,I,h}(x) \approx_\epsilon [\![K]\!]_{c,I,h}(x), \qquad \forall x \in \mathcal{D}. \tag{1}$$

Here $\approx_\epsilon$ denotes numerical equivalence within predefined absolute and relative tolerances; it allows small floating-point rounding differences without requiring bitwise identity. When formal verification is available, we additionally check exact equality after abstracting floating-point operations as real arithmetic. This check captures the intended arithmetic semantics rather than a particular sequence of compiler-generated floating-point operations.

AI lowering is an iterative synthesis process rather than a deterministic compilation procedure. At iteration t, the harness constructs a prompt $q_t = \mathrm{Prompt}(K, c, I, h, \mathcal{F}_{<t})$ using the compilation contract and, when available, feedback from earlier candidates, then queries the language model:

$$P_t \sim p_\theta(\cdot \mid q_t), \tag{2}$$

where p_θ is the model's conditional distribution.

Let PTX denote the space of PTX programs. The compilation contract defines the set of all valid lowerings of K for the fixed compile-time values c, runtime interface I, and target h as

$$\mathcal{V}(K, c, I, h) = \left\{ P \in \mathrm{PTX} \;\middle|\; \begin{array}{c} \mathrm{Compiles}(P,h) \\ \wedge\ \mathrm{Safe}(P) \\ \wedge\ \mathrm{ABI}(P){=}\mathrm{ABI}(K,I) \\ \wedge\ \forall x{\in}\mathcal{D}\colon [\![P]\!]_{c,I,h}(x){\approx_\epsilon}[\![K]\!]_{c,I,h}(x) \end{array} \right\}, \tag{3}$$

where $\mathrm{Compiles}(P, h)$ requires P to be a legal PTX program for target h, while $\mathrm{Safe}(P)$ requires memory safety and race freedom. Let $\widehat{L}(P; h)$ be the latency estimated by the fixed benchmarking protocol. The objective is $\min_{P \in \mathcal{V}(K,c,I,h)} \widehat{L}(P; h)$.

The remainder of the paper makes this formulation operational. Section 3.2 introduces TCENV, our environment for evaluating PTX. Section 3.3 describes TAIC, the harness that performs the iterative compilation process and NCU-guided optimization. Section 4.5 discusses verification and the limits of what can currently be verified soundly.

3.2 THE TRITON COMPILER ENVIRONMENT: AN ENVIRONMENT FOR AI LOWERING

We build the TRITON COMPILER ENVIRONMENT (TCENV), a controlled environment that allows TAIC to focus exclusively on AI lowering. It abstracts away configuration details that are orthogonal to lowering, such as compile-time constants, grid dimensions, threads per block, and the target GPU architecture: these are provided to Triton as fixed inputs during compilation, so they need not be inferred as part of AI lowering. Concretely, TCENV exposes a dictionary containing the Triton kernel, its hyperparameters, and execution metadata, together with a PTX template that the launcher can invoke without further configuration, with the function signature and arguments preserved in the PTX interface. It also provides the infrastructure required to assemble, launch, validate, and profile generated PTX. TAIC therefore receives a high-level kernel representation and produces a semantically equivalent low-level one, as defined in the previous section; Figure 2 illustrates this path alongside the standard compilation passes.

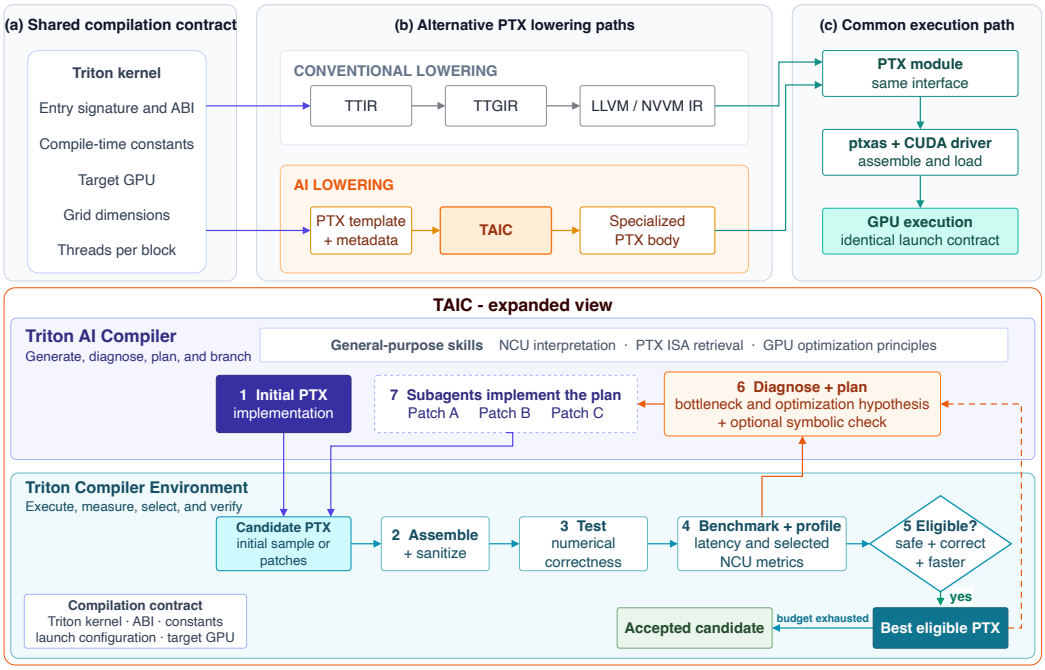

Figure 2: System overview and detailed AI-compilation loop. Top: conventional compilation and AI lowering start from the same compilation contract, produce an interface-compatible PTX module, and share the NVIDIA execution path. The orange block expands TAIC into the detailed loop below, where TAIC proposes and revises PTX while TCENV assembles, tests, benchmarks, profiles, and selects candidates.

An artificially weak baseline can make performance gains appear much larger than they are, effectively a form of reward hacking. This matters because semantically equivalent implementations can differ significantly in performance, as the fused softmax example in the Triton tutorial illustrates (OpenAI Triton Team, 2026a). We therefore construct each Triton reference following the optimization best practices of the official tutorials (OpenAI Triton Team, 2026b), and, before TCENV exposes a reference kernel, autotune it on the target GPU and use the resulting configuration throughout the evaluation.

TCENV combines randomized differential testing with optional symbolic verification (Section 4.5) to obtain correctness signals. The randomized tester generates inputs that stress overflow, underflow, cancellation, and precision loss, and rejects kernels whose outputs deviate from the reference beyond the prescribed tolerance. The symbolic verifier instead checks that the generated program implements the intended computation.

3.3 THE TRITON AI COMPILER: AN AGENT PERFORMING AI LOWERING

We build the TRITON AI COMPILER (TAIC), a harness that operates through TCENV (Section 3.2) to realize a direct Triton-to-PTX compilation path that bypasses Triton's intermediate compiler representations. TAIC is an LLM augmented by three general-purpose skills: interpreting Nsight Compute (NCU) metrics, retrieving PTX ISA documentation, and applying high-level GPU optimization principles. It performs generation and optimization, whereas TCENV provides the execution and verification infrastructure, a division of responsibilities made explicit in Figure 2.

TAIC first produces an initial PTX implementation, which TCENV assembles, sanitizes, checks for numerical correctness, and benchmarks. Inspired by verifier-backed inference-time scaling (Brown et al., 2024; Ouyang et al., 2025b), we then refine it: given the current PTX together with assembly and sanitizer outcomes, benchmark results, and selected profiler observations, TAIC diagnoses the dominant bottleneck and formulates an optimization plan, ranging from local instruction-level edits to global changes in thread mapping, data movement, or synchronization. Subagents realize this

plan independently as candidate patches, which TCENV evaluates under the same contract; only candidates that assemble, pass the safety and correctness gates, and improve the measured objective are eligible to seed the next round. The best eligible implementation and its measurement feedback return to TAIC, which repeats this diagnose-plan-branch-evaluate cycle until the search budget is exhausted.

4 EVALUATION

We use GPT-6 Astra as the compiler LLM. Each kernel is evaluated on three NVIDIA GPU generations: an L40S (Ada Lovelace, `sm_89`), an H100 (Hopper, `sm_90a`), and a B200 (Blackwell, `sm_100a`). Figure 3 summarizes the architecture-specific features present on these targets. Experiments run sequentially on otherwise idle GPUs. Generated code targets PTX ISA 8.7 with a 64-bit address space and is assembled for the corresponding architecture.

	Ada	used?	Hopper	used?	Blackwell	used?
MMA	`mma.sync` + `ldmatrix`	✓	`wgmma` + TMA loads	✓	`tcgen05` + TMEM accum.	✓
SMEM	XOR bank swizzle	✓	128-B XOR swizzle	✓	64-B offset swizzle	✓
Copies	`cp.async` pipeline	✓	TMA + `mbarrier`	✓	TMA 2-D + phase parity	✓

Figure 3: Architecture-specific features and their observed use in LLM-generated PTX. A check mark denotes that the generated kernels use the feature on the corresponding microarchitecture. The Hopper and Blackwell entries (`wgmma`, TMA, `mbarrier`, `tcgen05`, TMEM) fall outside Volta's originally validated semantics; verifying kernels that use them relies on the emulator extension of Section 4.5.

Our baseline uses Triton 3.6.0 and draws its kernels from TritonBench and OpenAI's official Triton tutorials (Li et al., 2025; OpenAI Triton Team, 2026b). To ensure a competitive baseline, we further optimize these kernels where applicable, including rewriting kernels when faster implementations can be obtained and removing boundary masks when the evaluated dimensions guarantee in-bounds accesses. For each kernel and GPU, we report the fastest correct configuration found through autotuning. To keep the comparison representative of standard Triton programming, the baseline kernels do not use experimental Triton features or manually written inline assembly.

We use GPT-6 Astra throughout our main evaluation because our model comparison shows that competitive AI lowering emerges only with the latest frontier models. Under the same protocol and inference budget on FP16 GEMM, GPT-5.4 reaches $0.73\times$ Triton performance, GPT-5.5 and GPT-5.6 Terra reach $0.87\times$, GPT-5.6 Luna fails to produce an accepted kernel, and GPT-5.6 Sol reaches $0.97\times$, while GPT-6 Astra is the first model to reach parity with Triton at $1.00\times$. Since Triton already provides a highly optimized baseline for this workload, we select GPT-6 Astra as the representative model for our experiments: it is the first model in our comparison for which direct AI lowering is consistently competitive with conventional compilation.

We evaluate kernels spanning four performance-critical LLM workloads: attention, GEMMs, activation functions, and normalization or regularization operations. These stress complementary aspects of GPU optimization, including Tensor Core utilization, reductions and synchronization, memory access, and instruction throughput. We use fixed problem sizes representative of dimensions commonly encountered in LLM workloads.

4.1 ARCHITECTURAL COVERAGE AND SOURCE SEMANTICS UNDERSTANDING

Architectural coverage. Figure 3 shows that the generated PTX uses each generation's modern mechanisms rather than a lowest-common-denominator instruction set: `mma.sync` with `ldmatrix` and `cp.async` pipelining on Ada, `wgmma` with TMA loads and `mbarrier` synchronization on Hopper, and `tcgen05` with tensor-memory accumulators and two-dimensional TMA transfers on Blackwell; shared-memory layouts likewise progress from XOR bank swizzling to 128-byte XOR and 64-byte offset swizzles. This coverage is also what makes verification difficult: the Hopper and Blackwell mechanisms fall outside the semantics originally validated by Volta, and reaching them requires the emulator extension of Section 4.5.

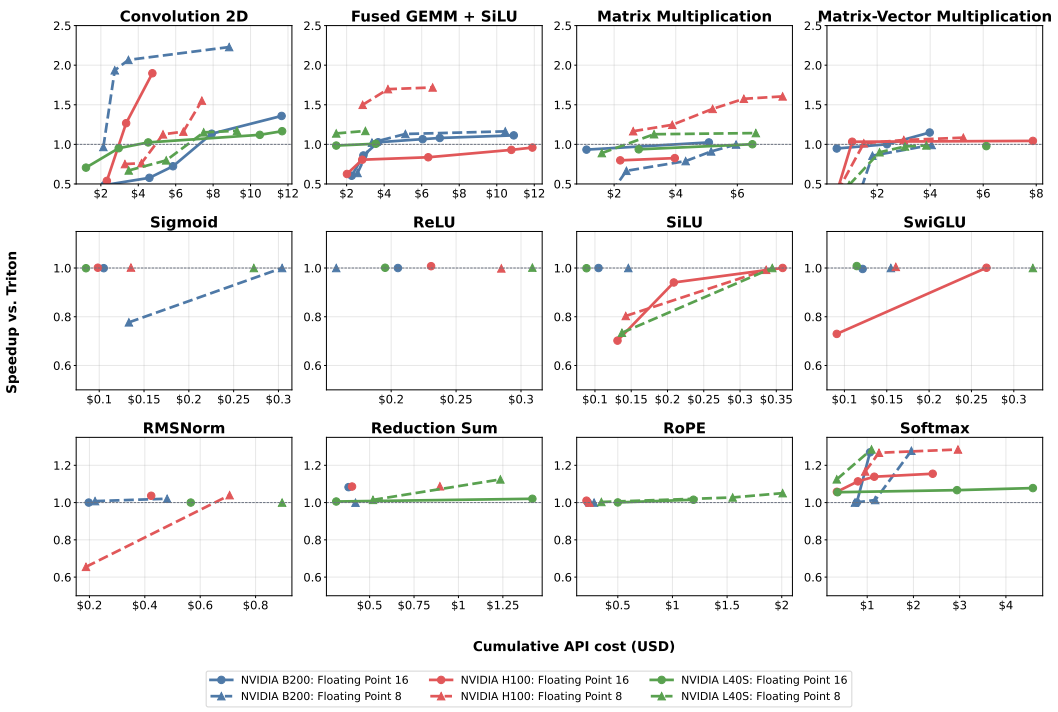

Figure 4: Per-kernel speedup/cost Pareto frontiers relative to Triton for FP16 and FP8 on three GPU architectures. Each point is a validated candidate that improves the best speedup reached at its cumulative API cost; the dashed line marks parity with the corresponding autotuned Triton baseline. Axes are scaled independently across panels.

Recovering semantics from source. Because many of our Triton kernels may have appeared in the model's training data, we evaluate whether TAIC actually recovers source semantics rather than relying on memorized implementations or familiar high-level cues. We first test randomly generated kernels of roughly 150 lines that implement no recognizable algorithm, operate on integers to avoid floating-point discrepancies, and exercise a broad range of Triton features, including integer and bitwise arithmetic, modulo and hashing-like computations, reductions such as `tl.sum` and `tl.max`, `tl.static_range` loops, `tl.where`, and hints such as `tl.multiple_of` and `tl.max_contiguous`; TAIC produces correct PTX for every one. We then introduce deliberately misleading cues: a matrix multiplication whose accumulation is changed to $c[i,j] \mathrel{-}= a[i,k] \cdot b[k,j]$, a FlashAttention variant with $\texttt{qk_scale} = \texttt{HEAD_DIM}^{-0.5} + 1.4426950408889634$, and a kernel named `ReLU` whose comments request GELU. In all three cases, the model follows the actual implementation rather than the name, comments, or familiar algorithmic structure, producing semantically equivalent PTX. Finally, when the Triton source requests IEEE-754 semantics, the generated PTX avoids Tensor Core operations, further indicating that the model captures semantic constraints expressed in the source rather than merely reproducing common optimized patterns.

4.2 Results on Common Kernels

Figure 4 shows that the benefit of AI lowering depends strongly on the operator, precision, and target GPU. Sigmoid, ReLU, SiLU, and SwiGLU finish within approximately 1% of Triton, and RMSNorm improves by at most 4%. These panels therefore primarily demonstrate recovery of baseline performance; their narrow vertical scales should not be read as large gains. RoPE improves by at most 5%, and matrix-vector multiplication ranges from $0.98\times$ to $1.15\times$. Softmax improves by 1.08-$1.29\times$, whereas convolution reaches 1.17-$2.23\times$. GEMM is more architecture- and precision-dependent: FP16 reaches parity on B200 and L40S but only $0.83\times$ on H100, while FP8 reaches $1.61\times$ on H100, $1.15\times$ on L40S, and approximately parity on B200. The fused GEMM-SiLU kernel exhibits the same H100 contrast ($0.96\times$ for FP16 versus $1.72\times$ for FP8). The cost axis also distinguishes easy recoveries from substantial searches: elementwise candidates generally cost less

than $1.50, while convolution and GEMM candidates require roughly $5-$17. The main grid reports a single run per experiment; Section 4.3 separately repeats four B200 workloads five times. To provide concrete examples of the optimizations discovered by TAIC, we briefly highlight three representative kernels. For reduction sum, the speedup comes from replacing Triton's generic reduction with architecture-specific Tensor Core or warp-reduction algorithms. For convolution, it comes from reusing inputs across overlapping windows and overlapping packed data movement with Tensor Core execution. For H100 FP8 GEMM, it comes from using larger WGMMA tiles, fewer synchronization points, and a cheaper transposing epilogue. See Appendix A for detailed comparisons with Triton.

4.3 Repeatability under a Fixed Search Budget

To measure run-to-run variation, we repeat four representative B200 experiments five times with independent model samples. We configure a maximum API budget of $15 per run for convolution, matrix multiplication, and FlashAttention, and $1 for the simpler elementwise SwiGLU kernel. The budget gate is evaluated before issuing each model request; because a request is atomic, its completion can carry the final recorded cost beyond the configured gate. Table 1 reports the best validated candidate from each run in chronological order. We observe that TRITON AI COMPILER matches or outperforms the Triton compiler, although with a moderate variance for kernels with significant gains over the baseline.

Kernel	Budget	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Mean $\pm$ SD	Best
Convolution (FP8)	$15	$2.081\times$	$1.706\times$	$2.229\times$	$1.933\times$	$2.070\times$	2.004 ± 0.197	$2.229\times$
Matrix multiplication (FP16)	$15	$1.002\times$	$1.026\times$	$1.032\times$	$1.029\times$	$1.024\times$	1.023 ± 0.012	$1.032\times$
SwiGLU (FP16)	$1	$1.000\times$	$1.002\times$	$1.000\times$	$1.000\times$	$1.000\times$	1.000 ± 0.001	$1.002\times$
FlashAttention forward (FP16)	$15	$1.261\times$	$1.075\times$	$1.214\times$	$1.263\times$	$1.366\times$	1.236 ± 0.106	$1.366\times$

Table 1: Repeatability on NVIDIA B200 across five independent TRITON AI COMPILER runs. Values are speedups over the same autotuned Triton baseline. Budget is the configured per-run API-cost gate, checked before each new model request; an in-flight request can finish above it.

4.4 Results on Recently Published Kernels

Beyond the twelve-kernel suite of Section 4, we evaluate ten kernels from recent ML papers: four attention variants, FlashAttention (Dao et al., 2022), FlashSinkhorn (Ye et al., 2026a), Forgetting Attention (Lin et al., 2025), and SageAttention (Zhang et al., 2025), two BitDelta GEMM variants (Liu et al., 2024), standard and batched matrix multiplication, two memory-bound kernels from Lion (Chen et al., 2023) and Dion (Ahn et al., 2025), and two Mamba-2 primitives (Dao & Gu, 2024). The Mamba-2 kernels differ substantially from attention, propagating compact recurrent state rather than forming pairwise token interactions. We use the authors' official open-source implementations without modifying their algorithms or optimizations, except for removing unnecessary masks, and autotune relevant parameters to obtain strong Triton baselines. Figure 5 summarizes the results.

AI lowering substantially outperforms Triton on several workloads, most notably the two BitDelta variants ($3.34\times$ and $2.51\times$), FlashSinkhorn ($1.55\times$), SageAttention ($1.41\times$), and FlashAttention ($1.36\times$). Memory-bound and recurrent workloads remain near parity, including Forgetting Attention ($1.01\times$), Lion ($1.00\times$), Dion ($0.97\times$), and Mamba-2 ($1.07\times$ and $1.10\times$). The largest gains therefore arise when generated PTX can reorganize computation and data movement. BitDelta decodes packed binary weights directly into Tensor Core operands; FlashAttention maps complete score rows to threads in the native tensor-memory layout, avoiding cross-lane reductions; and FlashSinkhorn performs independent warp-local log-sum-exp reductions while directly issuing FP16 MMA instructions. Appendix A provides detailed comparisons.

4.5 Formal Verification in the Loop

When we activate the use of Volta (Driscoll et al., 2026) in the compilation loop, a candidate is valid only if it passes the numerical check *and* Volta proves it equivalent to the Triton reference. When a kernel uses a pattern outside Volta's support, TCENV reports the unsupported feature to TAIC, which then revise the candidate. Table 2 shows that the resulting kernels are significantly less competitive: the features that deliver performance on modern GPUs are not yet all supported by Volta.

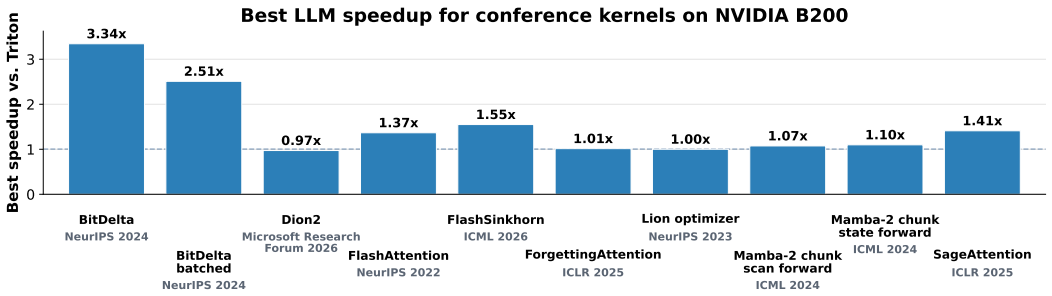

Figure 5: Best validated LLM-generated speedup relative to the corresponding Triton baseline for recently published conference kernels on NVIDIA B200. Each bar reports the best accepted candidate for one kernel; the dashed line marks parity with Triton.

We then iterate over the unsupported features that TAIC attempts to use when left to run free (without Volta in the loop), and do a best effort to add support for them in Volta: (i) support for mathematical constants as immediate, (ii) support for more instructions (e.g., the `tanh` instruction), (iii) asynchronous copies, (iv) asynchronous Tensor Core instructions and their operand movement (`tcgen05`, `wgmma`, `ldmatrix`), (v) a broader range of barriers, (vi) shuffle-based reductions and (vii) packed formats, in which several narrow values share one register. This support is not free: Volta's trusted computing base grows from 42 kLoC to 60 kLoC, and every new instruction semantics is a new risk of unsoundness. Appendix B gives details.

With these features, we can verify the unrestricted PTX output of TAIC (without constraining with Volta in the loop) for every kernel in the common-kernel suite except the reduction sum and the softmax, and for six of the ten recently published kernels (FlashAttention, Forgetting Attention, SageAttention, Lion, and both Mamba-2 primitives).

Kernel	Provably Verified	Numerically Verified
ReLU	1.00×	1.00×
GEMV	1.15×	1.15×
GEMM	0.24×	1.03×
FlashAttention	0.78×	1.37×
Convolution 2D	0.77×	1.35×

Table 2: Performance of the best provably verified and best implementation on an NVIDIA B200 using FP16 using original Volta.

Three classes of constructs remain unsupported: atomics, control, and bit-level value manipulation. Volta abstracts floating-point operations as real arithmetic, so it cannot reason about code that computes a function by operating directly on the bit-level encoding. For example, TAIC can implement floating-point max and min reductions with the integer `redux.sync` min/max instructions, after a sign-aware remapping of the bit patterns that makes integer order agree with floating-point order. The correctness of this reduction follows from the encoding, which the real-valued abstraction does not model. Atomic updates let threads write the same location by design; checking them requires reasoning about all interleavings of the conflicting updates, which lies outside Volta's execution model. Finally, Volta targets structured-CTA kernels without input-dependent control flow, which excludes, for instance, piecewise implementations that select a different formula depending on the input range.

5 Conclusion

We show that large language models can directly lower specialized Triton kernels to competitive PTX that passes random differential testing, and are now limited by the capacities of the verifier. Although generation can be orders of magnitude more expensive than millisecond-scale Triton compilation, this cost is incurred once and can be amortized over repeated kernel execution. AI lowering may be particularly compelling for new or lightly supported hardware, where synthesis could be substantially cheaper than developing and maintaining a dedicated optimizing compiler backend. More broadly, with sufficiently strong verification, AI lowering points toward a new paradigm for generating performance-critical code on hardware accelerators.

AI USE STATEMENT

Generative AI models are a central component of the methodology evaluated in this work: we use large language models to generate and optimize candidate GPU kernels, as described in the main text. All generated kernels are independently compiled, tested for correctness, benchmarked, and, where supported, verified using the evaluation pipeline described in the paper. We additionally used generative AI tools during the research process to assist with software development, code editing, brainstorming, and editing the manuscript for clarity and readability. All AI-assisted code, experimental results, technical claims, and manuscript content were reviewed and validated by the authors.

ETHICS STATEMENT

This work studies the use of large language models for GPU kernel compilation and optimization. It does not involve human subjects, personal or sensitive data, or the collection or release of new datasets. All generated code is evaluated in controlled computational environments using automated correctness, performance, and, where supported, formal verification checks. We do not identify specific ethical concerns arising from the experiments conducted in this work beyond the general risks associated with automated code generation, such as the possibility of producing incorrect or unsafe code when appropriate validation is absent. Our methodology explicitly emphasizes independent correctness testing and verification to mitigate these risks.

REPRODUCIBILITY STATEMENT

To support reproducibility, we will release the complete source code for TAIC and TCENV, together with the PTX generation and optimization pipeline, evaluation scripts, benchmark configurations, and the generated kernels used in our experiments. We will also release the scripts and configurations required to reproduce the experiments reported in the paper across the evaluated GPU architectures, including correctness testing, benchmarking, profiling, and formal verification where supported. All artifacts will be made publicly available upon publication.

REFERENCES

Kwangjun Ahn, Byron Xu, Natalie Abreu, Ying Fan, Gagik Magakyan, Pratyusha Sharma, Zheng Zhan, and John Langford. Dion: Distributed orthonormalized updates, 2025. URL https://arxiv.org/abs/2504.05295.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling, 2024. URL https://arxiv.org/abs/2407.21787.

Hongzheng Chen, Bin Fan, Alexander Collins, Bastian Hagedorn, Evghenii Gaburov, Masahiro Masuda, Matthew Brookhart, Chris Sullivan, Jason Knight, Zhiru Zhang, and Vinod Grover. Tawa: Automatic warp specialization for modern gpus with asynchronous references, 2025. URL https://arxiv.org/abs/2510.14719.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, abs/1802.04799, 2018. URL http://arxiv.org/abs/1802.04799.

Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, Yifeng Lu, and Quoc V Le. Symbolic discovery of optimization algorithms. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 49205–49233. Curran Associates, Inc., 2023. doi: 10.52202/075280-2140. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/9a39b4925e35cf447ccba8757137d84f-Paper-Conference.pdf.

Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, and Hugh Leather. Compilergym:

Robust, performant compiler optimization environments for ai research, 2021. URL https://arxiv.org/abs/2109.08267.

Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. Meta large language model compiler: Foundation models of compiler optimization, 2024. URL https://arxiv.org/abs/2407.02524.

Tri Dao and Albert Gu. Transformers are ssms: Generalized models and efficient algorithms through structured state space duality, 2024. URL https://arxiv.org/abs/2405.21060.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022. URL https://arxiv.org/abs/2205.14135.

Benjamin Driscoll, Kshitij Dubey, Anjiang Wei, Neeraj Kayal, Rahul Sharma, and Alex Aiken. Equivalence checking of ml gpu kernels, 2026. URL https://arxiv.org/abs/2511.12638.

Yue Guan, Yichen Lin, Xu Zhao, Jianzhu Yao, Xinwei Qiang, Zhongkai Yu, Pramod Viswanath, Yufei Ding, and Adnan Aziz. Tritongym: A benchmark for agentic llm workflows in triton gpu code generation. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, 2026.

William Hu, Drew Wadsworth, Sean Siddens, Stanley Winata, Daniel Y. Fu, Ryann Swann, Muhammad Osama, Christopher Ré, and Simran Arora. Hipkittens: Fast and furious amd kernels, 2025. URL https://arxiv.org/abs/2511.08083.

Robert Tjarko Lange, Aaditya Prasad, Qi Sun, Maxence Faldor, Yujin Tang, and David Ha. The ai cuda engineer: Agentic cuda kernel discovery, optimization and composition. Sakana AI technical report, 2025. URL https://sakana.ai/ai-cuda-engineer/.

Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A compiler infrastructure for the end of moore's law. *CoRR*, abs/2002.11054, 2020. URL https://arxiv.org/abs/2002.11054.

Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, Haojie Wang, Jianrong Wang, Xu Han, Zhiyuan Liu, and Maosong Sun. Tritonbench: Benchmarking large language model capabilities for generating triton operators, 2025. URL https://arxiv.org/abs/2502.14752.

Xiaoya Li, Albert Wang, Guoyin Wang, Jiwei Li, and Chris Shum. Cuda-l1: Improving cuda optimization via contrastive reinforcement learning, 2026. URL https://arxiv.org/abs/2507.14111.

Zhixuan Lin, Evgenii Nikishin, Xu Owen He, and Aaron Courville. Forgetting transformer: Softmax attention with a forget gate, 2025. URL https://arxiv.org/abs/2503.02130.

James Liu, Guangxuan Xiao, Kai Li, Jason D. Lee, Song Han, Tri Dao, and Tianle Cai. Bitdelta: Your fine-tune may only be worth one bit, 2024. URL https://arxiv.org/abs/2402.10193.

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

Sandya Mannarswamy and Dibyendu Das. Learning to combine instructions in llvm compiler, 2022. URL https://arxiv.org/abs/2202.12379.

NVIDIA Corporation. CuTe DSL documentation, 2025. URL https://docs.nvidia.com/cutlass/media/docs/pythonDSL/cute_dsl.html.

NVIDIA Corporation. CUTLASS: CUDA templates and Python DSLs for high-performance linear algebra. https://github.com/NVIDIA/cutlass, 2026a.

NVIDIA Corporation. Parallel thread execution ISA documentation. https://docs.nvidia.com/cuda/parallel-thread-execution/, 2026b.

OpenAI Triton Team. Fused softmax tutorial. Triton Documentation, 2026a. URL https://triton-lang.org/main/getting-started/tutorials/02-fused-softmax.html.

OpenAI Triton Team. Triton tutorials. Triton Documentation, 2026b. URL https://triton-lang.org/main/getting-started/tutorials/.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels?, 2025a. URL https://arxiv.org/abs/2502.10517.

Anne Ouyang, Azalia Mirhoseini, and Percy Liang. Surprisingly fast AI-generated kernels we didn't mean to publish (yet). Stanford Center for Research on Foundation Models, 2025b. URL https://crfm.stanford.edu/2025/05/28/fast-kernels.html.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman 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*, PLDI '13, pp. 519–530, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320146. doi: 10.1145/2491956.2462176. URL https://doi.org/10.1145/2491956.2462176.

Johannes Reifferscheid. New emitters for XLA:GPU. OpenXLA Request for Comments, 2024. URL https://github.com/openxla/xla/discussions/13093.

Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization, 2012. URL https://arxiv.org/abs/1211.0557.

Benjamin F. Spector, Simran Arora, Aaryan Singhal, Daniel Y. Fu, and Christopher Ré. Thunderkittens: Simple, fast, and adorable ai kernels, 2024. URL https://arxiv.org/abs/2410.20399.

Stuart H. Sul, Simran Arora, Benjamin F. Spector, and Christopher Ré. Parallelkittens: Systematic and practical simplification of multi-gpu ai kernels, 2025. URL https://arxiv.org/abs/2511.13940.

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

Lei Wang, Yu Cheng, Yining Shi, Zhengju Tang, Zhiwen Mo, Wenhao Xie, Lingxiao Ma, Yuqing Xia, Jilong Xue, Fan Yang, and Zhi Yang. Tilelang: A composable tiled programming model for ai systems, 2025. URL https://arxiv.org/abs/2504.17577.

Anjiang Wei, Tarun Suresh, Huanmi Tan, Yinglun Xu, Gagandeep Singh, Ke Wang, and Alex Aiken. Supercoder: Assembly program superoptimization with large language models, 2026. URL https://arxiv.org/abs/2505.11480.

Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Man Kit Ao, Praveen Velliengiri, Xupeng Miao, Oded Padon, and Zhihao Jia. Mirage: A Multi-Level superoptimizer for tensor programs. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pp. 21–38, Boston, MA, July 2025. USENIX Association. ISBN 978-1-939133-47-2. URL https://www.usenix.org/conference/osdi25/presentation/wu-mengdi.

Felix X. F. Ye, Xingjie Li, An Yu, Ming-Ching Chang, Linsong Chu, and Davis Wertheimer. Flashsinkhorn: Io-aware entropic optimal transport on gpu, 2026a. URL https://arxiv.org/abs/2602.03067.

Zihao Ye, Yingyi Huang, Hongyi Jin, Bohan Hou, Junru Shao, Zhongming Yu, Jinqi Chen, Meghan Cowan, Shiyi Cao, Shanli Xing, Hanfeng Chen, Vinod Grover, Tianqi Chen, and Luis Ceze. Cake: Compiler-agent co-design for frontier kernel evolution, 2026b. URL https://arxiv.org/abs/2608.12629.

Genghan Zhang, Yixin Dong, Chengze Fan, Zhichen Zeng, Yueming Yuan, Shaowei Zhu, and Kunle Olukotun. Ptxbench: Benchmark and adapt llms for gpu kernel optimization with architecture-specific ptx, 2026a. URL https://arxiv.org/abs/2608.17379.

Jintao Zhang, Jia Wei, Haofeng Huang, Pengle Zhang, Jun Zhu, and Jianfei Chen. Sageattention: Accurate 8-bit attention for plug-and-play inference acceleration, 2025. URL https://arxiv.org/abs/2410.02367.

Yunxiang Zhang, Ping Yu, Jianyu Wang, Max, Fan, Julian Reed, Azalia Mirhoseini, and Will Su. Kernelbench-verified: Do llm-generated kernels actually beat pytorch?, 2026b. URL https://arxiv.org/abs/2607.16241.

Chenggang Zhao, Zhean Xu, Liang Zhao, Jiashi Li, Chenhao Xu, Anyi Xu, Shengyu Liu, Kexing Zhou, and Kuai Yu. Deepgemm: clean and efficient blas kernel library on gpu. https://github.com/deepseek-ai/DeepGEMM, 2025.

Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor : Generating high-performance tensor programs for deep learning. *CoRR*, abs/2006.06762, 2020. URL https://arxiv.org/abs/2006.06762.

A Representative Optimization Examples

This section examines six representative kernels for which the generated PTX outperforms Triton. The first three examples come from the common-kernel suite; the remaining three come from recently published workloads evaluated on B200.

A.1 Common Kernels

Reduction sum: changing the reduction algorithm. For a 256×4096 input, Triton lowers `tl.sum` to a conventional reduction: it accumulates values within threads and warps, combines partial results through shared memory, and finally uses atomic updates. The generated PTX instead expresses each row sum as $X\mathbf{1}$ and uses Tensor Cores with packed FP16 or FP8 inputs and FP32 accumulation. On H100 FP16, two $16 \times 8 \times 16$ `mma.sync` instructions replace scalar conversion and reduction chains; three shuffle-and-add stages then reduce the MMA outputs, with one atomic update per warp and no shared-memory reduction or block barrier. On L40S FP8, a single warp uses one $16 \times 8 \times 32$ MMA and reaches $1.125\times$ Triton performance. On H100 FP8, the kernel instead scales E4M3 inputs, computes integer-valued partial sums with `redux.sync.add`, and combines four warp sums using one barrier, reaching $1.09\times$.

Convolution: reusing overlapping input windows. The convolution maps a $32 \times 64 \times 56 \times 56$ input and $128 \times 64 \times 3 \times 3$ filters to $32 \times 128 \times 54 \times 54$ outputs. Triton lowers it to a Tensor-Core implicit GEMM with a reduction dimension of 576, but uses scalar loads and shared-memory staging. The generated kernels instead reuse values shared by neighboring 3×3 windows. On H100 FP16, 32-bit loads fetch adjacent pixels, funnel shifts construct neighboring values, and packed shared-memory stores and double buffering overlap operand preparation with WGMMA, yielding a $1.90\times$ speedup. On B200 FP8, byte permutations reuse a small set of 32-bit loads across several filter offsets, while pipelined input and weight buffers feed `tcgen05.mma`, yielding $2.23\times$. On L40S FP8, a circular shared-memory cache provides similar reuse and reaches $1.17\times$. These kernels retain the same Tensor Core computation as Triton but eliminate redundant gathers through input reuse, packed accesses, and deeper pipelining.

H100 FP8 GEMM: reducing WGMMA and epilogue overhead. The generated kernel uses larger `wgmma.mma_async.m64n256k32` operations, whereas Triton uses `m64n64k32`, requiring more WGMMA instructions to cover the same output tile. Explicit double buffering prepares the next A/B tile while the current Tensor Core operations execute, and synchronization occurs only when a shared-memory stage is consumed or reused. The epilogues also differ: Triton repeatedly stores accumulator fragments to shared memory, synchronizes, and reloads them to obtain the required layout, whereas the generated kernel uses a transposing `stmatrix` variant, followed by one synchronization and vectorized global stores. These choices reduce instruction, synchronization, and layout-conversion overhead, explaining the gain on Hopper and its absence on the other architectures.

A.2 Recently Published Kernels

These examples compare each generated implementation with the Triton PTX from the same run. They show that the benefit comes from jointly reconsidering the instruction sequence, data movement, synchronization, and decomposition at the target level, rather than from emitting PTX directly in isolation.

BitDelta: decoding binary weights directly into Tensor Core operands. For the evaluated 64×128 output tile, Triton unpacks each binary weight, computes $2b - 1$ with integer arithmetic, converts the result to FP16, and stages the decoded weights in shared memory. The generated PTX constructs packed FP16 representations of ± 1 directly: a bit permutation aligns the packed weights with FP16 sign positions, after which each `lop3` produces two decoded values by modifying the sign bits of a packed constant. It also transposes the internal matrix product so that decoded weights can be written directly to the tensor-memory operand of `tcgen05.mma`, eliminating a shared-memory store-reload cycle. A four-stage activation pipeline, two alternating weight buffers, and a reduction step increased from 32 to 64 further reduce staging, synchronization, and loop overhead. The resulting kernel reaches $3.34\times$ Triton performance, and the batched variant reaches $2.51\times$ using the same strategy.

FlashAttention: aligning softmax ownership with tensor-memory layout. Both implementations use Blackwell Tensor Cores for QK^{T} and PV, but organize the intervening softmax differently. Triton distributes each score row across two lanes and therefore requires cross-lane reductions. The generated PTX assigns each thread a complete 32-score row in the native tensor-memory layout, making the maximum and sum register-local reductions and eliminating shuffle instructions. It also combines two 64-row query tiles into one 128-row tile, so each key/value tile serves twice as many queries. Packed `f32x2` arithmetic reduces scalar work, while double buffering and overlap between normalization updates and matrix operations reduce block barriers from three to one per loop iteration. Together, these changes yield a $1.36\times$ speedup.

FlashSinkhorn: partitioning the reduction across independent warps. For $n = m = 1024$ and coordinate dimension 64, Triton performs 32 sequential log-sum-exp updates, with eight warps cooperating on each 32×32 tile. The generated PTX assigns each warp an independent 128-point slice and keeps separate maximum and sum accumulators. Each warp executes eight iterations without block barriers or shuffles in the loop, and the partial results are combined only once at the end. The fixed coordinate tile remains in registers, and the original FP16 coordinates feed `mma.sync.m16n8k16` directly, replacing Triton's FP16-to-FP32 conversions and TF32 `m16n8k8` operations. Doubling the reduction extent per MMA also halves the number of matrix instructions per dot product. The resulting kernel passes the evaluation tolerances and reaches $1.55\times$ Triton performance.

B Verification and Volta Extensions

The set of features supported by Volta is not sufficient for verifying the correctness of PTX programs optimized for recent GPU architectures (See Table 2). As a proof of concept we extend Volta, on a best effort basis, to support the most common instructions emitted by AI models. We do not claim our extensions to be sound or complete. We describe the extensions bellow, but their formalization is left as future work.

The main addition is the support for asynchronous operation. Instructions such as asynchronous copies (`cp.async`) introduce a new class of memory hazard, beyond cross-thread data races on synchronous memory accesses. Volta tracks data races at the byte granularity by tracking reader and writer threads accesses between synchronization barriers. We handle asynchronous data movement by also tracking asynchronous operations at both the source and target destinations. An asynchronous operation is considered in flight from its initiating instruction until the execution of an appropriate barrier instruction. Our extended Volta reports a memory hazard if an asynchronous data movement overlaps with either synchronous or asynchronous operation.

The next significant addition is the support for the long tail of instructions commonly found in generated kernels. Of those tensor core operations, *e.g.*, the tcgen05, wgmma, and ldmatrix instruction families, are the more complex due to the large set of configuration and their interaction with asynchronous operations. We further implement missing math operators, such as `tanh` and `exp2`, which we express in terms of exponentials to leverage the existing solver simplification rules. We add support for packed representations, such as 16x2 and 8x4 packed values, as well as arithmetic, conversion, and load/store operations on those.

Finally we found the need to explicitly support common patterns. For instance, in shuffle reductions, all threads are typically racily writing the *same* value to a memory address, a well defined and valid data races commonly used in kernels. To handle that, we modify Volta to establish the equality of the values being written at the time it checks for the race, invoking the equivalence checker in the middle of the symbolic execution.

Our extensions are sufficient to generate kernels that match or exceed the Triton compiler, but there remain significant intrinsic limitations. First, Volta does not support data-dependant control flow. Although rare, data-dependant control flow can be found in kernels for patterns such as segmented polynomial approximations or conditional update during streaming computations. Atomic operations are also out of scope to avoid the need for considering all possible thread interleaving. Finally, due to the infinite precision floating point representation, kernels that rely on properties of the bit layout of floating points numbers might be rejected by Volta despite producing the expected result with finite-precision arithmetics.