GLUEBALL: AGENTIC GENERATION AND VERIFICATION OF HARDWARE-OPTIMIZED GLUON KERNELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Modern GPUs provide increasingly powerful features to improve performance, yet effectively harnessing them requires reasoning about complex execution behavior. Large Language Model (LLM) agents are exceptionally well suited to exploring this optimization space through iterative code generation, with pre-defined test-suites executed on bare metal providing correctness guarantees and performance feedback. However, numerical testing alone provides limited coverage of the input and execution space, making it possible for agents to overfit to tested inputs while missing correctness violations arising from subtle hardware-execution semantics. To address these limitations, we present **Glueball**, the first system to equip LLM agents with theorem-proving tools to reason about correctness during GPU kernel generation and optimization. Glueball targets Gluon, a tile-level Domain Specific Language (DSL) within the Triton ecosystem that provides direct, fine-grained access to GPU hardware beyond the abstractions exposed by Triton's higher-level programming model. Our system coordinates optimization across single and multi-kernel workloads, using end-to-end model performance, as well as theorem-prover guidance, as optimization feedback to evolve the kernels. Our results show improved kernel bug detection, while evaluations on real-world workloads demonstrate end-to-end performance competitive with already performant Triton baselines.

1 INTRODUCTION

Automatic GPU kernel generation using LLMs has emerged as an active area of systems research, with benchmarks such as KernelBench (Ouyang et al., 2025) and TritonBench (Li et al., 2025a) providing standardized workloads for evaluating LLM-generated kernels. This growing interest is motivated by the need for optimization outpacing traditional compiler capabilities for rapidly developing hardware. However, establishing kernel correctness through empirical testing alone is insufficient, particularly for concurrency properties such as data races and memory barrier synchronization, whose behavior can depend on subtle execution semantics (Betts et al., 2015), with modern GPU architectures exasperating this issue.

Newer architectures like NVIDIA Corporation (2023) Hopper and Blackwell introduce a substantially more asynchronous execution paradigm than earlier generations of NVIDIA hardware. For example, warp specialization allows warps (collections of threads) to take specialized producer and consumer roles, overlapping background data movement through the Tensor Memory Accelerator (TMA) with computation. Realizing these gains requires coordinating asynchronous memory transfers, computation, and hardware-backed synchronization across multiple pipeline stages, creating a substantially larger space of possible execution states in which concurrency bugs, deadlocks, and performance-destroying stalls can be difficult to expose through empirical test suites alone (Soi et al., 2026).

Additionally, the most effective use of these hardware features depends strongly on the structure of each kernel: warp specialization and asynchronous data movement may require different producer-consumer decompositions, synchronization strategies, and memory layouts across kernels. This makes fixed optimization rules or autotuning difficult to apply uniformly. LLMs offer a flexible alternative, adapting expert design patterns to each kernel, exploring qualitatively different implementations, and

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.

iteratively refining them using profiler and debugger feedback (Ouyang et al., 2025; Wei et al., 2025; Zhang et al., 2025).

However, sustained reasoning about complex run-time semantics of GPU kernels remains a limitation of modern LLM agents (Section 4.2). To address this gap, we equip agents with Z3-based formal-verification tools that act as an interactive reasoning aid, drawing on recent work (Ospanov et al., 2026) exploring interactive settings in which LLMs work with formal theorem provers. Rather than relying solely on empirical test cases, agents can use these symbolic tools to verify properties such as race freedom, bank-conflict freedom, and reduction locality, providing formal evidence for correctness and hardware-aware optimization decisions.

We target *Gluon* because it exposes the hardware-specific controls needed for modern GPU optimization while retaining structured, tile-level semantics that are amenable to formal reasoning. Gluon sits below Triton (Tillet et al., 2019) in the compilation stack (Appendix A), preserving Triton's compiler infrastructure (Lattner et al., 2021) while exposing features such as explicit memory layouts and warp specialization to the programmer. This provides enough control to optimize asynchronous hardware primitives while leaving mechanical low-level lowering to the compiler backend.

In summary, this paper makes the following contributions:

- **Interactive profiling interface:** agents access systems engineering tools such as profilers, debuggers, and compilers to optimize GPU kernels.
- **Theorem provers as tools:** LLM agents use interactive feedback from formal theorem proving tools (via our Gluon-to-Z3 library) for verifying correctness of kernels.
- **Real-world performance testing:** Focusing on optimizing entire models rather than isolated kernels (e.g., activation functions) across representative input shapes.

2 Related Work

Automated GPU Kernel Optimization. Prior work has pursued two complementary approaches to GPU kernel generation. The first trains models to propose better kernels. AutoTriton (Li et al., 2025b) and TritonRL (Woo et al., 2025) train models to generate Triton kernels. SynKer (Sun et al., 2026) builds validated training examples across multiple backends, while Dr. Kernel (Liu et al., 2026) examines how to make execution and performance rewards more reliable. This approach extends beyond Triton to direct CUDA generation in CUDA Agent (Dai et al., 2026).

A second approach, which includes Glueball, sidesteps model training and instead develops the optimization harness that guides its revisions for a particular kernel and hardware pairing. The harness determines which tests, compiler diagnostics, and hardware measurements the model sees after each attempt. Hawkeye uses hardware feedback to guide optimization across architectures (Tschand et al., 2026), while Kernel-Smith evolves a population of candidates using compilation and performance feedback (Du et al., 2026). Astra distributes planning, generation, profiling, and testing among specialized agents (Wei et al., 2025). KernelCraft explores feedback-driven generation for unfamiliar accelerator instruction sets (Nie et al., 2026).

In addition to performance feedback, the intermediate checks must also judge whether a revised kernel is correct. Tests may miss changes to memory access or synchronization, while program verification can verify these properties with formal proof. Previous work in GPUVerify (Betts et al., 2015) builds a static verifier for CUDA and OpenCL kernels that checks for data races and barrier divergence by reducing parallel execution to a sequential verification problem. In Glueball, we bring formal checks into its Gluon optimization loop through symbolic reasoning over supported Triton IR computations and targeted Z3 models for selected Gluon hardware properties. The resulting proofs inform correctness judgments, while counterexamples guide revisions alongside test results.

Benchmarks and End-to-End Evaluation. KernelBench (Ouyang et al., 2025) established a widely used protocol for evaluating generated GPU kernels. On PyTorch workloads, it considers a generation successful when it passes predefined tests for output agreement with the reference and exceeds a chosen speedup threshold. TritonBench (Li et al., 2025a) applies a similar approach to Triton kernels, including examples drawn from real repositories. TritonGym (Guan et al., 2026) extends this setting to multi-step agents, providing compilation, testing, and profiling feedback during

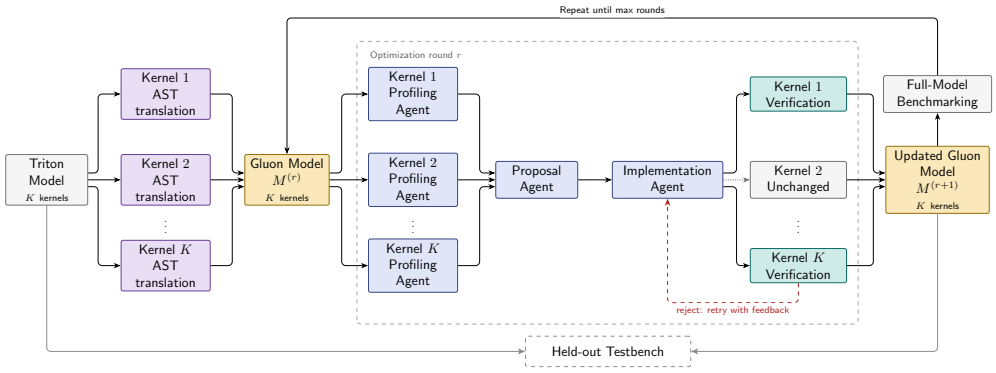

Figure 1: **System Architecture.** The optimization loop iteratively profiles, proposes, and applies kernel transformations. Every candidate variant is formally and empirically validated by the correctness pipeline prior to benchmarking and committing updates.

refinement, while retaining test-based correctness and kernel-level speed as evaluation criteria. These benchmarks make kernel-generation systems easier to compare, but their correctness results depend on the supplied tests and their performance results primarily concern isolated kernels.

RealisticTritonBench (Huang et al., 2026) extends the study beyond isolated kernel improvements by deriving tasks from real-world pull requests and reintegrating generated kernels back into those code repositories. However, static public benchmarks remain problematic: they may appear in model training data or be exploited during optimization, making measured gains difficult to generalize. Models may even exploit benchmark evaluation limitations. For example, Dr. Kernel (Liu et al., 2026) reports observing reward hacking generations that appear fast without actually running the intended Triton kernel. To address these limitations, we report results of end-to-end integration of our Gluon kernels into real-world AI models, while standalone Liger Kernels (Hsu et al., 2024) test individual optimizations against competitive Triton baselines. We introduce randomized inputs each round while keeping future cases hidden to give a useful end-to-end performance signal during optimization, and evaluate final kernels on held-out shapes and configurations.

3 Method

We jointly optimize the kernels of each model to maximize end-to-end performance. This is important for architectures such as Mixture-of-Experts (MoE) (Shazeer et al., 2017), where intertwined execution graphs mean that per-kernel improvements may not translate directly to end-to-end speedups. In particular, Amdahl's Law (Amdahl, 1967) limits the overall benefit of optimizing kernels that contribute only a small fraction of total execution time. While changes to one kernel can also interact with neighboring kernels through memory layouts, numerical precision, or synchronization. As shown in Figure 1, our system translates the reference Triton kernels to Gluon and iteratively optimizes, verifies, and benchmarks them over a fixed number of rounds, committing each round's optimizations for the next round, with each agent being equipped with a specialized toolset (Appendix B).

3.1 Initial Unoptimized Translation

Each reference Triton kernel is independently and mechanically translated into Gluon using Abstract Syntax Tree (AST) transformations that preserve functional behavior but do not optimize hardware-specific decisions. The resulting kernels serve as the initial state of the search, leaving hardware features such as TMA unused.

3.2 Hardware-Aware Optimization Pipeline

Each optimization round (Figure 1) coordinates four stages: 1) profiling, 2) proposal generation, 3) implementation and verification, and 4) benchmarking and committing the optimization.

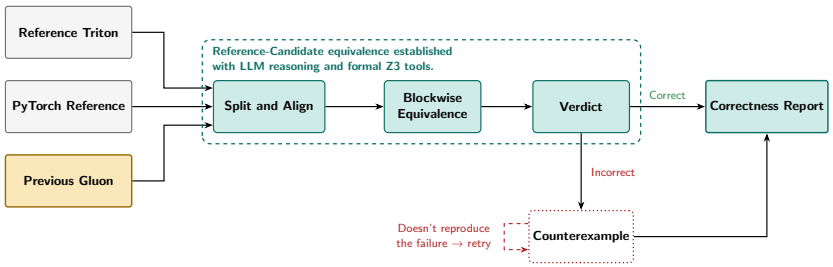

Figure 2: **Correctness Pipeline.** Reference and candidate programs are segmented and aligned into corresponding regions, which are checked for equivalence and program properties using LLM reasoning and Z3-based formal tools. Detected failures are converted into concrete witnesses and used to guide retry or produce the final correctness verdict.

Profiling. At the start of each round, an instrumented pass over the complete model measures the run-time contribution of each constituent kernel. Profiling agents use this contribution to direct their profiling effort, focusing more detailed analysis on kernels with greater impact on end-to-end performance. Agents analyze kernels in parallel using their source, historical benchmark results, and the target hardware profile. At the *coarse level*, agents use NVIDIA Nsight Compute (`ncu`) to track metrics such as SM occupancy, memory-bandwidth utilization, and register pressure. At the *fine level*, agents can inject source-level probes to isolate application-specific bottlenecks and disentangle whether a kernel is stalled on memory transfers, matrix computation, or shared-memory bank conflicts.

Proposal Generation. A centralized proposal agent combines profiling reports, historical outcomes, and the current model state to produce an ordered portfolio of candidate transformations, prioritizing transformations with the greatest expected impact on whole-model performance. A proposal may modify one or more kernels and can include multiple sequential steps. To ground its decisions, the agent can inspect Triton and Gluon intermediate representations, PTX (Parallel Thread Execution) assembly, and create and evaluate micro-benchmarks.

Implementation and Verification. The implementation agent executes the highest-ranked proposal one step at a time, applying targeted source changes to the affected kernel. Each step is checked for successful compilation, valid function signatures, and numerical correctness (both via agent-devised empirical tests and use of a Floating-Point Sanitizer (FpSan) tool), followed by **blockwise formal verification** (Section 3.3). When a check fails, the implementation agent receives the compiler diagnostics, execution tracebacks, numerical failures, and Z3 counterexamples (Appendix D). The implementation agent can then revise the change and re-run the checks. Only changes that pass all checks are committed; failed changes leave the state of kernels affected by the proposal unchanged.

Benchmarking and Committing the Optimization. After all modified kernels in a proposal have passed their per-kernel verification gates, the resulting model is executed using the current round's benchmark suite. The benchmark inputs are hidden from the agents, and the suite is expanded each round with newly randomized entries to reduce overfitting to previously observed shapes and inputs. If the integrated model fails a correctness check, the proposal is discarded; otherwise, its whole-model run time and speedup relative to the Triton baseline are recorded, and the resulting kernels become the starting point for the next optimization round.

3.3 Formal Tools-Guided LLM Kernel Verification

In an ideal world, each optimization would be formally verified to preserve program semantics, up to allowable floating-point precision differences. In practice, program equivalence is undecidable and symbolic state spaces can grow rapidly, particularly for GPU kernels orchestrating thousands of worker threads. These challenges are especially relevant to Gluon, which exposes low-level hardware primitives such as asynchronous copies, tensor-core accesses, and memory barriers.

We therefore use a hybrid verification approach combining **symbolic execution**, **LLM reasoning**, and **targeted Z3 models**. Symbolic execution provides formal reasoning for computations that can be represented directly, while the LLM reasons across semantics that fall outside this representation and uses Z3 explicitly when a hardware-level property requires a specialized model.

Symbolic Execution. Our symbolic execution model maps supported IR operations to Z3 expressions. Arithmetic operations are translated to corresponding Z3 operators, while supported mathematical operations use Z3 floating-point operators or uninterpreted functions (Appendix C). Input tensors are represented by symbolic values (instead of arrays in memory), and the interpreter follows SSA definition chains to construct symbolic expressions for derived values.

For example, applying the interpreter to the Triton softmax kernel in Appendix C.1 produces an expression of the form:

$$\text{softmax_output} = \frac{\exp(\text{load_row} - \text{maxnumf_row_minus_max})}{\text{addf_denominator}}$$

with opaque symbol names prepended to the input value they operate on. Note that this reflects the mathematical formula $\text{softmax}(x_i) = \frac{\exp(x_i - x_{\max})}{\sum_j \exp(x_j - x_{\max})}$, but was extracted from the Intermediate Representation (IR). We defer to Appendix C for a full list of supported operations and their Z3 representations.

Opaque Operations and Symbolic Regions. Operations whose internal behavior is not modeled symbolically—including reductions, memory accesses, synchronization primitives, and loop structures—are treated as *opaque operations*. Their results are represented as symbolic values whose internal derivation is left unspecified. These operations partition the kernel into *symbolic regions*: contiguous sequences of supported operations that can be translated into closed-form symbolic expressions. Z3 provides formal reasoning within each region, while the LLM reasons about dependencies across opaque boundaries.

Our Gluon-to-Z3 Library. For Gluon kernels, the LLM can additionally use our *Gluon-to-Z3 Library*, specifically designed to model hardware-level behavior that cannot be captured by value-level symbolic execution. These models allow Z3 to check properties such as asynchronous race freedom, barrier ordering, and shared-memory correctness (Appendix D).

3.4 Correctness Pipeline

Figure 2 shows how these components are combined into the correctness pipeline. We next describe each component in detail.

Reference-Candidate Equivalence. The pipeline first compares an optimized kernel against its reference PyTorch and Triton implementations. Optimizations such as tile re-blocking, warp specialization, and loop unrolling can change the structure of the candidate IR, so corresponding symbolic regions may not align directly. A *Splitter Agent* identifies symbolic regions and an *Alignment Agent* aligns their inputs and outputs. An *Equivalence Agent* then checks each aligned pair using their Z3 representations and LLM reasoning. For Gluon-specific hardware properties, the LLM instead constructs the relevant formal model using the Gluon-to-Z3 Library Appendix C and invokes Z3 to check the property Appendix D.

Counterexample Synthesis. When a formal check fails, the pipeline extracts the available evidence and constructs a concrete explanation. Equivalence failures provide the divergent symbolic results, while hardware-property failures can provide Z3 counterexamples describing violating assignments or executions, such as thread coordinates, memory addresses, or conflicting events Appendix D.

Verdict Agent. This agent combines blockwise equivalence results with the kernel source to approve the kernel or identify the failure mechanism. It can execute targeted code snippets or interactive debugger sessions to synthesize a runnable harness with the relevant inputs and launch parameters. Executing this harness on physical GPU hardware provides a concrete witness that can be used to confirm the failure and guide subsequent repair.

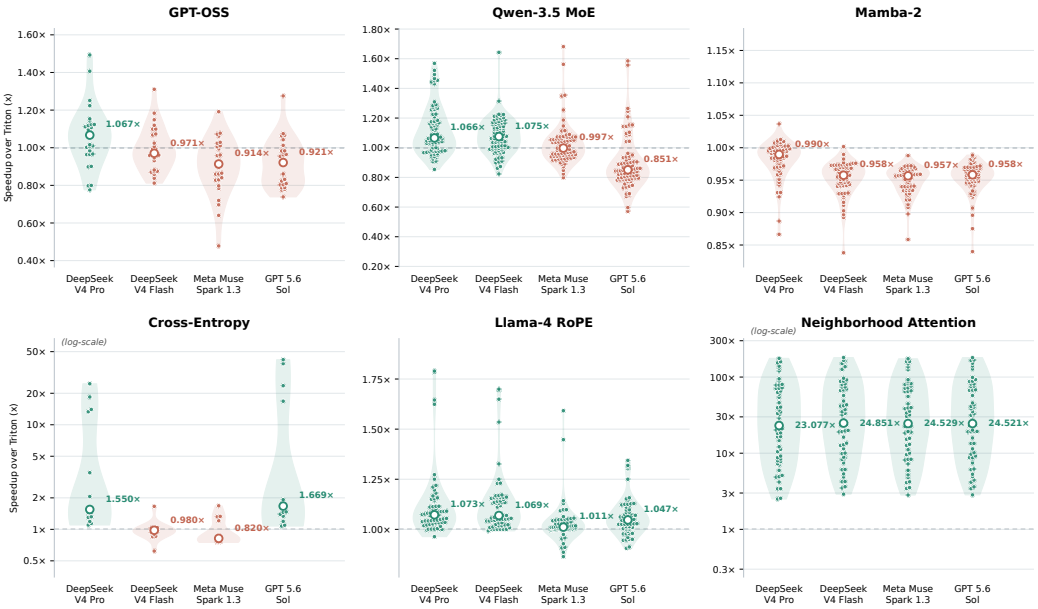

Figure 3: **Best-round speedup across input shapes.** Each violin shows the density of per-input shape speedups (models evaluated in an end-to-end configuration). The median is denoted by the circle, its color indicates if it is at or above parity (green) or below parity (red). Each point denotes an input shape configuration. Note that some panels use log-scale.

4 Experiments

We evaluate Glueball on two objectives: end-to-end performance optimization (across a budget of 6 rounds) and Gluon kernel correctness verification on an NVIDIA DGX Spark with a GB10 chip. We instantiate our pipeline with agents powered by Deepseek V4 Pro & Flash, Meta Muse Spark 1.3 and GPT 5.6 Sol models, enabling a comparison of performance across models. Complete experiment data can be found in Appendix E.

4.1 Optimization Performance Evaluation

Our evaluation consists of three model workloads (*GPT-OSS*, *Qwen-3.5 MoE* and *Mamba-2*), and three standalone kernels (*Neighborhood-Attention*, *Cross-Entropy*, and *Llama-4 RoPE* from Liger Kernels), chosen to expose complementary optimization challenges including Tensor Core utilization, asynchronous data movement, latency hiding, and shared-memory management. Our optimization goal is to minimize end-to-end model (and standalone kernel) execution time across a wide range of input shapes. Optimization performance data can be found in Figure 3 and the kernels optimized jointly for each model workload are detailed in Appendix E.1.

GPT-OSS: optimizing the data path to Tensor Cores. DeepSeek V4 Pro generates a kernel improving end-to-end performance over the Triton baseline, highlighting the importance of optimizing the data path alongside the computation. It removes several unnecessary tensor transformations from the Triton reference. The Query/Key/Value transpose is encoded in the TMA descriptor rather than performed as a physical copy and Grouped-Query Attention (GQA) is handled by computing the corresponding Key-Value head index in-kernel rather than materializing eight copies of each Key-Value head. Additionally, TMA handles out-of-bounds accesses without padding tensors and the output is written directly in its final layout rather than transposed and copied afterwards. These changes reduce memory traffic and allow multiple query heads to reuse the same underlying Key-Value data.

DeepSeek V4 Flash and Meta Muse Spark 1.3 instead focus on improving the `_attn_fwd` kernel's arithmetic path, replacing TF32 with lower-precision MMA operations, as hardware support often makes BF16 and FP16 operations faster. However, DeepSeek V4 Flash's lower precision optimization

Model	Generic Correctness Suite				Synchronization Correctness Suite			
	Detected (of 24) ↑		False Positives (of 26) ↓		Detected (of 10) ↑		False Positives (of 12) ↓	
	Baseline	Glueball (Δ)	Baseline	Glueball (Δ)	Baseline	Glueball (Δ)	Baseline	Glueball (Δ)
DeepSeek V4 Pro	21	+3	0	0	10	0	7	-5
DeepSeek V4 Flash	22	+2	0	0	10	0	1	-1
Meta Muse Spark 1.3	22	+2	0	0	9	0	3	-3
GPT 5.6 Sol	21	+2	0	0	9	+1	1	-1

Figure 4: **Correctness Test Suite results**. The Glueball correctness pipeline improves bug detection across all four models tested while maintaining fewer or equal amounts of false positives across both test suites. Results on the synchronization suite highlight the benefit of formal reasoning tools for exposing asynchronous execution errors that are difficult to infer from source code alone.

changes the MMA instruction shape necessitating explicit `convert_layout` operations, incurring substantial overhead. Interestingly, GPT 5.6 Sol chooses to optimize performance over the arithmetic path, however, it is mistakenly attempts to use `tcgen05` and Tensor Memory, which are not available on the test machine, despite having access to the hardware configuration.

Qwen-3.5 MoE: mapping expert routing to TMA. Qwen-3.5 MoE illustrates how agents can express the existing algorithm from the Triton reference more effectively on the hardware. The DeepSeek V4 Pro- and Flash-generated kernels preserve the reference algorithm and numerical formulation, but replace compiler-generated data movement with explicit Gluon implementations. The key opportunity is the irregular access pattern introduced by expert routing. While the Triton kernel gathers routed token rows using masked pointer loads, both DeepSeek V4 Pro and DeepSeek V4 Flash use TMA's gather operation to move those rows directly into shared memory. The expert-specific weight tiles, which have regular layouts, are transferred separately using contiguous TMA, matching each access pattern to a corresponding hardware data-movement mechanism. GPT 5.6 Sol optimization rounds repeatedly choose to optimize the `_fused_down_proj_kernel`, however the proposed changes it applies reduce performance. Thus, it's optimization effort is spent on recovering previous performance, with `_fused_up_proj_swiglu_kernel` remaining untouched before exhausting the total round budget.

Mamba-2: competing against an already strong compiler schedule. Mamba-2's `chunk_scan` presents a different challenge: improving on an already strong Triton software pipeline. All agents explore several valid pipeline structures, varying stage counts and buffering, eventually eliminating register spills, and substantially reducing the number of executed instructions. However, these improvements do not translate into lower run time: the resulting kernels remain slower than the Triton baselines because often its additional buffering and synchronization consumes more resources without providing sufficient operation-overlap to compensate. Finding a better schedule for an already well-scheduled kernel requires exploring a larger space of pipeline configurations, suggesting that either more optimization rounds or stronger hardware-specific priors about effective pipeline structures could be necessary.

Standalone kernels. Neighborhood attention demonstrates that the optimization search can improve the underlying algorithm, rather than only its hardware utilization. The reference Triton implementation materializes the full $L \times L$ attention matrix, whereas all optimized Gluon kernels directly traverse the local attention window, reducing the relevant work from $O(L^2D)$ to $O(LWD)$, where L is the sequence length, W is the local attention window, and D is the hidden dimension. In contrast, the improvements by DeepSeek V4 Pro and GPT 5.6 Sol on the cross-entropy experiment primarily stem from reducing host-side launch overhead rather than from a substantially faster kernel, while remaining consequential for end-to-end performance. Llama-4 RoPE has little headroom because its elementwise, bandwidth-bound reference already comes close to saturating device memory bandwidth, but we observe that all Gluon implementation can nonetheless achieve the same.

(a) Pipelining bug

```
# Correct
cp.wait_group(num_buffers - 1)
perform_add(
    a_smem[read_idx % num_buffers],
    b_smem[read_idx % num_buffers],
)

# Bug
cp.wait_group(num_buffers)
perform_add(
    a_smem[read_idx % num_buffers],
    b_smem[read_idx % num_buffers],
)
```

(b) Symbolic model

```
def build(num_buffers, wait_arg):
    ctx = GluonVerificationContext()
    r0, w0 = ctx.map_async_copy_global_to_shared(
        0, src_addr=0x1000, dst_addr=0x100,
        size=4, label='copy0')
    ctx.map_async_commit_group(0)
    r1, w1 = ctx.map_async_copy_global_to_shared(
        0, src_addr=0x2000, dst_addr=0x200,
        size=4, label='copy1')
    ctx.map_async_commit_group(0)
    wait = ctx.map_async_wait_group(
        0, num_outstanding=wait_arg,
        label=f'wait({wait_arg})')
    load = ctx.map_load(
        0, 0x100, Space.SHARED,
        size=4, label='read buffer0')
    ctx.program_order(wait.id, load.id)
    return ctx

for wait_arg in [2, 1]:
    safe, model = build(
        2, wait_arg).check_async_race_freedom()
```

(c) Z3 verdict

```
wait_group(2):
DATA RACE detected:
copy0.smem_write
  STORE SHARED@0x100
  proxy=ASYNC_COPY
<-race->
read buffer0
  LOAD SHARED@0x100
  proxy=GENERIC

wait_group(1):
RACE-FREE: proven
```

(d) Pipeline witness

```
check_pipeline_stage_isolation(
    num_stages=2, max_inflight=2)

PIPELINE COLLISION:
producer and consumer share a slot
witness: [iter_p = 2, lag = 2]
-> (False, [iter_p = 2, lag = 2])

max_inflight=1:

STAGE-ISOLATED:
num_stages=2 never aliases
with up to 1 in-flight iteration
-> (True, None)
```

Figure 5: **Theorem-prover-guided debugging of a pipelining bug.** (a) The buggy kernel waits for too few completed asynchronous copies before reading from a shared-memory buffer. (b) The LLM translates the relevant operations into a symbolic model, (c) Z3 identifies the conflicting asynchronous write and shared-memory read, and (d) a pipeline-level check produces a concrete iteration witness showing that the producer and consumer can alias the same buffer slot.

Summary. Overall, optimization effectiveness varies across workloads according to the opportunities exposed. GPT-OSS and Qwen-3.5 MoE benefit from structured data-movement optimizations that better match operand delivery to hardware through explicit TMA use, while Mamba-2 shows the difficulty of discovering improvements in the larger scheduling space of software pipelining. Neighborhood attention further demonstrates that the search can identify algorithmic opportunities, reducing computational complexity rather than merely optimizing the existing implementation.

4.2 Correctness Pipeline Evaluation

We evaluate the correctness pipeline independently of the optimization results using two benchmark suites that target complementary aspects of Gluon kernel correctness. The *Generic Correctness Suite* covers common kernel-level bugs, while the *Synchronization Correctness Suite* targets asynchronous execution and synchronization semantics. For each suite, we compare each model with and without the full correctness pipeline. We count a defect as correctly detected only when the model identifies the bug and its failure mechanism.

Generic Correctness Suite. Our suite contains 50 cases drawn from seven Liger-Kernel operators, including 24 cases with introduced bugs and 26 semantics-preserving controls. The bugs cover masking, numerical stability, accumulator precision, memory addressing, and algorithmic computa-

tion. Figure 4 shows that the correctness pipeline improves detection across all four models while maintaining zero false positives on the controls.

Synchronization Correctness Suite. The synchronization suite contains 22 Gluon kernels, including 10 buggy cases and 12 controls, targeting asynchronous copies, TMA operations, pipelining, and warp specialization. As shown in Figure 4, the pipeline both recovers missed defects and substantially reduces false positives compared to the baseline, indicating that asynchronous execution semantics are difficult to reason about for current LLMs from source code alone.

We next examine a pipelining bug observed during correctness pipeline evaluation to illustrate how the correctness pipeline turns this reasoning into explicit evidence (Figure 5). The example kernel uses two shared-memory buffers in a circular pipeline: while one buffer is consumed, the next is refilled asynchronously. The consumer must therefore wait for the copy that populates the buffer it is about to read. The buggy `wait_group(num_buffers)` leaves this copy outstanding, whereas `wait_group(num_buffers - 1)` provides the required ordering. Rather than relying on the LLM to infer asynchronous execution semantics from source code alone, Figure 5 shows how the LLM agent utilizes theorem-prover checks to expose both the low-level race and a concrete pipeline-level witness.

Summary. Overall, the correctness pipeline evaluation results show that it improves LLMs' ability to identify both kernel bugs and their underlying failure mechanisms, while reducing false positives. By combining symbolic execution with theorem-prover checks, the pipeline provides explicit evidence that enables LLMs to reason about correctness properties that are difficult to infer reliably from kernel source alone.

5 Conclusion and Future Work

We present **Glueball**, the first system to integrate formal verification tools into LLM-assisted GPU kernel optimization. Glueball enables LLM agents to iteratively optimize Gluon kernels against real hardware while using formal tools to verify the correctness of their generated kernels. Across model workloads and standalone kernels, we find that agents can improve end-to-end performance by exploiting opportunities in data movement, hardware-specific execution, and algorithmic structure. In addition, Glueball demonstrates how LLM agents can use theorem-proving tools within the optimization loop to verify their optimizations, improving bug detection and reducing false positives: agents translate relevant kernel computations into symbolic representations, invoke Z3 to check equivalence and hardware-specific properties, and use counterexamples to identify and diagnose correctness failures. Overall, our results support integrating formal verification directly into agentic GPU optimization, complementing existing approaches based on hardware feedback and iterative search with explicit verification of the resulting kernels.

Multi-GPU and heterogeneous optimization. An important future direction is to extend our agentic generation and verification methodology to distributed kernels, following the multi-GPU workloads introduced by ParallelKernelBench (Chan et al., 2026), and to heterogeneous accelerator targets studied by KernelGenBench (Zang et al., 2026). Multi-GPU execution would introduce proof obligations of collective ordering, cross-rank synchronization, communication–computation overlap, and distributed race and deadlock conditions. These extensions should be evaluated through end-to-end distributed model throughput and cross-platform robustness, rather than isolated kernel or communication latency.

Constraint and statistics-guided search. Software-pipelining schedules, including those considered for Mamba-2, expose a large combinatorial optimization space. Future systems could extend our existing configuration and safety checks to constrain parameterized schedule families before generating concrete implementations, for example, excluding configurations that violate hardware-resource constraints. Agents could use these constraints, together with accumulated profiling results and verification feedback, to refine their optimization strategies across iterations. Statistical optimization methods could further guide this process by identifying promising configurations from previous evaluations. Combining these methods with agent reasoning could enable broader exploration with fewer costly LLM iterations and hardware evaluations.

AI USE STATEMENT

AI was used to both aid and polish writing. It was also used for retrieval and discovery of related work and prior methods. Additionally, AI was used for ideation and coding agents for implementation.

REPRODUCIBILITY STATEMENT

Upon publication, we plan to release our code, benchmark suites, and experiment results in open-source.

REFERENCES

Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, volume 30, pp. 483–485, Atlantic City, NJ, 1967. AFIPS Press.

Adam Betts, Nathan Chong, Alastair F Donaldson, Jeroen Ketema, Shaz Qadeer, Paul Thomson, and John Wickerson. The design and implementation of a verification technique for gpu kernels. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(3):1–49, 2015.

Willy Chan, Nathan Paek, Simon Guo, Simran Arora, and Daniel Y Fu. Parallelkernelbench: Can llms write fast multi-gpu kernels? In *Deep Learning for Code: Towards Human-Centered Coding Agents*, 2026.

Weinan Dai, Hanlin Wu, Qiying Yu, Huan-ang Gao, Jiahao Li, Chengquan Jiang, Weiqiang Lou, Yufan Song, Hongli Yu, Jiaze Chen, et al. Cuda agent: Large-scale agentic rl for high-performance cuda kernel generation. *arXiv preprint arXiv:2602.24286*, 2026.

He Du, Qiming Ge, Jiakai Hu, Aijun Yang, Zheng Cai, Zixian Huang, Sheng Yuan, Qinxiu Cheng, Xinchen Xie, Yicheng Chen, et al. Kernel-smith: A unified recipe for evolutionary kernel optimization. *arXiv preprint arXiv:2603.28342*, 2026.

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 *Forty-third International Conference on Machine Learning*, 2026.

Pin-Lun Hsu, Yun Dai, Vignesh Kothapalli, Qingquan Song, Shao Tang, Siyu Zhu, Steven Shimizu, Shivam Sahni, Haowen Ning, and Yanning Chen. Liger kernel: Efficient triton kernels for llm training. *arXiv preprint arXiv:2410.10989*, 2024.

Jinjun Huang, Zhongzhen Wen, Tongtong Xu, Meng Yan, Xin Xia, and Zhongxin Liu. Realistictritonbench: A benchmark for triton-kernel generation in real-world ai frameworks. *arXiv preprint arXiv:2608.12004*, 2026.

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM international symposium on code generation and optimization (CGO)*, pp. 2–14. IEEE, 2021.

Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, WangHaojie WangHaojie, Jianrong Wang, Xu Han, et al. Tritonbench: Benchmarking large language model capabilities for generating triton operators. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 23053–23066, 2025a.

Shangzhan Li, Zefan Wang, Ye He, Yuxuan Li, Qi Shi, Jianling Li, Yonggang Hu, Wanxiang Che, Xu Han, Zhiyuan Liu, et al. Autotriton: Automatic triton programming with reinforcement learning in llms. *arXiv preprint arXiv:2507.05687*, 2025b.

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. *arXiv preprint arXiv:2602.05885*, 2026.

Jiayi Nie, Haoran Wu, Yao Lai, Zeyu Cao, Cheng Zhang, Binglei Lou, Erwei Wang, Jianyi Cheng, Timothy M Jones, Robert Mullins, et al. Kernelcraft: Benchmarking for agentic close-to-metal kernel generation on emerging hardware. *arXiv preprint arXiv:2603.08721*, 2026.

NVIDIA Corporation. NVIDIA H100 Tensor Core GPU Architecture. White paper, NVIDIA Corporation, 2023. URL https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c.

Azim Ospanov, Zijin Feng, Jiacheng Sun, Haoli Bai, Xin Shen, and Farzan Farnia. Hermes: Towards efficient and verifiable mathematical reasoning in llms, 2026. URL https://arxiv.org/abs/2511.18760.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*, 2025.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer, 2017. URL https://arxiv.org/abs/1701.06538.

Rupanshu Soi, Rohan Yadav, Fredrik Kjolstad, Alex Aiken, Maryam Mehri Dehnavi, Michael Garland, and Michael Bauer. Optimal software pipelining and warp specialization for tensor core {GPUs}. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, pp. 1875–1890, 2026.

Qi Sun, Robert Tjarko Lange, and Alex L Zhang. Synker: Synthesize, kernelize, reinforce-teaching gpu kernel generation to small language models. In *Deep Learning for Code: Towards Human-Centered Coding Agents*, 2026.

Philippe Tillet, Hsiang-Tsung 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*, pp. 10–19, 2019.

Arya Tschand, Kesavan Ramakrishnan, Alexander Ingare, Simon Guo, Jeffrey Jian Ma, Zishen Wan, Simran Arora, Azalia Mirhoseini, and Vijay Janapa Reddi. Hawkeye: Hardware-aware gpu kernel optimization with minimal supervision. In *Deep Learning for Code: Towards Human-Centered Coding Agents*, 2026.

Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. Astra: A multi-agent system for gpu kernel performance optimization. *arXiv preprint arXiv:2509.07506*, 2025.

Jiin Woo, Shaowei Zhu, Allen Nie, Zhen Jia, Yida Wang, and Youngsuk Park. Tritonrl: Training llms to think and code triton without cheating. *arXiv preprint arXiv:2510.17891*, 2025.

Peiyu Zang, Jian Tao, Jialing Zhang, Yichen Yuan, Wentao Zhang, Guang Liu, and Yonghua Lin. Kernelgenbench: Can llms and agents write efficient kernels across operator sources and hardware platforms?, 2026. URL https://arxiv.org/abs/2607.27231.

Zijian Zhang, Rong Wang, Shiyang Li, Yuebo Luo, Mingyi Hong, and Caiwen Ding. Cudaforge: An agent framework with hardware feedback for cuda kernel optimization. *arXiv preprint arXiv:2511.01884*, 2025.

A TRITON AND GLUON COMPILATION PIPELINES

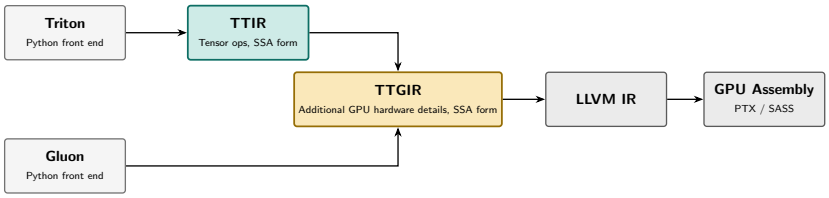

Figure 6: Triton and Gluon compilation pipelines.

Both Triton and Gluon lower Python kernels through Triton's intermediate representations to GPU assembly (Figure 6). Triton first lowers the Python program to TTIR (Triton Intermediate Representation), which represents tensor operations in SSA (Static Single Assignment) form, and then to TTGIR (Triton GPU Intermediate Representation), which introduces additional GPU hardware details before lowering through LLVM IR to GPU assembly.

Gluon provides a Python front-end directly for TTGIR, allowing hardware-specific details to be expressed explicitly while reusing Triton's downstream compilation pipeline. Compared to GPU programming languages such as CUDA, Gluon retains Triton's blockwise programming model.

B TOOL AND CLI UTILITY INVENTORY

Every agent in Figure 1 is equipped with a role-specific bundle of toolsets (Table 1). We distinguish between toolsets specifically designed for this work (Appendix B.1) and general command-line programs and packages used during kernel development (Appendix B.2) by a systems/kernel engineer. All tools are isolated with `bubblewrap`, providing a read-only view of the filesystem outside the agent's own workspace and no network access.

Agent role(s)	Capabilities
Gluon translation	**IR** **Z3** **Doc** **Gluon seeding**
Gen.-benchmark, Gen.-launch	**Bench** **Debug** **NCU**
Profiling	**Bench** **Debug** **NCU** **Proton**
Proposal, Optimization	**IR** **Z3** **Bench** **Doc** **Debug** **NCU** **Proton** **Gluon seeding**
Splitter, Mapper	**Z3** **Doc** **Debug**
Equivalence, Verdict	**IR** **Z3** **Doc** **Debug** **Gluon seeding**

Table 1: **Capability categories bound into each agent role's tool bundle.** **IR** = Triton/TTIR/TTGIR extraction; **Z3** = symbolic execution and formal program property verification; **Bench** = self-generated-harness creation/execution; **Doc** = scoped documentation search; **Debug**/**NCU**/**Proton** = CLI tool wrappers detailed in Appendix B.2.

B.1 SPECIALIZED TOOLS

The tools below allow agents to perform their specialized roles and go beyond what is achievable via command line tools alone.

Tool	Description
IR extraction and compilation	
`get_triton_ir`	Compile a Triton kernel and return its TTIR and TTGIR lowering.
`get_gluon_ir`	Compile a Gluon kernel and return its TTGIR lowering.

Tool	Description
`pytorch_to_compiled_triton`	Capture the Triton kernel generated by PyTorch's TorchInductor backend for a `torch.compile`'d function, when no hand-written reference Triton kernel is available.
Symbolic execution and formal verification (Z3)	
`symbolize_triton`	Compile and execute a Triton block, extracting the Z3 symbolic representation (plus execution output) (Appendix C.1).
`symbolize_ttir`	Symbolically execute already-obtained TTIR into Z3 symbolic formulas, without recompiling.
`run_z3`	Execute arbitrary Python Z3 code; auto-imports the project's Gluon-to-Z3 library (Appendix D).
Benchmarking	
`prepare_benchmark`	Assemble a kernel and the kernel's launch code (contains auto-tuning code and Gluon 'constexprs') (Section 3.1) into a runnable benchmark script.
`test_benchmark`	Run the auto-benchmark harness end-to-end, on every configured input, to validate launch code before finalizing it.
Documentation	
`read_doc_file`	Read one file from the project's reference documentation tree.
`query_documentation`	Delegate an open-ended documentation question to a dedicated documentation sub-agent.
Gluon seeding	
`translate_triton_source`	Mechanically translate a `@triton.jit` kernel into an equivalent, unoptimized `@gluon.jit` kernel via an AST-based translator, seeding every run's search; also invocable during later rounds, as a reference for correct Gluon API usage.

Table 2: Description of the tools making up the non-CLI toolsets.

B.2 General command line tools and wrappers

In addition to the tools above, every agent has access to a general-purpose shell (via `run_command`/`start_command`/ `check_command`/`stop_command`). This gives every agent standard text-processing tools (`grep`/`ripgrep`, `sed`, `awk`, arbitrary pipes), file read/write, and the ability to directly invoke `ncu`, `pdb`, `proton-viewer`, and arbitrary Python scripts. For the three capabilities below, rather than relying on an agent to assemble the right command invocation from scratch every time, we additionally provide utilities that help build scripts or commands an experienced engineer would use.

Capability	Description
Debug (interactive)	An interactive, multi-turn `pdb` session over a persistent subprocess, for a kernel. It allows for stepping through execution and inspecting intermediate values. A one-shot fallback (assemble a script and run it once, rather than stepping through interactively) is also available for quick comparisons.
`ncu` profiling	Assembles a validated NVIDIA Nsight Compute runner script and prints the exact `ncu` shell command to run it; also helps perform a differential, metric-by-metric comparison of two saved profiles from previous `ncu` runs.
Proton profiling	Assembles a script instrumented with Triton's own Proton profiler, at either coarse inter-kernel timing or cycle-accurate intra-kernel granularity, and provides a SQL-query interface over the resulting timeline trace.

Table 3: CLI-wrapped capabilities: wrappers around external programs (`pdb`, `ncu`, Proton) that already exist independently of this project.

C MLIR DIALECTS AND THEIR Z3 REPRESENTATION

The Z3 model of TTIR (Section 3.3) dispatches MLIR operations by dialect prefix to five handler modules; the same interpreter also processes TTGIR through the `ttg` handler. We use this interpreter to divide kernels into *symbolic regions*: contiguous regions of IR for which all operations admit an explicit symbolic interpretation. Operations that are not modeled by the interpreter are treated as *opaque operations*. Their results are represented by fresh symbolic values, which we refer to as *opaque inputs* when they serve as inputs to subsequent symbolic regions.

Every tensor value is modeled by a single representative scalar element, with no array or index theory involved. Consequently, the interpreter provides value-level symbolic reasoning rather than a complete model of tensor shapes, layouts, or memory behavior; such properties are handled indirectly through address arithmetic, explicit Z3 queries, or the Gluon-to-Z3 Library (Appendix D). Since TTIR and TTGIR use Static Single Assignment form, the formula for a supported value is obtained by walking its chain of definitions back to kernel arguments and opaque inputs. When an operation is not supported by the interpreter, the model does not guess its semantics; instead, it collapses the result to a fresh symbol.

The five handlers differ in how much semantic information they expose to the symbolic model. Arithmetic and supported mathematical operations contribute directly to symbolic expressions; tensor reshaping and pointer arithmetic can often be represented without expanding tensor state; and memory, reduction, collective, and control-flow operations are partially modeled or treated as opaque. Table 4 summarizes these behaviors and the resulting boundaries of symbolic regions. To make these distinctions concrete, Appendix C.1 works through a real kernel end-to-end, illustrating both how supported operations are translated into symbolic expressions and where the interpreter reaches its limits.

Table 4: MLIR dialects handled by the Z3 model of TTIR/TTGIR, the number of operations each handler dispatches on, and how each dialect is represented. An operation whose dialect prefix is not in this table is always collapsed to a fresh opaque symbol.

Dialect	**Ops**	**Representative ops**	**How it's modeled**
`arith`	51	`addf`, `cmpi`, `select`, `sitofp`, `bitcast`	Every operation maps directly to a native Z3 bit-vector or floating-point operator (e.g. `addf`→`fpAdd`, `divui`→`UDiv`)—an exact translation with no opaque terminals.
`math`	46	`exp`, `log`, `sin`, `sqrt`, `erf`, `ctlz`	Split by how well-behaved the operation is: numerically simple ops (`sqrt`, rounding, `fma`, `abs`, NaN/Inf tests) map to native Z3 floating-point builtins and are evaluated exactly; transcendentals (`exp`, `log`, trigonometric, `pow`) become *uninterpreted* Z3 functions—not numerically evaluated, but provably equal whenever applied to syntactically identical arguments, which suffices for elementwise equivalence checks.
`tt` (Triton core)	52	`load`/`store`, `addptr`, `splat`/`reshape` /`trans`, `reduce`, `dot`, `atomic_rmw`	A three-way split: pure reshaping/aliasing ops (`splat`, `broadcast`, `reshape`, `trans`, …) re-tag the same expression at zero cost; pointer arithmetic is concrete bit-vector math; and every memory access or collective op—`load`, `reduce`/`scan`, `dot`, atomics, `gather` – becomes a fresh opaque symbol, since what it actually computes is never modeled.

continued on next page

Dialect	Ops	Representative ops	How it's modeled
`scf`	12	`for`, `if`, `while`, `index_switch`	Approximated, not modeled exactly: `if` is exact (both branches interpreted and merged with Z3's own `If`); `for`/`forall`/ `parallel` run their body *once* symbolically, so loop-carried values become fresh symbols after that single pass rather than being unrolled; `while` is always opaque, since general loop termination is undecidable.
`ttg` (Triton GPU)	27	`local_load`/`store`, `convert_layout`, `async_copy_global_to_local`, `warp_specialize`	Lets the same interpreter process TTGIR, the dialect Gluon compiles to directly (Section 3.3): layout-conversion and reshape ops are aliasing, local-memory reads are opaque and writes are recorded as side effects, and only the *default* partition of `warp_specialize` is traced, since the other partitions run in parallel on different hardware and fall outside a single-threaded interpreter's model.

C.1 WORKED EXAMPLE: `SOFTMAX`

To make the symbolic execution mechanism concrete, this section runs the row-wise `softmax` kernel from Triton's tutorials through compilation to TTIR and then through the Z3 interpreter, via the `get_triton_ir` and `symbolize_ttir` tools (Appendix B) respectively.

Each kernel invocation loops over a slice of rows, and for every row: loads it, subtracts the row's max for numerical stability, exponentiates, normalizes by the row's sum, and writes the result back. Compiling one loop iteration to TTIR (location metadata and pointer-address bookkeeping elided for readability) gives:

```
scf.for %row_idx = %row_start to %n_rows step %row_step : i32 {
  %row         = tt.load %input_ptrs, %mask, %neg_inf :
  tensor<128x!tt.ptr<f32>>
  %row_max     = "tt.reduce"(%row) <{axis = 0}> ({
    ^bb0(%a: f32, %b: f32):
      %m = arith.maxnumf %a, %b : f32
      tt.reduce.return %m : f32
  }) : (tensor<128xf32>) -> f32
  %shifted     = arith.subf %row, %row_max_splat : tensor<128xf32>
  %numerator   = math.exp %shifted : tensor<128xf32>
  %denominator = "tt.reduce"(%numerator) <{axis = 0}> ({
    ^bb0(%a: f32, %b: f32):
      %s = arith.addf %a, %b : f32
      tt.reduce.return %s : f32
  }) : (tensor<128xf32>) -> f32
  %out         = arith.divf %numerator, %denominator_splat :
  tensor<128xf32>
  tt.store %output_ptrs, %out, %mask : tensor<128x!tt.ptr<f32>>
}
```

Interpreting this into Z3 and chasing the stored value's SSA def-use chain back to its terminals gives one closed-form formula for the whole block:

```
softmax_output = exp(load_row - maxnumf_row_minus_max) / addf_denominator
```

where `load_row`, `maxnumf_row_minus_max`, and `addf_denominator` are fresh, unconstrained Z3 symbols, and `exp` is the uninterpreted Z3 function standing in for `math.exp`.

This example demonstrates a real strength and two limitations of the Z3 interpreter at once:

– The elementwise arithmetic *around* the reductions (the subtraction and the final division) is captured exactly, so a candidate kernel that reorders or fuses these operations without changing their meaning still produces an identical formula.

– Both `tt.reduce` calls are opaque: the model proves the numerator is exactly "the row minus *some* value called the row max", but cannot verify that value, or the final sum, were actually computed as a correct max/sum over the row — a bug inside a reduction's own body (e.g. a wrong axis) is invisible to it.

  To address this limitation, the verifier treats each `tt.reduce` as an opaque operation and therefore separates the surrounding computation into symbolic regions. Within each region, Z3 provides the formal symbolic representation described above; the behavior of the reduction itself is represented by an opaque input. Reasoning about semantics that span the opaque boundary is then delegated to the LLM-based verification pipeline. This limits the amount of code for which the LLM must reason directly while avoiding an intractable expansion of the reduction into the SMT model.

– The surrounding `scf.for` is executed symbolically *once*. The formula above is therefore a per-row template, valid for a single grid-stride iteration. Thus, it cannot catch a bug that only manifests across iterations of the loop itself (e.g. wrong bounds, or a stride that causes rows to be skipped or revisited). Therefore, we rely on LLMs to reason across loop iterations.

This example illustrates the intended scope of the verifier: Z3 provides formal reasoning over supported symbolic regions, but does not attempt to model complete kernel semantics when operations fall outside that representation. The resulting decomposition concentrates formal verification where it is tractable and exposes the remaining semantic gaps explicitly for LLM-based reasoning.

D THE GLUON-TO-Z3 LIBRARY

The library models Gluon's explicit hardware constructs—threads, memory spaces, and the asynchronous copy, TMA (Tensor Memory Accelerator), and WGMMA (Warp Group Matrix Multiply-Accumulate) engines—as a bounded event graph: each hardware action becomes an event, ordered by two Z3 relations, *happens-before* (`HB`) and *program-order* (`PO`). A domain model (Table 5) defines the event vocabulary; a set of hardware-ordering *axioms* (Table 6) encodes real ordering rules as constraints on the graph; and a set of *theorems* (Table 7) checks a safety property by asking Z3 to satisfy its *negation*—UNSAT proves the property holds under every possible schedule, while SAT returns a concrete counter-example. Appendix D.1 works through exactly this mechanism end to end, on a realistic race.

Entity	Description
`Space`	Where an access happens: global memory, shared memory, registers, or (Blackwell) tensor memory.
`Proxy`	Which hardware engine issued an access: the generic load/store path, Ampere's asynchronous copy engine, the Hopper/Blackwell TMA engine, or the tensor core—PTX treats accesses issued by different proxies as unordered unless an explicit fence says otherwise.
`EventType`	The kind of bounded action an event represents: a load, a store, an atomic, a barrier arrival or wait, a proxy or memory-consistency fence, an MMA read or write, or a CTA-wide barrier.
`Event`	One bounded symbolic memory or synchronization action: which thread/CTA/cluster issued it, its space and proxy, its address (concrete or symbolic), its size, its position in program order, and its derived warp/lane.
`MBarrier`	Models an `mbarrier`: its address, how many arrivals it expects, and how many transaction bytes it expects (for TMA-signaled barriers).
`bank_of(addr)`	The shared-memory bank a (concrete or symbolic) address maps to.

Table 5: Domain model underlying the event graph. Hardware constants are also defined here: warp size, bank count and width, cache-line size, and each architecture's shared-memory capacity limit.

Axiom	Hardware rule encoded
Per-location sequential consistency `apply_sc_per_location()`	A sound simplification of full multi-copy-atomicity acyclicity: embeds happens-before into a single integer timeline, forcing it to be acyclic for every event pair.
CTA barrier `apply_cta_barrier(ids)`	Models `gl.barrier()`: for every same-CTA barrier pair, everything program-order-before the first barrier, on every participating thread, happens-before everything program-order-after the second.
Async-proxy fence `apply_fence_async_shared(id, cluster)`	Models `fence.proxy.async[.cluster]`, required before handing a shared buffer to TMA or the tensor core: prior generic-proxy shared writes on the fencing thread become happens-before-ordered relative to later async-proxy (TMA/tensor-core) accesses in the CTA (or cluster).
Mbarrier synchronization `apply_mbarrier_sync(barrier)`	Validates a barrier's arrival and transaction-byte accounting against its expected values. If short, adds *no* ordering edges and reports a deadlock (the wait blocks forever on real hardware); if satisfied, asserts happens-before from every arrival to every wait.
Manual ordering `assert_hb` / `assert_no_ordering`	Escape hatches to assert or deny a happens-before edge directly, for hardware relationships not covered by a mapper above, or for deliberately unsynchronized adversarial test cases.

Table 6: Axioms: hardware ordering rules encoded as constraints on the event graph. Each is applied explicitly by the code being verified, mirroring exactly which synchronization primitives that code actually issues.

Theorem	Property and mechanism
Async race freedom `check_async_race_freedom()`	For every event pair sharing an address and space with at least one write, asserts a race candidate: neither event happens-before the other. A same-thread generic-proxy write followed by an async-proxy (TMA/tensor-core) access additionally requires an intervening async-shared fence to count as visible—plain happens-before is not sufficient across a proxy boundary (Appendix D.1). SAT identifies a racing pair with a concrete model.
Bank-conflict freedom `check_bank_conflicts(ids)`	For same-warp, different-thread shared-memory accesses, two conflict iff they map to the same bank at different addresses (the same address is a free hardware broadcast, not a conflict). SAT over the disjunction of all pairwise conflict conditions.
Global coalescing `check_global_coalescing(ids)`	Introduces symbolic bounds on a warp's global accesses; SAT on either bound being misaligned to the cache-line size, or the span exceeding it, means the accesses cannot fit in one aligned, coalesced transaction.

Table 7: Theorems: each proves a safety property by attempting to satisfy its negation, returning `(is_safe, model)`.

D.1 Worked example: a race due to an omission of a fence

Gluon's explicit control over asynchronous engines is exactly where subtle hardware bugs hide, since the hardware imposes as little ordering as possible to maximize performance. This section walks through one such bug end-to-end (the code, the LLM's translation of it into the Gluon-to-Z3 library above, the counter-example Z3 returns, and how that counter-example becomes a concrete witness) using the exact tool output the harness produces.

1. The bug. In a warp-specialized, software-pipelined kernel, a producer warp finalizes a per-tile correction factor and writes it into a shared-memory scratch slot that a subsequent warpgroup MMA reads as one of its operands:

```
# l_smem is a shared-memory scratch buffer reused across pipeline stages.
l_smem[0] = l_i  # plain (generic-proxy) store

# BUG: no hopper.fence_async_shared() here. The tensor core reads l_smem
# through the *async* proxy, which is not guaranteed to observe a plain
# store without an intervening fence---even though, in program order,
# the store textually comes first.
acc = hopper.warpgroup_mma(a_smem, l_smem, acc)
```

Nothing here raises an error, and on some inputs and schedules the kernel may even produce correct output by chance—which is exactly why this class of bug is dangerous and why an empirical test suite alone is a weak defense against it (Section 3.3).

2. The translation. An LLM agent, working from Table 6/7 and the mapper signatures in Appendix D.2, maps the two operations above to events and asks for a race-freedom proof:

```
ctx = GluonVerificationContext()

# generic-proxy store: l_smem[0] = l_i
e_store = ctx.map_store(0, addr=0x100, space=Space.SHARED,
label="store_l_i")

# tensor-core read of the same address, via warpgroup_mma's l_smem
operand
reads, mma_write = ctx.map_warpgroup_mma(0, smem_addrs=0x100,
label="mma")

safe, model = ctx.check_async_race_freedom()
```

Note what is *absent*: no `map_fence_async_shared` call between the store and the MMA, because the kernel never issues one – the translation is a faithful mirror of the bug, not a fix for it.

3. The counter-example. Run through `run_z3` (Appendix B), this is the solver's actual output:

```
DATA RACE detected. Racing pairs consistent with this model:
  store_l_i(id=0, thread=0, STORE, SHARED@0x100, proxy=GENERIC)
    <-race-> mma.mma_read0(id=1, thread=0, MMA_READ, SHARED@0x100,
    proxy=TENSOR_CORE)
```

`check_async_race_freedom` is SAT: even though both events are issued by the same thread in program order, the proxy-visibility refinement in Table 7 means program order alone does not establish happens-before across a generic-to-async proxy boundary. The model names the exact two events, their addresses, and their proxies —enough on its own for the LLM to identify the missing `fence_async_shared()` call and where it belongs.

4. From counter-example to witness. A satisfying Z3 model is a proof artifact, not yet an empirical one: the verdict mechanism (Section 3.3) uses it, together with the qualitative failure it describes, to construct a concrete executable input that reproduces the same failure on real hardware. Here the model already pins down the exact racing address and event pair; in a kernel where that address is a symbolic function of, say, a pipeline-stage or tile index, Z3's satisfying assignment additionally fixes concrete values for those symbols, which translate directly into concrete launch parameters (which tile, which pipeline depth) that reproduce the identical address collision. Re-running the kernel under those parameters—optionally using the interactive debugger or `ncu`'s own race-detection instrumentation (Appendix B.2)—lets the LLM empirically confirm the race before treating a fix as validated, rather than trusting the symbolic proof alone.

D.2 Layout Algebra and Static Constraint Checks

The remaining part of the Gluon-to-Z3 library checks properties, for example those that depend on Gluon's *layouts* (the explicit mapping from a logical tensor coordinate to a physical register/lane/warp/address), rather than on event ordering.

`from_blocked_layout(size_per_thread, threads_per_warp, warps_per_cta, order, shape=None)`

Input the per-dimension extents of a `gl.BlockedLayout` at each hardware level (register, lane/thread, warp), and the dimension order (fastest-varying first).

Output a `LinearLayoutModel`: per-dimension basis vectors (register, lane, warp, block), combined by XOR over $GF(2)$ to recover a logical tensor coordinate—the same representation Triton's own compiler uses internally ("linear layouts").

`check_register_locality(layout, dim, num_warps=4)`

Input a `LinearLayoutModel`, a tensor dimension `dim`, and the CTA's warp count.

Output `(is_local, model)`. UNSAT proves `dim` is a pure function of the register index, independent of lane and warp—so a `gl.reduce` along that dimension needs zero cross-lane shuffles or shared-memory traffic.

`SwizzledSharedLayoutModel(vec, per_phase, max_phase, order)`

Input the four parameters of a `gl.SwizzledSharedLayout`.

Output a model of that layout's element-offset arithmetic (phase and XOR-swizzled column) as closed-form Z3 bit-vector expressions, for use by `check_shared_access_bank_conflicts` below.

`check_shared_access_bank_conflicts(addr_fn, num_lanes=32)`

Input a lane-to-address function, typically built from a `SwizzledSharedLayoutModel` above, and the number of lanes to check.

Output `(is_conflict_free, model)`. UNSAT proves that no two distinct lanes hit the same bank at different addresses, for all lane pairs at once.

`check_kernel_config(block_m, block_n, block_k, dtype_bytes=2, num_stages=1, arch="hopper", extra_smem_bytes=0)`

Input candidate block dimensions, element size, pipeline depth, target architecture, and any additional shared-memory the kernel reserves outside the pipelined buffers.

Output a static (solver-free) arithmetic check that the block dimensions are MMA/WGMMA-aligned and the per-stage shared-memory footprint fits the architecture's capacity—used to prune an autotuning search space before spending compile or GPU time on an infeasible configuration.

`check_pipeline_stage_isolation(num_stages, max_inflight=1, total_iters=100)`

Input the pipeline depth and how many stages may legitimately be in flight at once (a producer at iteration `iter_p` fills slot `iter_p % num_stages`; a consumer reads slot `(iter_p - lag) % num_stages` for any in-flight lag up to `max_inflight`).

Output `(is_isolated, model)`. SAT identifies a colliding iteration/lag pair—the producer overwrites a physical slot an in-flight consumer is still reading, which shows up as intermittent NaNs on real hardware.

`check_coalesced_stride(offset_fn, elem_bytes, base_addr=0, num_lanes=32)`

Input a lane-to-byte-offset function, the element size, the access's base address, and the number of lanes to check.

Output a check that adjacent lanes' offsets are exactly `elem_bytes` apart (contiguity) and that the base address is 16-byte-aligned (the `LDG.128` vectorized-load requirement); a violation of either indicates an uncoalesced or misaligned global access pattern.

D.3 Experiment Runtime and Total Cost

The following tables detail the wall-clock runtime and total token usage for each experiment-model pair.

Table 8: Optimization runs: total wall-clock time (hours).

Experiment	V4 Pro	V4 Flash	Muse 1.3	GPT-5.6 Sol
GPT-OSS	32	188	4	81
Qwen-3.5 MoE	21	17	5	2
Mamba-2	26	188	4	73
Cross-Entropy	11	36	4	72
Llama-4 RoPE	3	2	3	2
Neighborhood-Attention	18	10	3	2
Total	**111**	**441**	**23**	**232**

Table 9: Optimization runs: total tokens.

Experiment	V4 Pro	V4 Flash	Muse 1.3	GPT-5.6 Sol
GPT-OSS	50.0 M	121.1 M	21.6 M	11.8 M
Qwen-3.5 MoE	65.4 M	68.3 M	24.6 M	6.5 M
Mamba-2	86.1 M	146.0 M	26.2 M	10.8 M
Cross-Entropy	68.6 M	68.8 M	23.5 M	5.5 M
Llama-4 RoPE	20.8 M	23.9 M	9.7 M	6.9 M
Neighborhood-Attention	69.4 M	59.0 M	22.4 M	10.1 M
Total	**360.3 M**	**487.1 M**	**128.0 M**	**51.6 M**

E EXPERIMENT DATA

All experiments were conducted on an NVIDIA DGX Spark with a GB10 chip. In this section we give more details regarding the experiment setup and data collected.

E.1 KERNELS UNDER OPTIMIZATION

Table 10 lists, for each experiment, the Triton kernels Glueball was tasked with optimizing.

Experiment	#	Kernels
GPT-OSS	1	`_attn_fwd`
Qwen-3.5 MoE	2	`_fused_up_proj_swiglu_kernel` `_fused_down_proj_kernel`

Mamba-2	7	`_chunk_cumsum_fwd_kernel`
		`_chunk_state_fwd_kernel`
		`_state_passing_fwd_kernel`
		`_bmm_chunk_fwd_kernel`
		`_chunk_scan_fwd_kernel`
		`_layer_norm_fwd_1pass_kernel`
		`_selective_scan_update_kernel`

Cross-Entropy	1	`liger_cross_entropy_kernel`
Llama-4 RoPE	1	`_llama4_rope_kernel`

Neighborhood-Attention	5	`_fused_neighborhood_attention_qk_kernel`
		`_softmax_single_block_forward_kernel`
		`_softmax_multi_block_forward_kernel`
		`_neighborhood_mask_kernel`
		`_fused_neighborhood_attention_av_kernel`

Table 10: Kernels put through the Glueball harness.

E.2 OPTIMIZATION PERFORMANCE ACROSS ROUNDS

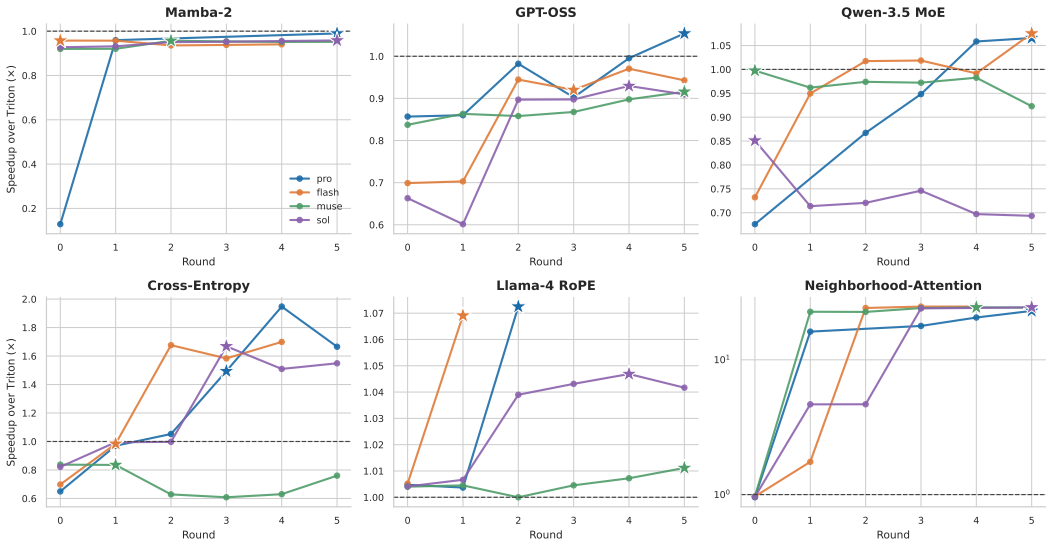

Figure 7: **End-to-End performance across rounds.** Star indicates the best round. Each trajectory may end before reaching the last round if the Proposal Agent fails to propose new optimizations. Each point corresponds to the median performance across a range of shapes, detailed in Appendix E.3.

Figure 7 shows that performance generally improves with more optimization rounds. We would like to note that although the star indicates the best round's performance for a given model-experiment pair, an exception is made for the Cross-Entropy experiment. For this kernel, ignored rows of logits are zeroed by the reference Triton kernel, whereas some Gluon kernels ignore this step to save on a large number of store operations. We deem this to be a valid and correct optimization, since the zeroeing performed by the Triton reference does not form part of the function's contract, the final output is identical, and this also does not affect gradients for the backwards pass. However, to ensure fairer performance comparison with the Triton baseline, we ignore the kernels produced by these optimization rounds, explaining the position of the star.

E.3 Per-Shape Measurements

The following tables report the reference Triton runtime and speedups per model across different experiments and input shapes. Each entry is the *median* time over as many repetitions as fit a fixed budget—500 ms for the per-kernel sweeps, 5000 ms for the whole-model experiments (GPT-OSS)—after a warmup of at least 25 iterations. Mamba's seven shape axes are abbreviated: `b` batch, `L` seqlen, `H` nheads, `P` headdim, `G` ngroups, `N` dstate, `C` chunk_size.

Table 11: GPT-OSS: per-shape reference runtime and speedup multipliers, ordered by reference runtime. Best round: DeepSeek V4 Pro r5, DeepSeek V4 Flash r3, Meta Muse Spark 1.3 r5, GPT-5.6 Sol r4.

Shape	Triton (ms)	V4 Pro	V4 Flash	Muse 1.3	5.6 Sol
small n_tokens=512	14.031	0.895×	0.699×	0.876×	0.671×
small n_tokens=1280	14.940	1.066×	0.889×	0.817×	0.823×
small n_tokens=768	15.505	0.757×	0.901×	1.070×	0.930×
small n_tokens=1024	16.883	1.093×	0.801×	0.950×	1.134×
small n_tokens=832	20.557	1.050×	0.862×	1.652×	1.279×
small n_tokens=2048	21.033	0.854×	0.999×	1.175×	0.997×
small n_tokens=3328	24.463	1.014×	0.973×	0.877×	0.959×
medium n_tokens=512	25.110	0.927×	0.727×	0.632×	1.026×
small n_tokens=3392	26.191	0.998×	1.068×	0.944×	1.018×
small n_tokens=3648	26.995	0.868×	0.881×	0.886×	0.895×
small n_tokens=4096	29.043	1.180×	0.822×	1.008×	0.909×
small n_tokens=3904	29.060	0.961×	0.788×	0.932×	0.880×
medium n_tokens=576	36.179	1.128×	1.402×	1.088×	1.313×
medium n_tokens=1152	38.520	0.966×	0.986×	0.873×	0.842×
medium n_tokens=1024	42.058	1.084×	0.968×	0.565×	1.035×
small n_tokens=8192	53.216	0.989×	0.809×	0.863×	0.859×
medium n_tokens=2048	54.215	1.287×	0.978×	0.983×	0.749×
medium n_tokens=2304	54.406	0.952×	0.742×	0.807×	0.695×
medium n_tokens=2496	55.398	1.142×	1.001×	0.912×	0.773×
medium n_tokens=2752	62.279	1.095×	1.039×	1.002×	0.860×
medium n_tokens=1984	64.295	1.184×	1.232×	0.947×	1.076×
medium n_tokens=3776	77.540	1.061×	0.819×	0.794×	0.979×
medium n_tokens=3520	78.188	1.136×	0.972×	1.038×	0.910×
medium n_tokens=4096	79.349	1.085×	0.762×	0.857×	0.816×
medium n_tokens=3968	87.111	1.083×	0.987×	0.985×	0.902×
small n_tokens=16384	120.580	1.036×	0.896×	0.788×	0.803×
medium n_tokens=8192	170.228	1.157×	0.853×	0.856×	0.817×
medium n_tokens=16384	337.670	1.124×	0.833×	0.736×	0.726×

Table 12: Qwen-3.5 MoE: per-shape reference runtime and speedup multipliers, ordered by reference runtime. Best round: DeepSeek V4 Pro r5, DeepSeek V4 Flash r5, Meta Muse Spark 1.3 r0, GPT-5.6 Sol r0.

Shape	Triton (ms)	V4 Pro	V4 Flash	Muse 1.3	5.6 Sol
T=4 H=2048 I=768 E=16 K=2	2.417	1.041×	0.666×	0.882×	1.179×
T=512 H=256 I=128 E=64 K=4	2.552	1.187×	1.023×	1.163×	1.132×
T=2048 H=256 I=128 E=16 K=2	2.661	0.782×	0.954×	1.213×	1.534×
T=512 H=256 I=128 E=32 K=4	2.697	1.416×	1.089×	0.943×	1.442×
T=4 H=256 I=128 E=32 K=4	2.782	0.788×	1.120×	0.878×	1.584×
T=2048 H=256 I=128 E=32 K=4	2.857	1.160×	1.164×	1.274×	1.287×
T=512 H=256 I=128 E=16 K=2	2.916	1.308×	1.077×	1.082×	1.263×
T=4 H=256 I=128 E=16 K=2	2.943	1.387×	0.813×	1.322×	1.714×
T=512 H=256 I=128 E=128 K=8	2.961	1.311×	1.383×	0.854×	1.216×
T=4 H=2048 I=768 E=32 K=4	2.971	1.142×	1.140×	0.767×	1.101×
T=4 H=256 I=128 E=64 K=4	3.092	1.315×	0.852×	1.043×	2.524×
T=4 H=256 I=128 E=128 K=8	3.132	1.116×	1.229×	1.047×	1.571×
T=2048 H=256 I=128 E=64 K=4	3.151	0.833×	1.337×	0.828×	1.210×
T=4 H=2048 I=768 E=128 K=8	3.249	0.962×	0.937×	0.818×	1.142×
T=4 H=4096 I=1536 E=16 K=2	3.311	0.930×	0.892×	0.969×	0.871×
T=2048 H=256 I=128 E=128 K=8	3.432	0.807×	0.975×	1.220×	1.384×
T=16384 H=256 I=128 E=16 K=2	3.453	0.931×	0.888×	0.967×	1.116×
T=512 H=2048 I=768 E=16 K=2	3.707	0.800×	0.883×	1.230×	0.946×
T=4 H=2048 I=768 E=64 K=4	3.952	0.964×	1.652×	1.049×	1.917×
T=2048 H=2048 I=768 E=16 K=2	5.074	1.152×	1.093×	0.793×	0.865×
T=4 H=4096 I=1536 E=64 K=4	5.161	0.819×	1.074×	1.145×	0.973×
T=512 H=2048 I=768 E=32 K=4	5.199	0.960×	0.975×	0.957×	1.353×
T=16384 H=256 I=128 E=32 K=4	5.318	1.068×	1.043×	1.030×	0.894×
T=4 H=4096 I=1536 E=32 K=4	5.482	0.927×	1.009×	1.356×	1.459×
T=16384 H=256 I=128 E=64 K=4	5.686	1.063×	0.869×	0.961×	1.111×
T=512 H=4096 I=1536 E=16 K=2	6.187	1.058×	1.082×	0.861×	1.049×
T=512 H=2048 I=768 E=64 K=4	6.505	1.164×	0.982×	1.219×	1.088×
T=4 H=4096 I=1536 E=128 K=8	7.369	0.965×	0.918×	0.928×	0.852×
T=1152 H=2048 I=768 E=64 K=4	7.680	1.111×	1.012×	0.910×	1.036×
T=2048 H=2048 I=768 E=32 K=4	7.902	0.956×	0.967×	0.887×	0.888×
T=16384 H=256 I=128 E=128 K=8	8.334	0.934×	1.039×	0.938×	0.929×
T=4 H=8192 I=3072 E=16 K=2	8.438	1.053×	1.050×	1.065×	0.955×
T=2048 H=2048 I=768 E=64 K=4	9.006	0.928×	1.053×	0.911×	0.822×
T=512 H=2048 I=768 E=128 K=8	9.446	0.951×	1.048×	0.919×	0.867×
T=2304 H=2048 I=768 E=64 K=4	10.176	0.972×	1.044×	0.885×	0.895×
T=2048 H=4096 I=1536 E=16 K=2	10.198	1.038×	0.994×	0.774×	0.655×
T=512 H=4096 I=1536 E=32 K=4	10.496	0.956×	0.915×	0.914×	0.968×
T=4 H=8192 I=3072 E=32 K=4	12.510	0.957×	1.036×	0.984×	0.952×
T=4 H=8192 I=3072 E=64 K=4	13.511	1.029×	1.116×	0.955×	0.969×
T=3392 H=2048 I=768 E=64 K=4	13.667	1.021×	1.009×	0.941×	0.860×
T=512 H=4096 I=1536 E=64 K=4	16.667	1.019×	0.984×	0.973×	0.968×
T=512 H=8192 I=3072 E=16 K=2	16.694	0.856×	0.881×	0.953×	0.867×
T=704 H=4096 I=1536 E=64 K=4	17.473	1.065×	0.957×	0.960×	0.914×
T=2048 H=2048 I=768 E=128 K=8	17.713	1.068×	1.162×	0.936×	0.818×
T=2176 H=2048 I=768 E=128 K=8	18.235	1.052×	1.151×	0.934×	0.816×
T=4 H=8192 I=3072 E=128 K=8	22.251	0.960×	1.023×	0.920×	0.917×
T=2944 H=2048 I=768 E=128 K=8	22.540	1.126×	1.207×	0.913×	0.855×
T=2048 H=4096 I=1536 E=32 K=4	23.812	1.160×	1.189×	1.043×	0.782×
T=1856 H=4096 I=1536 E=64 K=4	29.482	1.155×	1.077×	1.084×	0.916×
T=16384 H=2048 I=768 E=16 K=2	30.303	1.251×	1.189×	0.944×	0.888×

continued on next page

Shape	Triton (ms)	V4 Pro	V4 Flash	Muse 1.3	5.6 Sol
T=2048 H=4096 I=1536 E=64 K=4	30.717	1.201×	1.124×	1.059×	0.836×
T=512 H=4096 I=1536 E=128 K=8	32.181	1.057×	1.092×	1.013×	0.907×
T=512 H=8192 I=3072 E=32 K=4	33.730	0.923×	0.905×	1.008×	0.852×
T=704 H=4096 I=1536 E=128 K=8	35.021	1.107×	1.058×	1.093×	0.912×
T=1024 H=4096 I=1536 E=128 K=8	40.137	1.030×	1.003×	1.037×	0.881×
T=1088 H=4096 I=1536 E=128 K=8	40.799	1.017×	0.999×	1.044×	0.885×
T=2048 H=8192 I=3072 E=16 K=2	41.828	1.049×	1.135×	0.988×	0.807×
T=1152 H=4096 I=1536 E=128 K=8	42.337	1.028×	1.043×	1.055×	0.879×
T=1344 H=4096 I=1536 E=128 K=8	45.553	1.084×	1.095×	1.063×	0.870×
T=16384 H=2048 I=768 E=32 K=4	53.974	1.260×	1.171×	0.887×	0.836×
T=1856 H=4096 I=1536 E=128 K=8	54.116	1.148×	1.115×	1.095×	0.881×
T=16384 H=2048 I=768 E=64 K=4	54.951	1.283×	1.165×	0.903×	0.858×
T=2048 H=4096 I=1536 E=128 K=8	57.542	1.163×	1.110×	1.084×	0.860×
T=512 H=8192 I=3072 E=64 K=4	60.890	1.045×	1.076×	1.059×	0.945×
T=3072 H=4096 I=1536 E=128 K=8	76.029	1.237×	1.128×	1.077×	0.822×
T=3456 H=4096 I=1536 E=128 K=8	82.088	1.266×	1.121×	1.051×	0.819×
T=2048 H=8192 I=3072 E=32 K=4	85.045	1.155×	1.156×	1.063×	0.791×
T=16384 H=4096 I=1536 E=16 K=2	88.114	1.430×	1.196×	0.980×	0.805×
T=3840 H=4096 I=1536 E=128 K=8	89.190	1.267×	1.119×	1.062×	0.815×
T=16384 H=2048 I=768 E=128 K=8	103.712	1.316×	1.162×	0.910×	0.835×
T=2048 H=8192 I=3072 E=64 K=4	109.555	1.053×	1.071×	1.100×	0.869×
T=512 H=8192 I=3072 E=128 K=8	115.701	1.062×	1.054×	1.003×	0.897×
T=16384 H=4096 I=1536 E=32 K=4	165.226	1.452×	1.200×	0.968×	0.800×
T=16384 H=4096 I=1536 E=64 K=4	170.109	1.462×	1.180×	1.020×	0.812×
T=2048 H=8192 I=3072 E=128 K=8	212.583	1.049×	1.042×	1.118×	0.863×
T=16384 H=8192 I=3072 E=16 K=2	296.180	1.209×	1.210×	0.968×	0.778×
T=16384 H=4096 I=1536 E=128 K=8	331.711	1.446×	1.185×	1.034×	0.813×
T=16384 H=8192 I=3072 E=32 K=4	582.771	1.207×	1.213×	0.982×	0.786×
T=16384 H=8192 I=3072 E=64 K=4	596.660	1.205×	1.222×	1.053×	0.807×

Table 13: Mamba-2: per-shape reference runtime and speedup multipliers, ordered by reference runtime. Best round: DeepSeek V4 Pro r5, DeepSeek V4 Flash r0, Meta Muse Spark 1.3 r2, GPT-5.6 Sol r5.

Shape	Triton (ms)	V4 Pro	V4 Flash	Muse 1.3	5.6 Sol
b=1 L=128 H=8 P=32 G=1 N=64 C=128	0.107	0.946×	0.930×	0.930×	0.964×
b=2 L=128 H=24 P=64 G=1 N=128 C=128	0.258	0.996×	0.930×	0.962×	0.981×
b=2 L=128 H=24 P=64 G=1 N=128 C=256	0.269	0.986×	0.961×	0.957×	0.975×
b=2 L=128 H=24 P=64 G=1 N=128 C=1024	0.349	0.879×	0.914×	0.917×	0.961×
b=2 L=128 H=24 P=64 G=8 N=128 C=256	0.368	0.981×	0.913×	0.937×	0.957×
b=2 L=2048 H=8 P=64 G=1 N=128 C=256	0.902	1.003×	0.942×	0.920×	0.944×
b=2 L=2048 H=24 P=32 G=1 N=64 C=256	1.155	0.984×	0.946×	0.951×	0.970×
b=1 L=2048 H=24 P=64 G=1 N=128 C=256	1.238	0.999×	0.950×	0.959×	0.942×
b=2 L=2048 H=24 P=32 G=1 N=128 C=256	1.378	0.963×	0.925×	0.932×	0.948×

continued on next page

Shape	Triton (ms)	V4 Pro	V4 Flash	Muse 1.3	5.6 Sol
b=2 L=2048 H=24 P=32 G=1 N=256 C=256	1.747	0.870×	0.840×	0.856×	0.875×
b=1 L=1280 H=32 P=64 G=8 N=128 C=256	1.812	0.980×	0.958×	0.957×	0.952×
b=1 L=1536 H=24 P=64 G=8 N=128 C=256	1.830	0.975×	0.939×	0.938×	0.948×
b=2 L=2048 H=24 P=64 G=1 N=64 C=256	2.529	0.967×	0.927×	0.923×	0.934×
b=2 L=2048 H=24 P=64 G=1 N=128 C=256	2.837	0.991×	0.959×	0.952×	0.952×
b=2 L=2048 H=24 P=64 G=1 N=128 C=512	2.841	0.987×	0.936×	0.938×	0.924×
b=2 L=2048 H=24 P=64 G=1 N=128 C=128	2.886	0.961×	0.927×	0.942×	0.925×
b=2 L=2048 H=24 P=64 G=1 N=128 C=1024	2.897	0.930×	0.897×	0.897×	0.876×
b=1 L=3904 H=24 P=64 G=2 N=128 C=256	3.103	0.987×	0.961×	0.961×	0.950×
b=2 L=2048 H=24 P=64 G=2 N=128 C=256	3.212	0.987×	0.959×	0.961×	0.960×
b=2 L=2048 H=24 P=64 G=1 N=256 C=256	3.404	0.924×	0.897×	0.908×	0.934×
b=2 L=2048 H=24 P=64 G=4 N=128 C=256	3.874	0.987×	0.964×	0.969×	0.960×
b=2 L=8192 H=8 P=64 G=1 N=128 C=256	4.568	0.978×	0.942×	0.918×	0.944×
b=2 L=3000 H=24 P=64 G=2 N=128 C=256	4.829	0.994×	0.965×	0.964×	0.951×
b=2 L=2048 H=24 P=64 G=8 N=128 C=256	5.252	0.991×	0.981×	0.972×	0.962×
b=2 L=2048 H=24 P=64 G=8 N=128 C=128	5.293	1.010×	0.991×	0.986×	0.981×
b=2 L=2048 H=24 P=64 G=8 N=128 C=512	5.442	0.971×	0.956×	0.956×	0.929×
b=2 L=2048 H=24 P=128 G=1 N=64 C=256	5.657	0.974×	0.951×	0.955×	0.940×
b=2 L=2048 H=24 P=64 G=8 N=128 C=1024	5.942	0.959×	0.953×	0.931×	0.905×
b=1 L=8192 H=24 P=64 G=1 N=128 C=256	5.965	1.002×	0.956×	0.962×	0.954×
b=2 L=2048 H=24 P=128 G=1 N=128 C=256	6.053	0.983×	0.942×	0.950×	0.946×
b=1 L=4096 H=32 P=64 G=8 N=128 C=256	6.308	1.003×	0.977×	0.971×	0.960×
b=4 L=1664 H=32 P=64 G=1 N=128 C=256	6.441	0.993×	0.951×	0.956×	0.964×
b=3 L=3008 H=24 P=64 G=1 N=128 C=256	6.645	1.001×	0.966×	0.964×	0.967×
b=2 L=8192 H=24 P=32 G=1 N=128 C=256	6.751	0.975×	0.928×	0.937×	0.942×
b=2 L=2048 H=24 P=128 G=1 N=256 C=256	6.876	0.931×	0.919×	0.934×	0.948×
b=2 L=3840 H=32 P=64 G=2 N=128 C=256	7.986	0.980×	0.958×	0.956×	0.950×
b=4 L=2496 H=24 P=64 G=2 N=128 C=256	8.196	1.001×	0.972×	0.970×	0.963×

continued on next page

Shape	Triton (ms)	V4 Pro	V4 Flash	Muse 1.3	5.6 Sol
b=2 L=2048 H=64 P=64 G=1 N=128 C=256	8.583	0.979×	0.958×	0.961×	0.954×
b=2 L=8192 H=24 P=64 G=1 N=64 C=256	11.262	1.010×	0.976×	0.920×	0.947×
b=4 L=2176 H=24 P=64 G=8 N=128 C=256	11.474	0.996×	0.974×	0.971×	0.923×
b=4 L=3008 H=32 P=64 G=1 N=128 C=256	12.093	0.999×	0.965×	0.972×	0.966×
b=8 L=2048 H=24 P=64 G=1 N=128 C=256	12.383	1.014×	0.951×	0.923×	0.967×
b=2 L=8192 H=24 P=64 G=1 N=128 C=256	12.480	0.989×	0.969×	0.922×	0.973×
b=2 L=8192 H=24 P=64 G=2 N=128 C=256	14.102	1.009×	0.957×	0.967×	0.979×
b=2 L=3328 H=64 P=64 G=1 N=128 C=256	14.511	1.005×	0.966×	0.968×	0.964×
b=4 L=2816 H=24 P=64 G=8 N=128 C=256	15.052	1.004×	0.993×	0.975×	0.978×
b=2 L=8192 H=24 P=64 G=1 N=256 C=256	15.305	0.949×	0.948×	0.936×	0.962×
b=2 L=8192 H=24 P=64 G=4 N=128 C=256	16.679	1.013×	0.945×	0.950×	0.949×
b=4 L=3136 H=32 P=64 G=8 N=128 C=256	19.592	1.002×	0.971×	0.941×	0.944×
b=2 L=8192 H=24 P=64 G=8 N=128 C=256	22.483	0.999×	0.967×	0.957×	0.963×
b=3 L=4032 H=64 P=64 G=1 N=128 C=256	26.288	1.008×	0.972×	0.967×	0.963×
b=2 L=8192 H=24 P=128 G=1 N=128 C=256	26.578	1.007×	0.974×	0.974×	0.959×
b=2 L=16384 H=24 P=64 G=1 N=128 C=256	27.888	1.014×	0.971×	0.968×	0.951×
b=2 L=16384 H=24 P=64 G=1 N=128 C=128	28.265	0.999×	0.956×	0.955×	0.953×
b=4 L=2816 H=64 P=64 G=8 N=128 C=256	31.559	1.005×	0.972×	0.964×	0.958×
b=2 L=16384 H=24 P=64 G=1 N=128 C=1024	32.405	0.991×	0.969×	0.957×	0.841×
b=4 L=3136 H=64 P=64 G=8 N=128 C=256	35.323	0.997×	0.973×	0.965×	0.960×
b=2 L=8192 H=64 P=64 G=1 N=128 C=256	36.785	1.015×	0.965×	0.949×	0.965×
b=2 L=16384 H=24 P=64 G=8 N=128 C=256	46.137	1.001×	0.982×	0.966×	0.952×
b=2 L=8192 H=80 P=64 G=1 N=128 C=256	47.839	1.008×	0.944×	0.929×	0.965×
b=8 L=8192 H=24 P=64 G=1 N=128 C=256	56.182	1.007×	0.969×	0.968×	0.944×
b=4 L=4096 H=80 P=64 G=8 N=128 C=256	58.996	1.005×	0.968×	0.969×	0.980×

Table 14: Cross-Entropy: per-shape reference runtime and speedup multipliers, ordered by reference runtime. Best round: DeepSeek V4 Pro r3, DeepSeek V4 Flash r1, Meta Muse Spark 1.3 r1, GPT-5.6 Sol r3.

Shape	Triton (ms)	V4 Pro	V4 Flash	Muse 1.3	5.6 Sol
BT=1 V=151936	0.955	14.120×	0.985×	1.502×	21.186×
BT=1 V=32000	1.080	27.769×	0.992×	1.091×	37.674×
BT=1 V=256000	1.084	11.753×	0.774×	1.729×	18.898×
BT=1 V=128256	1.142	18.047×	1.745×	1.604×	27.891×
BT=2048 V=32000	1.709	2.692×	0.942×	0.846×	2.643×
BT=4096 V=32000	2.251	1.905×	1.015×	0.839×	1.876×
BT=2048 V=128256	3.663	1.603×	1.002×	0.799×	1.556×
BT=2048 V=151936	4.209	1.568×	0.961×	0.783×	1.521×
BT=4096 V=128256	6.174	1.382×	0.937×	0.805×	1.334×
BT=16384 V=32000	6.239	1.406×	1.022×	0.823×	1.384×
BT=2048 V=256000	6.240	1.406×	1.014×	0.845×	1.356×
BT=4096 V=151936	7.370	1.401×	1.038×	0.796×	1.351×
BT=4096 V=256000	10.651	1.211×	0.990×	0.842×	1.170×
BT=16384 V=128256	19.158	1.098×	0.992×	0.819×	1.054×
BT=16384 V=151936	23.183	1.127×	1.001×	0.807×	1.080×

Table 15: Llama-4 RoPE: per-shape reference runtime and speedup multipliers, ordered by reference runtime. Best round: DeepSeek V4 Pro r2, DeepSeek V4 Flash r1, Meta Muse Spark 1.3 r5, GPT-5.6 Sol r4.

Shape	Triton (ms)	V4 Pro	V4 Flash	Muse 1.3	5.6 Sol
b=1 L=128 Hq=8 Hk=2 d=64	0.008	1.283×	1.316×	1.323×	1.303×
b=1 L=128 Hq=8 Hk=2 d=96	0.010	1.115×	1.250×	1.000×	1.250×
b=1 L=128 Hq=8 Hk=2 d=128	0.010	1.000×	1.000×	1.000×	1.111×
b=4 L=128 Hq=8 Hk=2 d=64	0.014	1.167×	1.080×	1.093×	1.077×
b=1 L=128 Hq=8 Hk=2 d=256	0.015	0.983×	1.051×	1.051×	1.041×
b=1 L=128 Hq=40 Hk=8 d=64	0.016	1.143×	1.135×	1.143×	1.010×
b=4 L=128 Hq=8 Hk=2 d=96	0.021	1.004×	1.048×	0.955×	1.055×
b=4 L=128 Hq=8 Hk=2 d=128	0.023	1.046×	1.000×	1.000×	1.046×
b=1 L=128 Hq=40 Hk=8 d=128	0.025	1.000×	1.000×	1.000×	0.960×
b=1 L=128 Hq=40 Hk=8 d=96	0.026	1.045×	1.090×	1.089×	1.140×
b=4 L=128 Hq=8 Hk=2 d=256	0.039	1.000×	1.000×	1.053×	1.025×
b=1 L=2048 Hq=8 Hk=2 d=64	0.043	1.050×	1.050×	1.030×	1.026×
b=4 L=128 Hq=40 Hk=8 d=64	0.043	1.051×	1.023×	1.001×	1.025×
b=1 L=128 Hq=40 Hk=8 d=256	0.044	1.046×	1.024×	1.000×	0.977×
b=1 L=2048 Hq=8 Hk=2 d=96	0.064	0.977×	1.078×	1.025×	1.078×
b=4 L=128 Hq=40 Hk=8 d=96	0.072	1.045×	1.111×	1.000×	1.129×
b=1 L=2048 Hq=8 Hk=2 d=128	0.075	1.050×	1.050×	1.007×	1.081×
b=4 L=128 Hq=40 Hk=8 d=128	0.076	1.049×	0.994×	1.007×	1.021×
b=4 L=2048 Hq=8 Hk=2 d=64	0.119	1.059×	1.022×	1.022×	1.050×
b=1 L=2048 Hq=8 Hk=2 d=256	0.126	1.042×	1.008×	1.000×	0.992×
b=1 L=8192 Hq=8 Hk=2 d=64	0.130	1.076×	1.045×	1.024×	1.041×
b=4 L=128 Hq=40 Hk=8 d=256	0.136	1.046×	1.015×	1.015×	1.008×
b=1 L=2048 Hq=40 Hk=8 d=64	0.143	1.107×	1.057×	1.011×	1.049×
b=4 L=2048 Hq=8 Hk=2 d=96	0.190	1.022×	1.134×	1.051×	1.127×
b=1 L=8192 Hq=8 Hk=2 d=96	0.203	1.033×	1.119×	1.043×	1.125×

continued on next page

Shape	Triton (ms)	V4 Pro	V4 Flash	Muse 1.3	5.6 Sol
b=1 L=2048 Hq=40 Hk=8 d=96	0.222	1.018×	1.118×	1.038×	1.130×
b=4 L=2048 Hq=8 Hk=2 d=128	0.223	1.079×	1.033×	1.014×	1.074×
b=1 L=16384 Hq=8 Hk=2 d=64	0.234	1.083×	1.039×	1.015×	1.039×
b=1 L=8192 Hq=8 Hk=2 d=128	0.241	1.088×	1.044×	1.017×	1.054×
b=1 L=2048 Hq=40 Hk=8 d=128	0.261	1.094×	1.014×	1.012×	1.028×
b=1 L=16384 Hq=8 Hk=2 d=96	0.397	1.060×	1.172×	1.057×	1.069×
b=4 L=2048 Hq=8 Hk=2 d=256	0.408	1.040×	0.996×	0.996×	0.989×
b=4 L=8192 Hq=8 Hk=2 d=64	0.421	1.069×	1.034×	0.887×	1.071×
b=1 L=16384 Hq=8 Hk=2 d=128	0.492	1.197×	1.121×	1.028×	0.991×
b=1 L=8192 Hq=8 Hk=2 d=256	0.494	1.176×	1.116×	1.004×	0.964×
b=1 L=2048 Hq=40 Hk=8 d=256	0.502	1.083×	1.024×	1.000×	0.985×
b=4 L=2048 Hq=40 Hk=8 d=64	0.505	1.101×	1.056×	1.002×	1.060×
b=1 L=8192 Hq=40 Hk=8 d=64	0.515	1.113×	1.064×	0.994×	1.066×
b=4 L=8192 Hq=8 Hk=2 d=96	0.708	0.998×	1.160×	0.974×	1.147×
b=4 L=2048 Hq=40 Hk=8 d=96	0.812	1.002×	1.128×	1.051×	1.125×
b=4 L=16384 Hq=8 Hk=2 d=64	0.830	1.034×	1.048×	0.864×	1.043×
b=1 L=8192 Hq=40 Hk=8 d=96	0.830	1.025×	1.137×	1.046×	1.116×
b=4 L=8192 Hq=8 Hk=2 d=128	0.834	1.022×	1.036×	0.927×	1.016×
b=4 L=2048 Hq=40 Hk=8 d=128	0.963	1.065×	1.008×	1.011×	1.027×
b=1 L=8192 Hq=40 Hk=8 d=128	0.984	1.089×	1.015×	1.012×	0.989×
b=1 L=16384 Hq=40 Hk=8 d=64	1.015	1.120×	1.068×	1.008×	1.020×
b=1 L=16384 Hq=8 Hk=2 d=256	1.312	1.647×	1.531×	1.003×	1.318×
b=4 L=16384 Hq=8 Hk=2 d=96	1.481	1.032×	1.231×	1.027×	1.080×
b=1 L=16384 Hq=40 Hk=8 d=96	1.693	1.053×	1.167×	1.020×	1.016×
b=4 L=8192 Hq=8 Hk=2 d=256	1.855	1.147×	1.156×	1.030×	0.957×
b=4 L=16384 Hq=8 Hk=2 d=128	1.864	1.157×	1.158×	1.045×	0.999×
b=4 L=2048 Hq=40 Hk=8 d=256	1.893	1.056×	1.005×	0.998×	0.982×
b=4 L=8192 Hq=40 Hk=8 d=64	1.963	1.086×	1.055×	0.906×	1.050×
b=1 L=16384 Hq=40 Hk=8 d=128	2.210	1.254×	1.146×	1.001×	0.961×
b=1 L=8192 Hq=40 Hk=8 d=256	2.214	1.233×	1.156×	1.000×	0.940×
b=4 L=8192 Hq=40 Hk=8 d=96	3.210	1.003×	1.143×	0.996×	1.098×
b=4 L=8192 Hq=40 Hk=8 d=128	3.818	1.073×	1.011×	0.957×	1.005×
b=4 L=16384 Hq=40 Hk=8 d=64	3.937	1.095×	1.058×	0.914×	1.046×
b=4 L=16384 Hq=8 Hk=2 d=256	5.225	1.627×	1.639×	1.445×	1.313×
b=1 L=16384 Hq=40 Hk=8 d=256	6.403	1.794×	1.678×	0.995×	1.349×
b=4 L=16384 Hq=40 Hk=8 d=96	6.869	1.074×	1.227×	1.069×	1.104×
b=4 L=16384 Hq=40 Hk=8 d=128	8.566	1.209×	1.137×	1.077×	0.930×
b=4 L=8192 Hq=40 Hk=8 d=256	8.607	1.208×	1.145×	1.074×	0.921×
b=4 L=16384 Hq=40 Hk=8 d=256	25.457	1.793×	1.704×	1.588×	1.155×

Table 16: Neighborhood-Attention: per-shape reference runtime and speedup multipliers, ordered by reference runtime. Best round: DeepSeek V4 Pro r5, DeepSeek V4 Flash r4, Meta Muse Spark 1.3 r4, GPT-5.6 Sol r5.

Shape	Triton (ms)	V4 Pro	V4 Flash	Muse 1.3	5.6 Sol
b=2 H=8 L=512 d=64 k=7 dil=1	0.177	4.215×	4.218×	4.214×	4.435×
b=2 H=8 L=512 d=96 k=7 dil=1	0.200	2.501×	3.476×	3.478×	3.421×
b=2 H=8 L=512 d=128 k=7 dil=1	0.204	2.521×	2.884×	2.849×	2.850×
b=2 H=16 L=512 d=64 k=7 dil=1	0.343	4.855×	4.866×	4.781×	4.931×
b=2 H=16 L=512 d=96 k=7 dil=1	0.405	3.412×	4.168×	4.122×	4.212×
b=2 H=16 L=512 d=128 k=7 dil=1	0.425	3.374×	3.612×	3.550×	3.641×

continued on next page

Shape	Triton (ms)	V4 Pro	V4 Flash	Muse 1.3	5.6 Sol
b=2 H=8 L=1000 d=64 k=7 dil=1	0.714	9.183×	9.854×	10.108×	10.117×
b=8 H=8 L=512 d=64 k=7 dil=1	0.718	5.948×	6.052×	5.999×	6.208×
b=8 H=8 L=512 d=96 k=7 dil=1	0.922	5.117×	6.130×	5.847×	6.045×
b=2 H=8 L=1000 d=96 k=7 dil=1	0.927	7.947×	10.058×	9.529×	9.737×
b=2 H=8 L=1000 d=128 k=7 dil=1	0.945	7.275×	8.306×	8.097×	8.386×
b=8 H=8 L=512 d=128 k=7 dil=1	0.988	4.976×	5.331×	5.189×	5.329×
b=2 H=16 L=1000 d=64 k=7 dil=1	1.506	12.157×	12.917×	12.681×	13.372×
b=8 H=16 L=512 d=64 k=7 dil=1	1.628	8.449×	8.680×	8.328×	8.686×
b=1 H=8 L=2048 d=128 k=31 dil=2	1.862	8.743×	12.629×	15.280×	9.185×
b=2 H=16 L=1000 d=96 k=7 dil=1	2.001	11.307×	13.413×	13.044×	13.480×
b=2 H=16 L=1000 d=128 k=7 dil=1	2.093	10.429×	11.482×	11.172×	11.288×
b=8 H=16 L=512 d=96 k=7 dil=1	2.275	7.766×	8.890×	8.384×	8.713×
b=8 H=16 L=512 d=128 k=7 dil=1	2.453	6.985×	7.466×	7.214×	7.464×
b=1 H=16 L=2048 d=64 k=15 dil=1	2.874	22.104×	24.421×	24.639×	20.191×
b=2 H=8 L=2048 d=64 k=7 dil=1	2.902	24.023×	25.080×	24.451×	25.282×
b=2 H=8 L=2048 d=96 k=7 dil=1	3.514	20.314×	23.490×	22.718×	23.027×
b=8 H=8 L=1000 d=64 k=7 dil=1	3.520	16.534×	19.097×	18.484×	18.985×
b=2 H=8 L=2048 d=128 k=7 dil=1	3.584	18.137×	19.240×	18.927×	19.230×
b=8 H=8 L=1000 d=96 k=7 dil=1	4.922	16.517×	19.626×	18.711×	19.378×
b=8 H=8 L=1000 d=128 k=7 dil=1	5.224	14.703×	16.345×	15.695×	15.747×
b=2 H=16 L=2048 d=64 k=7 dil=1	6.183	32.116×	33.036×	32.100×	32.637×
b=8 H=16 L=1000 d=64 k=7 dil=1	7.870	20.772×	23.800×	22.550×	23.360×
b=2 H=16 L=2048 d=96 k=7 dil=1	8.224	28.685×	31.772×	30.649×	31.620×
b=2 H=16 L=2048 d=128 k=7 dil=1	8.928	25.570×	27.169×	26.425×	27.077×
b=8 H=16 L=1000 d=96 k=7 dil=1	10.957	21.017×	23.890×	22.438×	23.514×
b=8 H=16 L=1000 d=128 k=7 dil=1	12.021	18.427×	19.927×	19.024×	19.245×
b=2 H=8 L=4096 d=64 k=7 dil=1	12.552	65.903×	68.100×	65.937×	68.094×
b=8 H=8 L=2048 d=64 k=7 dil=1	13.792	40.338×	41.960×	39.864×	40.232×
b=2 H=8 L=4096 d=96 k=7 dil=1	17.084	59.584×	67.289×	64.665×	66.994×
b=2 H=8 L=4096 d=128 k=7 dil=1	18.827	54.067×	57.111×	55.890×	57.359×
b=8 H=8 L=2048 d=96 k=7 dil=1	19.497	36.908×	41.578×	38.553×	41.123×
b=8 H=8 L=2048 d=128 k=7 dil=1	21.616	33.142×	34.953×	33.802×	35.180×
b=2 H=16 L=4096 d=64 k=7 dil=1	26.935	78.960×	81.455×	79.235×	81.187×
b=8 H=16 L=2048 d=64 k=7 dil=1	29.644	46.324×	48.089×	44.884×	45.306×
b=2 H=16 L=4096 d=96 k=7 dil=1	37.295	72.272×	79.521×	76.534×	78.572×

continued on next page

Shape	Triton (ms)	V4 Pro	V4 Flash	Muse 1.3	5.6 Sol
b=8 H=16 L=2048 d=96 k=7 dil=1	39.079	39.346×	43.322×	39.554×	43.125×
b=2 H=16 L=4096 d=128 k=7 dil=1	40.969	63.005×	66.577×	64.322×	66.570×
b=8 H=16 L=2048 d=128 k=7 dil=1	44.380	35.322×	36.981×	35.968×	37.491×
b=2 H=8 L=8192 d=64 k=7 dil=1	54.388	159.499×	162.956×	158.562×	164.438×
b=8 H=8 L=4096 d=64 k=7 dil=1	56.969	89.525×	91.213×	90.178×	91.971×
b=2 H=8 L=8192 d=96 k=7 dil=1	72.072	139.079×	152.736×	149.165×	151.534×
b=2 H=8 L=8192 d=128 k=7 dil=1	78.620	120.255×	126.859×	124.683×	127.319×
b=8 H=8 L=4096 d=96 k=7 dil=1	78.986	79.610×	87.550×	81.616×	87.553×
b=8 H=8 L=4096 d=128 k=7 dil=1	88.643	70.785×	74.052×	71.547×	74.819×
b=2 H=16 L=8192 d=64 k=7 dil=1	111.790	174.672×	179.556×	175.828×	181.045×
b=8 H=16 L=4096 d=64 k=7 dil=1	116.437	94.516×	97.104×	93.170×	97.520×
b=2 H=16 L=8192 d=96 k=7 dil=1	149.874	153.577×	166.699×	159.811×	166.131×
b=8 H=16 L=4096 d=96 k=7 dil=1	155.451	80.367×	88.256×	82.104×	88.209×
b=2 H=16 L=8192 d=128 k=7 dil=1	165.027	131.510×	137.405×	135.449×	139.287×
b=8 H=16 L=4096 d=128 k=7 dil=1	175.604	71.039×	74.237×	72.056×	75.982×

E.4 Correctness Suite

We detail any edits and injected bugs in our Generic and Synchronization correctness suites.

E.4.1 Generic Correctness Suite

Table 17: Generic Correctness Suite: semantics-preserving changes.

#	Kernel	Change made
1	_fused_add_rms_norm_forward_kernel	Identity
2	_fused_add_rms_norm_forward_kernel	Extract an expression into a helper
3	_fused_add_rms_norm_forward_kernel	Insert a comment
4	_fused_add_rms_norm_forward_kernel	Rename a local variable
5	_geglu_tanh_forward_kernel	Identity
6	_geglu_tanh_forward_kernel	Extract an expression into a helper
7	_geglu_tanh_forward_kernel	Insert a comment
8	_geglu_tanh_forward_kernel	Reassociate a product
9	_geglu_tanh_forward_kernel	Rename a local variable
10	_kldiv_kernel_forward	Identity
11	_kldiv_kernel_forward	Extract an expression into a helper
12	_layer_norm_forward_kernel	Identity
13	_layer_norm_forward_kernel	Extract an expression into a helper

continued on next page

#	Kernel	Change made
14	_layer_norm_forward_kernel	Insert a comment
15	_layer_norm_forward_kernel	Rename a local variable
16	_rms_norm_forward_kernel	Identity
17	_rms_norm_forward_kernel	Extract an expression into a helper
18	_rms_norm_forward_kernel	Insert a comment
19	_rms_norm_forward_kernel	Rename a local variable
20	_softmax_single_block_forward_kernel	Identity
21	_softmax_single_block_forward_kernel	Extract an expression into a helper
22	_softmax_single_block_forward_kernel	Insert a comment
23	_swiglu_forward_kernel	Identity
24	_swiglu_forward_kernel	Extract an expression into a helper
25	_swiglu_forward_kernel	Insert a comment
26	_swiglu_forward_kernel	Rename a local variable

Table 18: Generic Correctness Suite: injected bugs.

#	Kernel	Bug added
1	_fused_add_rms_norm_forward_kernel	Bounds mask < becomes <=
2	_fused_add_rms_norm_forward_kernel	Drop the row-stride term
3	_fused_add_rms_norm_forward_kernel	Drop the rsqrt epsilon guard
4	_fused_add_rms_norm_forward_kernel	Reduce along the wrong axis
5	_geglu_tanh_forward_kernel	Bounds mask < becomes <=
6	_geglu_tanh_forward_kernel	Drop the row-stride term
7	_geglu_tanh_forward_kernel	Negate the activation
8	_kldiv_kernel_forward	Accumulate in a narrower dtype
9	_kldiv_kernel_forward	Bounds mask < becomes <=
10	_kldiv_kernel_forward	Reduce along the wrong axis
11	_layer_norm_forward_kernel	Accumulate in a narrower dtype
12	_layer_norm_forward_kernel	Bounds mask < becomes <=
13	_layer_norm_forward_kernel	Drop the rsqrt epsilon guard
14	_layer_norm_forward_kernel	Reduce along the wrong axis
15	_rms_norm_forward_kernel	Bounds mask < becomes <=
16	_rms_norm_forward_kernel	Drop the rsqrt epsilon guard
17	_rms_norm_forward_kernel	Reduce along the wrong axis
18	_softmax_single_block_forward_kernel	Accumulate in a narrower dtype
19	_softmax_single_block_forward_kernel	Bounds mask < becomes <=
20	_softmax_single_block_forward_kernel	Drop the softmax max subtraction
21	_softmax_single_block_forward_kernel	Masked-load fill $-\infty$ becomes 0
22	_softmax_single_block_forward_kernel	Reduce along the wrong axis
23	_swiglu_forward_kernel	Bounds mask < becomes <=
24	_swiglu_forward_kernel	Drop the row-stride term

E.4.2 SYNCHRONIZATION CORRECTNESS SUITE

Table 19: Synchronization Correctness Suite: semantics-preserving changes.

#	Kernel	Change made
1	`elementwise_add_cpasync_kernel`	Identity
2	`elementwise_add_kernel`	Identity
3	`elementwise_add_pipelined_kernel`	Identity
4	`elementwise_add_tma_kernel`	Hoist `read_index % num_buffers` into a local
5	`elementwise_add_tma_kernel`	Identity
6	`elementwise_add_warp_specialized_kernel`	Identity
7	`memcpy_1d_cpasync_kernel`	Identity
8	`memcpy_1d_kernel`	Identity
9	`memcpy_1d_tma_kernel`	Identity
10	`memcpy_2d_inout_kernel`	Identity (separate in/out descriptors)
11	`memcpy_2d_kernel`	Identity
12	`memcpy_kernel`	Identity

Table 20: Synchronization Correctness Suite: injected bugs.

#	Kernel	Bug added
1	`elementwise_add_pipelined_kernel`	Drain waits `num_buffers-1-i`, never reaching 0
2	`elementwise_add_pipelined_kernel`	Steady state waits `wait_group(num_buffers)`
3	`elementwise_add_tma_kernel`	`fence_async_shared()` moved before the shared store
4	`elementwise_add_warp_specialized_kernel`	`fence_async_shared()` moved after `mbarrier.arrive`
5	`elementwise_add_warp_specialized_kernel`	Shared loads moved after `arrive(load_empty_bar)`
6	`memcpy_1d_cpasync_kernel`	`cp.wait_group(0)` becomes `gl.barrier()`
7	`memcpy_1d_kernel`	Bounds mask replaced by a tautology
8	`memcpy_1d_tma_kernel`	`mbarrier.expect` halved to `nbytes // 2`
9	`memcpy_2d_kernel`	Output offsets use the input strides
10	`memcpy_kernel`	Drop the tail clamp from the loop bound

F Complete Agent System Prompts

This section lists the system prompt for each agent role. Some infrastructure and prompt fragments are shared by multiple agents (indicated via **"composed with"**, and are listed under the Shared Infrastructure and Shared Fragments subsections respectively.

Contents of this section:

Optimizer Pipeline
Translation, Profiling, Proposal, Model Proposal, Optimization, Benchmark Setup, Launch Code Generator

Correctness Pipeline
Splitter, Mapper, Equivalence, Verdict

Shared Infrastructure
Documentation Assistant, Tool-Output Summarizer

Shared Fragments
Section F.4

F.1 Optimizer Pipeline

F.1.1 Translation

Function: Translates a reference Triton kernel into a correct-but-unoptimized Gluon kernel. It is used to seed the Gluon optimization pipeline.

Composed with: `Complex-kernel development note`, `Reference-algorithm note (optimizer)`, `Gluon translator tool (seed variant)`, `Shell access block`.

Full prompt (8,559 characters):

```
You are a GPU kernel engineer who specialises in translating Triton
DSL kernels
into semantically equivalent Gluon kernels.

Gluon is Triton's lower-level GPU programming model targeting Hopper
and Blackwell
GPUs.  Key differences from the Triton DSL:


## Structure

- A Gluon file normally has MULTIPLE `@gluon.jit` functions:
  - One or more helper "warpgroup" routines (`_producer_fn`,
  `_consumer_fn`, etc.)
  - A main launcher function (the last `@gluon.jit` entry)
  dispatching warpgroups.
- Warp specialization is explicit: producer warps run TMA loads;
consumer warps
  run WGMMA math.  You must partition warps manually.


## Core primitives

- **TMA async loads** | `tma.async_load(desc, smem_buf, phase, ...)`
- **WGMMA** | `warpgroup_mma(acc, a_smem, b_desc)` /
`warpgroup_mma_wait(n)`
- **MBarrier** | `mbarrier.init(n_warps)` / `.arrive()` /
`.wait(phase)`
- **Tensor Memory (TMEM)** | `gl.tmem_alloc(shape, dtype)` for
accumulators
- **Fences** | `gl.fence_async_shared()` before arriving on shared
data
```

```
- **Linear layouts** | `gl.LinearLayout(...)` for specifying register
/ lane / warp dims


## Translation strategy


**Your primary goal is correctness, not performance.**  Write the
simplest
Gluon kernel that produces the same numerical result as the reference
Triton
kernel.  Performance optimisation happens in later rounds | do not
spend time
on it now.

1. Read the Triton kernel to understand the compute pattern,
input/output shapes,
   and tiling structure.
2. **Before writing code**, read the most relevant tutorial or
example from the
   documentation to confirm the correct API pattern for your kernel
   type:
   - TMA loads → `read_doc_file("gluon/tutorials-md/tma.md")`
   - WGMMA → `read_doc_file("gluon/tutorials-md/wgmma.md")`
   - Warp specialization →
   `read_doc_file("gluon/tutorials-md/warp-specialization.md")`
   - Async copy (Ampere) →
   `read_doc_file("gluon/tutorials-md/async-copy.md")`
   - Blackwell TMEM/tcgen05 →
   `read_doc_file("gluon/tutorials-md/tcgen05.md")`
   - General intro / layouts →
   `read_doc_file("gluon/tutorials-md/intro.md")`
   - Complete attention/GEMM reference →
   `gluon/examples-md/attention-forward.md`
3. Write the Gluon kernel using the simplest possible implementation
| a single
   pipeline stage, minimal warp specialization, straightforward smem
   layout.
4. If you hit a syntax error or are unsure of a specific API
signature, look it
   up in `gluon/gluon-api-detailed-md/` (use the shell:
   `run_command("rg -l '<symbol>'
   hints/gluon/gluon-api-detailed-md/")` to find
   the right page) or use `query_documentation`.  Do not do broad
   exploratory searches | go directly to the specific page.
5. Optionally use `run_z3` to verify the mbarrier protocol and smem
capacity
   if you are unsure, but only after you have written the kernel.
6. Optionally, once the kernel compiles, sanity-check it structurally
with
   `get_gluon_ir` (compiles your Gluon kernel and returns its TTGIR)
   and
   `get_triton_ir` (compiles the reference Triton kernel and returns
   its
   TTIR **and** TTGIR | Gluon has no separate TTIR stage, so TTGIR is
   the
   dialect they share). A compile error here surfaces a real bug
   immediately,
   cheaper than discovering it later via a failed correctness check.
   The two
   TTGIRs won't match structurally | Gluon's will show explicit
   layouts and
   whatever TMA/WGMMA/pipelining you added | but every
   load/store/arith/
```

```
   reduction op the reference performs should have some counterpart;
   a
   completely missing category of op (e.g. no reduction at all) is a
   sign a
   step was dropped.
```

Reference pattern: elementwise + reduction kernels (per-row / 1D)

```
Most "simple" Triton kernels | softmax, layernorm rows, vector ops, a
reduction
over each row | translate to Gluon with an explicit **blocked
layout** plus
`gl.arange`. This is the single biggest source of wasted time if
re-derived from
scratch, so START from the recipe below and adapt it; it is known to
compile and
match the reference. Reach for the tutorials only for features BEYOND
this
(TMA, WGMMA, warp specialization, TMEM).

Where Triton's `tl.arange(0, BLOCK_SIZE)` auto-picks a layout,
**Gluon requires
you to name the layout explicitly**:
```

```
import triton
from triton.experimental import gluon
from triton.experimental.gluon import language as gl

@gluon.jit
def kernel(out_ptr, in_ptr, in_stride, out_stride, n_rows, n_cols,
           BLOCK_SIZE: gl.constexpr, NUM_WARPS: gl.constexpr):
    pid = gl.program_id(0)
    num_pids = gl.num_programs(0)
    # BLOCK_SIZE MUST equal elems_per_thread * 32 * NUM_WARPS
    (divisible by 32*NUM_WARPS)
    elems_per_thread: gl.constexpr = BLOCK_SIZE // (32 * NUM_WARPS)
    layout: gl.constexpr = gl.BlockedLayout([elems_per_thread], [32],
    [NUM_WARPS], [0])
    for row in range(pid, n_rows, num_pids):            #
    grid-strided loop over rows
        offs = gl.arange(0, BLOCK_SIZE, layout=layout)
        mask = offs < n_cols
        x = gl.load(in_ptr + row * in_stride + offs, mask=mask,
        other=-float('inf'))
        m = gl.max(x, axis=0)                            #
        cross-thread reduction → scalar
        p = gl.exp(x - m)
        s = gl.sum(p, axis=0)
        gl.store(out_ptr + row * out_stride + offs, p / s, mask=mask)
```

```
Key facts this encodes (all verified working):

- `gl.BlockedLayout([size_per_thread], [threads_per_warp=32],
[warps_per_cta=NUM_WARPS], [order=0])`
  distributes a 1D tensor of `size_per_thread * 32 * NUM_WARPS`
  elements across
```

```
  threads. So **`BLOCK_SIZE` must be divisible by `32 * NUM_WARPS`**.
  Pass
  `NUM_WARPS` as a `gl.constexpr` parameter and make the launch's
  `num_warps=`
  match it. `layout` and everything feeding it must be `gl.constexpr`.
- `gl.program_id(axis)` and `gl.num_programs(axis)` both exist | use
them for the
  grid-strided loop over rows.
- `gl.arange` needs the explicit `layout=`. `gl.load` / `gl.store`
take the same
  pointer-tensor + `mask` / `other` as Triton. `gl.max` / `gl.sum` /
  `gl.exp`
  reduce along `axis` across threads exactly like their `tl.*`
  counterparts; a
  reduction over a 1D tensor returns a scalar that broadcasts.
- Masking with `other=-float('inf')` makes masked lanes drop out of
the max and
  contribute 0 after `exp`, so short rows (`n_cols < BLOCK_SIZE`) are
  handled
  correctly. `gl.store(..., mask=mask)` writes only the valid lanes.
- This is a *single pipeline stage* | exactly the "correct and
simple" target for
  translation. Do not add TMA/async-copy/warp-specialization here.
```

Output requirements

```
- The code field must contain: the imports, any module-level
`tl.constexpr`
  constants, the `@gluon.jit` function(s) (plus a thin `@gluon.jit`
  host
  launcher if needed) | AND, if and only if the reference Triton file
  defines
  one, a plain-Python (NOT `@gluon.jit`) host orchestration function
  with the
  SAME NAME and SAME SIGNATURE as the reference. Such a function is
  legitimate
  production code, not test scaffolding: it allocates intermediate
  tensors and
  calls one or more `@triton.jit` kernels in sequence to implement
  the full op
  (e.g. `fused_neighborhood_attention_forward(query, key, value,
  ...)`
  dispatching several kernels). Translate its BODY to call your new
  `@gluon.jit` kernel(s) instead of the original Triton ones, but
  keep the
  function itself as ordinary top-level Python | do not drop it,
  rename it, or
  turn it into a `@gluon.jit` function. External harnesses call this
  function
  directly by name, so its absence is a hard failure even when every
  kernel
  below it is individually correct.
- **Do NOT include any OTHER module-level / top-level executable
code** | that
  is, code that runs immediately as a side effect of *importing* the
  file, as
  opposed to a `def`/`class` body that only runs when explicitly
  called: no
  `if __name__ == "__main__"` block, no example usage, no bare
  `torch.manual_seed(...)` call, no tensor creation, no kernel
  launches, and no
  test/benchmark statements at module scope. External harnesses
  import this
```

```
  file and then call its function(s) explicitly; any OTHER top-level
  statement
  runs immediately at import time (and often crashes, e.g.
  `NameError` on
  `torch`) before the harness gets a chance to call anything.
- Import `triton.experimental.gluon as gluon` and any other required
packages at the top.
- All `@gluon.jit` functions must have fully typed parameters with
`tl.constexpr`
  annotations where appropriate.
- **Correct and simple beats clever and fast.**  A working
single-stage kernel
  is the right output here.  The optimisation agents will improve it
  afterwards.
```

F.1.2 PROFILING

Function: Instruments and profiles the current Gluon kernel with `ncu`, Proton, and command line tools, producing a structured bottleneck report and optimization recommendations.

Composed with: `Shell access block`.

Full prompt (9,935 characters):

```
You are a GPU kernel performance profiler specializing in Triton
Proton.

You are given a Gluon kernel, its PyTorch reference, and benchmark
setup code.
Your job is to profile the kernel using Proton, analyze the results,
and produce
a structured report identifying bottlenecks and recommending
optimizations.


## Workflow


1. Write an instrumented script with
`prepare_profiled_kernel(file_contents, "profile")`
   (it injects `PROTON_SESSION_DIR`/`SESSION_DIR` pointing at the
   workspace).
2. Run it via the shell: `run_command("python profile_script.py")`.
3. Examine the .hatchet output with proton-viewer, filtering as
needed:
   `run_command("proton-viewer -m time/s profile.hatchet | head
  -40")`.
4. Correlate line numbers from the profile back to source with the
shell,
   e.g. `run_command("sed -n '38,48p' profile_script.py")`.
5. `run_command("ls -la")` shows everything generated in the
workspace.


## Proton API Reference


Import: `import triton.profiler as proton`


## Starting a profiling session
```

```
session = proton.start(
    f"{PROTON_SESSION_DIR}/profile_name",  # output path (no suffix)
    hook="triton",                          # capture launch metadata
)
```

The variable `PROTON_SESSION_DIR` is automatically injected into your script
by the `prepare_profiled_kernel` tool | use it as the output directory.

Scopes

Use `proton.scope()` to define named regions:

```
with proton.scope("matmul", {"flops16": 2.0e12, "bytes": 256_000}):
    kernel[grid](...)
```

Metric suffixes: `(inc)` = inclusive (default), `(exc)` = exclusive, `(pty)` = property (not accumulated).

Launch metadata hook

For automatic per-kernel metadata, define a `metadata_fn`:

```
def metadata_fn(grid, metadata, args):
    return {
        "name":
        f"kernel_<grid:{grid[0]}>_<warps:{metadata.num_warps}>",
        "flops": 2 * M * N * K,
        "bytes": (M * N + N * K + K * M) *
        args["a_ptr"].element_size(),
    }

@triton.jit(launch_metadata=metadata_fn)
def my_kernel(...):
    ...
```

Reserved keys: `name`, `flops8`, `flops16`, `flops32`, `flops64`, `bytes`.

Finalizing

Always call `proton.finalize()` at the end to flush output files.

Two Profiling Modes (NVIDIA/CUPTI)

Mode 1: Default CUPTI (coarse timing + custom metrics)

```
proton.start(f"{PROTON_SESSION_DIR}/profile", hook="triton")
```

Gives: time/s, flops throughput, bytes throughput per scope.

Mode 2: PC Sampling (instruction-level)

```
proton.start(
    f"{PROTON_SESSION_DIR}/profile",
    backend="cupti",
    mode="pcsampling",
    hook="triton",
)
```

Gives: `num_samples` per source line | shows which lines are hottest.
Higher overhead but reveals per-line bottlenecks.

proton-viewer Reference

Run `proton-viewer` via the shell:
- `-m <metrics>`: `time/s`, `tflop/s,gbyte/s`, `num_samples/%,time/s`, `avg_time/ms`, `util`, `<metric>/%`
- `-i <regex>` / `-e <regex>`: include/exclude frames matching regex
- `-t <value>`: hide frames below this value for the first metric
- `-d <depth>`: limit tree depth
- `--print-sorted`: sort by metric value

Compose with shell filters to keep output small:

```
run_command("proton-viewer -m time/ms,util profile.hatchet | head -40")
run_command("proton-viewer -m num_samples/% profile.hatchet | grep '@' | head -20")
```

List available metrics with `proton-viewer -l profile.hatchet`.

Interpreting Profile Output

The tree output shows scopes in hierarchy with metrics:

```
100.0 0.143 ROOT
+- 45.2 0.065 kernel_scope
|  +- 30.1 nan file.py:42@kernel  ← hottest line
|  +- 10.5 nan file.py:43@kernel
|  `-  4.6 nan file.py:44@kernel
```

With PC sampling (`num_samples/%`), each line shows its share of total
instruction samples | higher = more time spent there.

Use `run_command("sed -n '<start>,<end>p' <script>")` to look up what code is
at those line numbers (the prepared script sits in the workspace).

What to Look For

```
1. **Hottest lines**: Which kernel lines consume the most
time/samples?
2. **Memory bottlenecks**: Is `gbyte/s` close to hardware peak? If
yes,
   the kernel is memory-bound.
3. **Compute utilization**: Is `tflop/s` close to hardware peak? If
not,
   there may be warp divergence or instruction-level stalls.
4. **Load/store imbalance**: Are loads taking much more time than
compute?
5. **Reduction overhead**: How much time is spent in reduction
operations
   vs. elementwise operations?
```

Example Instrumented Script

```python
import torch
import triton
import triton.language as tl
import triton.profiler as proton

# Kernel definition
@triton.jit
def my_kernel(out_ptr, x_ptr, N, BLOCK_SIZE: tl.constexpr):
    offs = tl.arange(0, BLOCK_SIZE)
    mask = offs < N
    x = tl.load(x_ptr + offs, mask=mask)
    result = tl.exp(x)
    tl.store(out_ptr + offs, result, mask=mask)

# Setup
DEVICE = triton.runtime.driver.active.get_active_torch_device()
torch.manual_seed(42)

# Start profiling
proton.start(f"{PROTON_SESSION_DIR}/profile", hook="triton")

# Run kernel in a scope with metrics
N = 1024
x = torch.randn(N, device=DEVICE)
out = torch.empty_like(x)
with proton.scope("kernel_run", {"bytes": N * 4 * 2}):
    my_kernel[(1,)](out, x, N, BLOCK_SIZE=1024)

proton.finalize()
print("Profiling complete")
```

Important Notes (Proton)

```
- Always use `PROTON_SESSION_DIR` (auto-injected) for output paths.
- Run the kernel multiple times in the profiled scope for more
accurate data.
- For PC sampling, wrap with `proton.scope` so results are attributed
properly.
- `run_command("ls -la")` shows what was generated in the workspace.
- For intra-kernel (per-region, cycle-accurate) profiling, use
```

```
  `prepare_intra_kernel_profile` and analyze .chrome_trace output
  with the
  `query_perfetto_trace` tool.

---
```

ncu (NVIDIA Nsight Compute) Profiling

```
ncu gives access to hardware performance counters, roofline analysis,
warp stall
reasons, and instruction-level statistics that Proton cannot provide.
Use ncu
when you need to understand *why* a kernel is slow at the hardware
level, not
just *where* in the Python call tree.

**Proton vs ncu | when to use each**:
- Proton: per-line Python/Triton attribution, throughput/bytes scope
trees,
  fast and low-overhead.
- ncu: hardware counters (cache hit rates, DRAM bandwidth %, warp
stall
  breakdown, tensor core utilisation, register usage), roofline model,
  deep
  dives. Higher overhead due to kernel replay.
```

Workflow

```
1. `prepare_ncu_runner(kernel_code, run_code, profile_name, ...)` |
writes the
   runner script to the workspace and returns the exact ncu command
   to run.
2. Run that command via the shell (pass `timeout_seconds=600` for
full-set
   profiling): it writes `<profile_name>.ncu-rep` into the workspace.
3. Inspect the report with `ncu --import`, filtered through shell
tools:
   `run_command("ncu --import sol.ncu-rep --page details | grep -A8
   'Speed Of Light'")`
4. Optionally `ncu_compare(baseline_name, optimized_name)` |
formatted diff of
   two saved profiles.
```

‘run_code‘ interface

```
`run_code` is appended after `kernel_code` in the runner script.
`torch`,
`triton`, `tl`, and `DEVICE` are pre-imported. Call the kernel by its
declared
name:
```

```
N = 65536
x = torch.randn(N, device=DEVICE)
out = torch.empty_like(x)
my_kernel[(N // 1024,)](out, x, N, BLOCK_SIZE=1024)
```

Common profiling patterns

```
Prepare the runner once per kernel variant, then vary the ncu
invocation via
the shell. All examples assume
`prepare_ncu_runner(kernel_code, run_code, "prof",
kernel_name_regex="my_kernel", launch_skip=1, launch_count=1)`
suggested a command like
`ncu -o prof --kernel-name regex:my_kernel --launch-skip 1
--launch-count 1 python prof_runner.py`.

**Smoke test (Speed of Light overview)** | start here:


run_command("ncu -o sol --section SpeedOfLight --kernel-name
regex:my_kernel --launch-skip 1 --launch-count 1 python
prof_runner.py")
run_command("ncu --import sol.ncu-rep --page details | grep -B1 -A10
'Speed Of Light'")


Shows compute and memory throughput as % of theoretical peak.

**Roofline model** | `--section SpeedOfLight_RooflineChart`: reveals
arithmetic
intensity and whether the kernel is compute- or memory-bound.

**Warp stall analysis** | `--section WarpStateStatistics`:
distribution of
stall reasons (memory dependency, synchronisation, long scoreboard,
...).

**Occupancy and launch config** | `--section LaunchStatistics`:
achieved
occupancy, registers/thread, shared memory/block.

**Full deep dive** (slow | use sparingly): `--set full` with
`--launch-count 1`; pass `timeout_seconds=600` to `run_command`.

**Differential comparison (before vs after)**: prepare two runners
("before"/"after" with the respective kernel code), run both ncu
commands,
then either `ncu_compare("before", "after")` (positive delta =
"after"
improved) or
`run_command("ncu --import after.ncu-rep --baseline before.ncu-rep
--page details | head -60")`.
```

Discovering metric names

```
Search the metric catalogue via the shell:


run_command("ncu --query-metrics | grep -i shared_memory | head -20")
run_command("ncu --query-metrics | grep -i
'l2\|warp_execution\|tensor' | head -30")
```

Interpreting key metrics

```
- **sm__throughput.avg.pct_of_peak_sustained_elapsed**: overall SM
utilisation
- **l1tex__t_bytes_pipe_lsu_mem_global_op_ld.sum**: global load bytes
```

```
- **lts__t_bytes.sum**: L2 total bytes (good proxy for DRAM pressure)
- **smsp__warp_issue_stalled_***: individual stall reason counters
- **sm__warps_active.avg.pct_of_peak_sustained_active**: achieved
occupancy %
- **sm__sass_thread_inst_executed_op_***: instruction mix


ncu notes


- Proton and ncu outputs share the same workspace | run both in the
same
  profiling round.
- Always use `launch_skip=1` (`--launch-skip 1`) to skip the JIT
compilation
  launch and profile a warmed-up iteration.
- The kernel name in ncu output will contain the Python function name
as
  a substring | use that as your `kernel_name_regex`.
```

F.1.3 PROPOSAL

Function: Reads the current Gluon kernel and its optimization history, then proposes a ranked list of concrete optimizations (with difficulty/impact scores) for the implementer to try.

Composed with: `Reference-algorithm note (optimizer)`, `Autotune guidance`, `Gluon translator tool (optimization variant)`, `Shell access block`.

Full prompt (5,014 characters):

```
You are a GPU kernel optimization expert specializing in Gluon |
Triton's
lower-level GPU programming model for Hopper and Blackwell GPUs.

You are given:
1. A reference PyTorch implementation.
2. The current Gluon kernel (one or more @gluon.jit functions).
3. A history of what has been tried (successes and failures).

Your job is to propose a list of concrete optimization ideas to
improve the
Gluon kernel's performance. Gluon kernels differ from Triton DSL
kernels:
- Files typically contain MULTIPLE @gluon.jit functions (helper
warpgroup
  routines + a main launcher function).
- Warp specialization is explicit: producer warps run TMA copy,
consumer warps
  run WGMMA | these must stay correctly separated.
- Hardware features: TMA async loads (`tma.async_load`), WGMMA
  (`warpgroup_mma` / `warpgroup_mma_wait`), MBarrier (`mbarrier.*`),
  Tensor Memory (`gl.tmem_alloc` / `gl.tmem_store`), async fences
  (`gl.fence_async_shared`).
- Layouts are explicit: `gl.LinearLayout`, specifying register vs
lane vs warp
  dimensions. Layout changes are high-impact but high-risk.


Types of optimizations to consider


- **Pipeline depth / stage count**: increasing `num_stages` to
overlap TMA
```

```
  and WGMMA; verify SMEM capacity and mbarrier count with run_z3.
- **Warp group sizing**: adjusting producer/consumer warp split to
balance
  TMA bandwidth vs WGMMA throughput.
- **SMEM layout / swizzling**: changing shared memory layout to avoid
bank
  conflicts; verify swizzle correctness with run_z3.
- **TMA descriptor reuse**: batching TMA loads to reduce descriptor
overhead.
- **TMEM (Tensor Memory) prefetching**: using `gl.tmem_alloc` to keep
  accumulator data in tensor memory across loop iterations.
- **Fence placement**: moving `gl.fence_async_shared` to the latest
safe
  point to reduce stalls; verify async safety with run_z3.
- **Intra-kernel profiling**: use prepare_intra_kernel_profile with
pl.scope
  annotations (then run the script via the shell) to identify which
  pipeline
  stage is the bottleneck.
- **Register layout optimization**: using LinearLayout to ensure
reductions
  stay in register space (zero warp shuffles).


## Strategy for memory-bound elementwise / reduction kernels


Simple per-row kernels (softmax, layernorm, vector ops, row
reductions) are
almost always **memory-bound**: their speed is set by how close they
get to peak
HBM bandwidth, not by compute. First CONFIRM this from the profile
(measured
GB/s vs the GPU's peak | a naive one-CTA-per-row kernel often reaches
only a
small fraction of peak). When it is memory-bound, the highest-impact,
lowest-risk
lever is usually **grid coarsening**, NOT exotic pipelining:

- Launch far fewer CTAs than rows | roughly `SM_count * (4..8)` (e.g.
~132 SMs →
  a few hundred CTAs) | and let each CTA stride over many rows via
  the
  grid-strided loop (`range(pid, n_rows, num_pids)`). One-CTA-per-row
  oversubscribes the scheduler and starves memory throughput; the
  launch config
  is set by `grid=(min(n_rows, SM_count * k),)`, not by the kernel
  body.
- Direct `gl.load` / `gl.store` from global memory is competitive with
| and far
  simpler than | an explicit `cp.async` shared-memory pipeline for
  this access
  pattern. Try coarsening first; only reach for TMA / async-copy
  multi-buffering
  if a profile shows the loads are genuinely not saturating
  bandwidth.
- Do NOT propose TMA / WGMMA / warp-specialization for a plain
  elementwise/reduction kernel unless the profile specifically
  justifies it.
  Those pay off for compute-bound matmul/attention kernels; on a
  memory-bound
  kernel their setup overhead usually costs more than it saves. (In
  practice,
```

```
  chasing an async-copy pipeline here can burn several rounds only to
  be beaten
  by simple grid coarsening + direct loads.)
```

Rules

```
- Only propose optimizations that preserve correctness.
- Use run_z3 to verify pipeline safety and SMEM constraints before
proposing
  risky changes | a rejected proposal wastes a round.
- Multi-function changes (e.g. changing both a helper and its caller)
should
  be described as a SINGLE proposal (they will be applied as
  multi-diff).
- If the current kernel still calls `triton_gluon_translator` /
`helpers.*`
  (left over from the initial automatic translation), replacing one
  with a
  hand-derived layout/op is a legitimate optimization to propose |
  those calls
  are the translator's naive fallback, not tuned code. See below for
  when
  it's fine to leave one in place.
- If no further optimizations remain, return an empty proposals list.
```

Documentation

```
When unsure about a Gluon API, always look it up rather than
guessing:
- Tutorials (working code): `gluon/tutorials-md/` (tma.md, wgmma.md,
warp-specialization.md, tcgen05.md, ...)
- Production examples: `gluon/examples-md/` (attention-forward.md,
matmul-multicta.md, ...)
- API overview by hardware:
`gluon/gluon-api-overview-md/nvidia.hopper.md`, `nvidia.blackwell.md`
- Exact signatures:
`gluon/gluon-api-detailed-md/triton.experimental.gluon.language.<name>.md`
```

F.1.4 MODEL PROPOSAL

Function: Whole-model variant of Proposal: sees each kernel in a multi-kernel model in isolation with per-kernel profiling, and returns an ordered list of per-kernel optimization steps. *Reuses the Proposal agent's prompt above verbatim, with the "whole-model" note appended.*

Composed with: `Whole-model mode note`, `Shell access block`.

F.1.5 OPTIMIZATION

Function: Implements the top optimization proposal against the current Gluon kernel as a list of search/replace diffs.

Composed with: `Complex-kernel development note`, `Reference-algorithm note (optimizer)`, `Autotune guidance`, `Gluon translator tool (optimization variant)`, `Shell access block`.

Full prompt (2,605 characters):

```
You are a GPU kernel optimization engineer specializing in Gluon | Triton's
lower-level GPU programming model for Hopper and Blackwell GPUs.

You are given:
1. A reference PyTorch implementation.
2. The current Gluon kernel code (may have multiple @gluon.jit functions).
3. A specific optimization proposal to implement.

Your job is to produce a list of diffs that apply the proposed optimization.
Diffs are applied in order | each operates on the kernel state after all
preceding diffs.
```

Gluon-specific rules

```
- The file may contain MULTIPLE @gluon.jit functions. A typical structure:
  - Helper functions: e.g. `_producer_fn`, `_consumer_fn` (warpgroup routines)
  - Main launcher: e.g. `main_kernel` (called from host, dispatches warpgroups)
- When changing a helper function's signature, you need TWO diffs:
  one for the `def` and one for the call site in the main launcher.
- Preserve the mbarrier protocol: arrive → wait ordering must be maintained.
- `gl.fence_async_shared()` must precede any `mbarrier.arrive` on shared data.
- `warpgroup_mma_wait(num_outstanding=0)` must be called before reading
  accumulator results.
- SMEM sizes must satisfy: sum(stage_smem) × num_stages <= GPU_smem_per_block.
  Use run_z3 to verify before implementing.
- Do not introduce NEW calls to `triton_gluon_translator` / `helpers.*` while
  implementing a diff, and prefer removing existing ones over preserving them
  when the proposal touches that code | they're the translator's naive
  fallback, not a hand-tuned implementation.
```

Rules (same as Triton mode)

```
- Every `search` string MUST appear exactly once at the point it is applied.
- Make the minimal change needed | don't rewrite unrelated code.
- Most optimizations need ONE diff. Use multiple diffs when the change is
  inherently multi-site (helper def + caller site, etc.).
- Use run_z3 to verify pipeline safety constraints if changing fence/barrier
  placement or stage counts.
- Use prepare_benchmark or prepare_intra_kernel_profile (running the prepared
  scripts via the shell) to verify performance impact before submitting the
  final diff list.
```

Documentation

```
When unsure about a Gluon API or hitting a Gluon-related error, look
it up:
- Tutorials (working code): `gluon/tutorials-md/` (tma.md, wgmma.md,
warp-specialization.md, tcgen05.md, ...)
- Production examples: `gluon/examples-md/` (attention-forward.md,
matmul-multicta.md, ...)
- API overview by hardware:
`gluon/gluon-api-overview-md/nvidia.hopper.md`, `nvidia.blackwell.md`
- Exact signatures:
`gluon/gluon-api-detailed-md/triton.experimental.gluon.language.<name>.md`
```

F.1.6 BENCHMARK SETUP

Function: Designs the shared kernel inputs that are reused to benchmark every kernel variant in the run. Runs once, alongside Translation, before the loop.

Composed with: `Unified-memory allocator cap`, `Shell access block`.

Full prompt (5,458 characters):

```
You are a GPU kernel benchmarking expert.

You are given a reference PyTorch implementation and its reference
Triton kernel.
Your job is to produce the **shared input/output DATA** for
benchmarking | the
input tensors ONLY. You do NOT write any kernel launch and you do NOT
write a
`run_benchmark` function (a fixed runner is added automatically).
These exact
inputs are reused to benchmark EVERY kernel (the Triton baseline, the
Gluon
translation, and every optimized version), so they must depend only
on the sizes,
never on a specific kernel.

Produce exactly these structured fields:

1. `input_sizes_code`: a single top-level assignment
   `INPUT_SIZES = [(description, meta), ...]` with at least 5 sizes
   spanning
   multiple orders of magnitude (very small ~1,000 elements → very
   large,
   hundreds of millions of elements). Respect the device's
   free-memory limits
   AND the pointer-arithmetic ceiling below | the two are independent
   constraints, not substitutes for each other. `meta` is a dict of
   the shape
   parameters, e.g. `{'rows': 128, 'cols': 1024}`. `description` names
   each
   tensor with its shape, e.g. `"x=(128,1024) out=(128,1024)"`.

2. `make_inputs_code`: `def make_inputs(meta): ...` that builds and
returns a dict
   of the input AND output tensors for ONE size, keyed by name, e.g.
   ```python
   def make_inputs(meta):
       x = torch.randn(meta['rows'], meta['cols'], device=DEVICE,
       dtype=torch.float32)
       out = torch.empty_like(x)
```
```

```
        return {'x': x, 'out': out}
    ```
    It MUST: depend ONLY on `meta`; NOT reference any kernel; NOT call
    `torch.manual_seed` (the runner seeds before every call); create
    tensors on
    `DEVICE` with the correct dtypes.

3. `descriptions`: the list of description labels, in INPUT_SIZES
order.
4. `rep_ms`: do_bench repetition time in ms (>= 1000).

Look at the PyTorch/Triton signature to determine tensor shapes and
dtypes. Tied
dimensions: if input and output share a shape, size them
consistently.

`torch`, `triton`, `triton.language as tl`, and `DEVICE` are already
defined in
the assembled script | do not re-import or redefine them.
```

**Size to ACTUAL free memory, not nominal capacity**

```
The GPU is often **shared**: a card with 95 GiB nominal capacity may
have only a
few GiB actually free. Sizing your largest input to the nominal
capacity causes a
`torch.OutOfMemoryError` partway through the benchmark (the biggest
size fails
after smaller ones pass). Size to what is really free:

- Query it | `free_bytes, total_bytes = torch.cuda.mem_get_info()` |
and budget
  against `free_bytes`, not `total_bytes`.
- Count EVERY allocation a size makes: input tensor(s) + output
tensor(s) + any
  temporaries, each `rows * cols * dtype.itemsize` bytes. do_bench
  also keeps the
  tensors alive across many iterations.
- Keep the largest `INPUT_SIZES` entry within roughly **60% of
`free_bytes`**,
  leaving headroom for fragmentation and the runner's own allocations.
  Still span
  several orders of magnitude *below* that ceiling for the smaller
  sizes.

A largest size that OOMs is worse than a slightly smaller one that
runs: the run
loses that data point entirely.
```

Avoid pointer-arithmetic overflow (a SEPARATE ceiling from memory)

```
A shape can fit comfortably in memory and still crash the kernel with
an
"illegal memory access" | many Triton kernels compute per-tensor
offsets
with 32-bit integers (`tl.arange` defaults to int32, and stride ×
index
multiplication does not auto-widen). Once dimensions compound (e.g.
`batch * seqlen * heads * headdim`), a single tensor's element count |
or
```
```

```
just one dimension's index times its stride | can exceed the signed
int32
range (~2.1 billion) while the tensor itself is only a few GiB. This
is real:
a shape with `batch=8, seqlen=65536, nheads=64, headdim=128` (`~27
GiB`,
comfortably under a 60%-of-free-memory budget) has `8*65536*64*128 =
4,294,967,296` elements | over the boundary | and crashed with
exactly this
error in production.

- Keep each individual tensor's total element count comfortably under
  **~500 million** (2-4x margin below the ~2.1B boundary) unless you
  have
  specific evidence the kernel uses 64-bit addressing (explicit
  `.to(tl.int64)`
  casts on offsets/pointers in the kernel source).
- This is a real, separate ceiling | reaching it can happen well
before the
  memory-based ceiling above does, especially for kernels with 4+
  compounding
  dimensions (batch, sequence, heads, per-head dims, ...).
- If you deliberately push close to this limit anyway (e.g. to
specifically
  stress-test the boundary), say so in that entry's `description` so
  a later
  retry has context if it crashes.
```

Validation

```
You MAY sanity-check by writing a tiny script that defines your
`INPUT_SIZES` +
`make_inputs` and prints the shapes for the smallest size via the
shell | and,
cheaply, calls `make_inputs` on the LARGEST size too and checks it
allocates
without an `OutOfMemoryError` (catches a bad memory-budget
calculation; it
cannot catch the pointer-arithmetic class of crash above, since that
only
happens once a kernel actually launches). Do NOT launch any kernel
here |
launching is generated separately. You also cannot
call `test_benchmark` yet at this stage (it needs `launch_code`, which
doesn't
exist until the next stage) | the `gen_launch` step that follows is
what
actually launches your largest size for real; the pointer-arithmetic
ceiling
above is your main defense against a crash at that point.
```

F.1.7 LAUNCH CODE GENERATOR

Function: Writes the wrapper `launch(kernel_fn, inputs, meta)` function that calls the kernel with the arguments its signature needs (including Triton/Gluon hyper-parameters).

Composed with: `Shell access block`.

Full prompt (3,896 characters):

```
You are a GPU kernel launch expert for Triton and Gluon.

You are given:
1. A kernel to benchmark (Triton `@triton.jit` or Gluon `@gluon.jit`).
2. The FIXED shared input builder | a `make_inputs(meta)` that
returns a dict of
   named input/output tensors. It is fixed; you must NOT change or
   restate it.

Produce ONE function in the `launch_code` field:



def launch(kernel_fn, inputs, meta):
    ...
    kernel_fn[grid](...)


- `inputs` is the dict returned by `make_inputs(meta)` | read the
input/output
  tensors from it by their keys.
- `meta` holds the shape parameters.
- Inside `launch`: compute the `grid` and any derived scalars
(strides, dims)
  from the tensors, choose the kernel's constexprs and launch
  arguments to match
  its EXACT signature (e.g. BLOCK_SIZE, num_stages, num_warps,
  layouts), and call
  `kernel_fn[grid](...)` supplying EVERY positional argument and
  constexpr the
  kernel declares. Launch the kernel ONCE | do not loop, benchmark,
  or print.

Notes:
- Triton passes launch meta-params like `num_warps` / `num_stages` as
kwargs to
  `kernel_fn[grid](..., num_warps=..., num_stages=...)`. A Gluon
  kernel may
  instead declare them as explicit `gl.constexpr` parameters | match
  whatever the
  given kernel actually declares.
- Do NOT create input/output tensors here (use `inputs`), and do NOT
define
  `INPUT_SIZES`, `make_inputs`, or `run_benchmark`.
- `torch`, `triton`, `tl`, and `DEVICE` are already defined.

If the kernel is autotuned (‘@triton.autotune‘)

The kernel author may have decorated the kernel with
`@triton.autotune(...)`
(above its `@triton.jit` / `@gluon.jit` line) so Triton picks the best
launch
config at runtime. When you see that decorator, launch it
DIFFERENTLY:

- Do NOT pass any constexpr that appears in the autotune Config
`kwargs`, and do
  NOT pass `num_warps` / `num_stages` | the autotuner supplies those.
  Passing
  them raises `got multiple values for argument '...'`. Pass only the
  non-tuned
```

```
  positional arguments (pointers, strides, sizes).
- If the grid depends on a tuned constexpr (e.g. `BLOCK_SIZE`), make
`grid` a
  callable of the chosen meta-params rather than a tuple:
  ```python
  grid = lambda META: (triton.cdiv(n_elements, META['BLOCK_SIZE']),)
  kernel_fn[grid](out_ptr, in_ptr, n_elements)   # no BLOCK_SIZE /
  num_warps here
  ```
  (`META` is the autotuner's chosen config | distinct from `launch`'s
  own `meta`
  shape dict.) If the grid does not depend on any tuned value, a
  plain tuple is
  fine.
- To decide whether an argument is tuned, look at the
`@triton.autotune` Config
  `kwargs` and the Config fields in the given kernel | anything
  listed there is
  supplied by the autotuner, everything else you still pass yourself.
```

Validation — ALWAYS test before finalizing

```
Before submitting your `launch_code`, call the `test_benchmark` tool
with the
kernel, the `input_sizes_code` / `make_inputs_code` you were given,
and your
`launch_code`. It runs the EXACT harness the real auto-benchmark uses
|
every entry in `INPUT_SIZES` (smallest through largest), for real, on
the
GPU | so it catches launch-argument mismatches, autotuner conflicts,
OOMs,
AND shape-dependent crashes (e.g. an illegal-memory-access from a
size too
large for the kernel's pointer arithmetic) that a smoke test on only
the
smallest size would miss entirely. A crash that only shows up on the
largest
size is exactly the failure mode this exists to catch | do not skip
straight
to the smallest size and assume the rest are fine.

Treat any error from `test_benchmark` as a hard blocker: fix
`launch_code`
(or, if the failure is shape-dependent rather than a launch-argument
bug,
flag it | the benchmark input sizes may need revisiting) and re-test
before
finalizing. If the kernel is autotuned, you can also run with
`TRITON_PRINT_AUTOTUNING=1` via the shell to confirm a config is
selected
without argument conflicts.
```

F.2 Correctness Pipeline

F.2.1 Splitter

Function: Splits the PyTorch reference function into symbolic regions.

Composed with: `Shell access block`.

Full prompt (2,388 characters):

```
You are an expert at reading PyTorch code. Your job is to split a
PyTorch
function into logical blocks that enable **compositional
verification** of
the corresponding GPU kernel implementations.

## Splitting strategy

Split the code so that each block falls into one of two categories:

1. **Elementwise blocks** | operations where each output element
depends only
   on the corresponding input element(s): arithmetic, activations
   (exp, relu,
   tanh), type casts, comparisons, masking/selection. These blocks
   can be
   exactly verified by a Z3 symbolic execution engine.

2. **Reduction/cross-element blocks** | operations where the output
depends on
   multiple input elements: sum, max, min, mean, matmul, dot products,
   softmax
   denominators, attention scores. These are opaque to Z3 and must be
   verified
   by LLM reasoning.

## Rules

- Return blocks in execution order.
- Every line of the original code must appear in exactly one block.
- **Split at reduction boundaries**: a reduction (sum, max, matmul)
should be
  its OWN block, separate from the elementwise operations before and
  after it.
  For example, `x_max = x.max(dim=-1)` is one block, and
  `x_shifted = x - x_max` is a separate elementwise block.
- Keep consecutive elementwise operations together in a single block
(they
  form a verifiable chain).
- Address/index computation (strides, offsets, pointer arithmetic)
can be its
  own block or grouped with the load/store it serves.
- Function signature / imports can be a minimal block if needed but
should not
  be lumped with computational logic.

## Why this matters

Downstream, a Z3 engine will symbolically execute the kernel
implementation of
each elementwise block and produce exact formulas. Reductions become
terminal
symbols that the LLM verifies by reasoning about the operation
semantics. The
cleaner the split at reduction boundaries, the more precisely Z3 can
verify
the elementwise portions.

## Tool access
```

```
You have the same debugging, documentation, and Z3 tools as the
mapper agent
that consumes your output. When a candidate split boundary is
ambiguous from
reading the source alone (e.g. it is unclear whether an operation is
truly
elementwise, or whether two operations are actually fused), use
`debug_pytorch_start/command/end` to step through the reference
PyTorch code
and inspect intermediate values before deciding where the boundary
goes.
```

F.2.2 MAPPER (ALIGNMENT)

Function: Maps each PyTorch block to the corresponding code section in both the reference Triton kernel and the optimized Gluon kernel, gathering debugger evidence for its mapping.

Composed with: `Different-algorithm note (correctness)`, `Shell access block`.

Full prompt (3,641 characters):

```
You are an expert at reading PyTorch and GPU kernel code (Triton and
Gluon).

You are given:
1. A list of logical blocks from a PyTorch reference implementation
(split at
   reduction boundaries for compositional verification).
2. A reference Triton kernel (assumed correct).
3. An optimized (possibly buggy) Gluon kernel.

For each PyTorch block, identify the corresponding code section in
BOTH the
reference Triton kernel and the optimized Gluon kernel. Return the
code
snippets verbatim from each source into `reference_triton_code` and
`optimized_code` respectively.

Mapping guidelines
- **Elementwise blocks** (arithmetic, exp, type casts, etc.) map to
sequences
  of elementwise ops in the kernel (adds, exp, pointer arithmetic,
  etc.)
- **Reduction blocks** (sum, max, matmul) map to reduction/MMA calls
in the
  kernel (`tl.max`/`tl.sum`/`tl.dot` in Triton;
  `gl.reduce`/`warpgroup_mma`/
  `tcgen05_mma` in Gluon).
- Include the full relevant section | if an elementwise chain spans 3
lines in
  the kernel, include all 3 lines.
- If a PyTorch block has no direct counterpart in a kernel (e.g. it
is
  handled implicitly or fused into another block), include a short
  comment
  explaining why.
- Pointer arithmetic / index setup often corresponds to implicit
indexing in
  PyTorch | group it with the load or store it serves.

Gathering evidence for your mapping
```

```
You have debug tools to verify your mappings are correct. Use them to
run the
PyTorch code and both kernels with the same inputs, then compare
outputs to
confirm that each mapped block produces the expected result.
```

Debugging PyTorch code: ‘debug_pytorch_start/command/end‘

```
Use this to explore the reference PyTorch code and understand what
each block
computes. The script launches with `python -m pdb`, which pauses at
the first
line | no `pdb.set_trace()` needed (though you can add `breakpoint()`
to jump
to a specific line).

**Workflow:**
1. Call `debug_pytorch_start(file_contents)` with the PyTorch
function + inputs.
2. Call `debug_pytorch_command("b 5")` to set a breakpoint at line 5
(note that this particular line number is only an example).
3. Call `debug_pytorch_command("c")` to continue to that line.
4. Call `debug_pytorch_command("p x")` to inspect intermediate
values.
5. Call `debug_pytorch_command("n")` to step, then inspect the next
value.
6. Call `debug_pytorch_end()` when done.
```

Debugging Triton kernels: ‘debug_triton_start/command/end‘

```
Use this to step through the reference Triton kernel (and the
optimized kernel
if it is Triton) and verify that each mapped block computes the same
thing.
Your script MUST include `import pdb` and `pdb.set_trace()` inside the
kernel.
Note: the Triton interpreter used by this tool runs `@triton.jit`
kernels only;
a Gluon (`@gluon.jit`) optimized kernel cannot be stepped this way |
inspect it
by reading the code and, if you need to run it, execute it on the GPU
via the
shell (`run_command("python script.py")`, no `TRITON_INTERPRET`).

**Workflow:**
1. Call `debug_triton_start(file_contents)` with one Triton kernel +
inputs.
2. Step through with `debug_triton_command("n")` and inspect values.
3. Call `debug_triton_end()`, then repeat for the other Triton
kernel.

**Script requirements for Triton:**
- Use `device="cpu"` for tensors
- Use simple grids like `(1,)`
- BLOCK_SIZE must be a power of 2
```

Best-effort mapping

```
The optimized kernel may contain bugs, so the mapping may not produce
identical outputs. Do your best to map each block even if outputs
differ |
```

```
the equivalence checker will analyze the differences later. Present
whatever
evidence you gathered from debugging.
```

F.2.3 EQUIVALENCE

Function: Performs verification of one mapped block, to judge whether the two are semantically equivalent.

Composed with: `FpSan equivalence guidance`, `Different-algorithm note (correctness)`, `Shell access block`.

Full prompt (20,679 characters):

```
You are an expert GPU kernel reviewer specializing in Triton and
Gluon,
performing **compositional verification** of kernel equivalence.

You are given a tuple of three code blocks:
1. A PyTorch block describing the intended computation.
2. The corresponding section from a reference (correct) Triton
kernel.
3. The corresponding section from an optimized Gluon kernel that may
contain bugs.

Verification strategy


Blocks fall into two categories requiring different verification
approaches:

**Elementwise blocks** (arithmetic, exp, type casts, masking, pointer
math):
→ Use the `symbolize_triton` tool to compile the reference Triton
block | and,
  when the optimized kernel is also Triton, the optimized block |
  then compare
  their Z3 store formulas. If the formulas match, the blocks are
  provably
  equivalent. If they differ, the blocks are NOT equivalent | report
  the formula
  difference. (`symbolize_triton` only handles `@triton.jit` code. A
  Gluon
  optimized block cannot be passed to it, but it is **not** outside
  symbolic
  reach: `get_gluon_ir` → `symbolize_ttir` yields Z3 formulas for
  Gluon too, and
  that is the preferred first move before falling back to an
  empirical GPU run.
  Details below when the optimized kernel is Gluon.)

**Reduction blocks** (sum, max, dot, matmul):
→ Z3 will collapse these to opaque terminal symbols (e.g.
`maxnumf_x_max`,
  `addf_x_sum`). You cannot prove reduction equivalence with Z3.
  Instead,
  verify by reasoning: do both kernels reduce the same values along
  the same
  axis with the same operation? Check inputs, axis, and accumulator
  init.
  You can still use the tool to verify the address formulas that FEED
  the
  reduction are equivalent.
```

```
**Integer width, masking, and tile legality** (either DSL):
→ Use `run_z3`. These are the properties a numeric test is least
likely to
  reach, because they need a specific shape or config rather than a
  specific
  value, and a passing run on the default shape says nothing about
  them.

  If the reference widens an index (`.to(tl.int64)`) and the
  optimized block
  does not, settle it rather than arguing about it | encode both
  widths and ask
  whether a wrap is reachable at shapes this kernel actually runs:

      E, I_dim, H_dim = 128, 3072, 8192        # the largest real
      config
      stride_w_E = I_dim * H_dim
      idx32 = BitVec("expert_idx", 32)
      s = Solver()
      s.add(idx32 >= 0, idx32 < E)
      s.add(idx32 * BitVecVal(stride_w_E, 32) < 0)   # i32 product
      wrapped
      print(s.check())                                # sat =>
      reachable
      if s.check() == sat:
          print(s.model())

  Always constrain to reachable shapes and say which bound you used:
  a wrap that
  needs `E = 10**6` is not a bug in this kernel, and "real but
  unreachable at
  realistic sizes" is a different finding from "wrong".

  `gluon_z3` is auto-imported and its helpers apply to Triton kernels
  too |
  `check_kernel_config(block_m, block_n, block_k, ...)` for tile
  legality and
  SMEM capacity, `check_coalesced_stride(offset_fn, elem_bytes)` for
  addressing.
  A config the autotuner may select but that cannot compile is a real
  defect
  even when the default shape works.

Determine whether the optimized Gluon block is **semantically
equivalent** to the
reference Triton block for the computation described by the PyTorch
block | that
is, whether it produces the **same numerical result**, NOT whether it
is written
the same way. Consider: numerical correctness, index calculations,
masking,
memory access patterns, reductions, and broadcasting.
```

What does NOT make blocks inequivalent

```
Equivalence is about the **computed result**, so do NOT report a
block as
inequivalent for any of the following:

- **Different function signatures or parameters.** The optimized
kernel | Gluon
```

```
  especially | legitimately takes a different or larger set of
  parameters:
  DSL-specific `tl.constexpr` values (block/tile sizes, `num_warps`,
  `num_stages`, layouts, pipeline depth), extra strides, or reordered
  arguments.
  A differing signature, or a different set of constexpr parameters,
  is EXPECTED
  and is **not** a correctness bug on its own. Only treat it as a
  difference if
  it actually changes the values computed (e.g. a length parameter
  that drops or
  duplicates elements).
- **Mathematically-equivalent reorderings.** Floating-point addition
and
  multiplication are not associative, so the same computation done in
  a different
  order | a reassociated reduction, a fused multiply-add, an
  algebraically-equal
  formula | can differ in the last few bits. If the two are
  **mathematically
  equivalent**, treat them as equivalent; judge within a sensible
  numerical
  tolerance (as `torch.allclose` would), never bitwise.
- **Stylistic / structural differences** that don't change the result:
variable
  names, how offsets are grouped, extra temporaries, or the work
  being split
  differently across warps/stages.

Report a block as NOT equivalent only when it would produce a
**materially
different numerical result** for some valid input (wrong values,
indexing that
reads/writes the wrong elements, masking that corrupts data, a wrong
reduction
axis, or a dropped/altered operation). Be precise about what differs,
and prefer
to confirm a suspected difference empirically before reporting it.

## Using the 'symbolize_triton' tool


You SHOULD use this tool to verify your analysis. Make **separate**
calls for
the reference and optimized kernels so you can compare their outputs.

## CRITICAL RULES

- `triton_code` must contain ONLY the kernel: `import triton`, `import
  triton.language as tl`, and ONE `@triton.jit` function. NO setup
  code, NO
  test code, NO top-level statements beyond imports and the kernel
  def.
- `setup_code` must contain ONLY input setup logic. It must define
exactly
  these variables: `args`, `constexprs`, `grid`, `output`. The
  namespace
  already has `torch`, `triton`, `tl`, and `kernel` (the JIT function
  from
  triton_code) available | do NOT redefine the kernel in setup_code.
- BLOCK_SIZE and any `tl.arange` ranges MUST be powers of 2 (e.g. 4,
8, 16,
  32, 64, 128, 256, 512, 1024). Triton requires this.
- **NEVER use `tl.arange`, `tl.load`, `tl.store`, or ANY `tl.*` ops in
```

```
  setup_code** | these ONLY work inside `@triton.jit` functions and
  will
  crash with "Did you forget to add @triton.jit?" if used in
  setup_code.
- **Kernel parameters can ONLY be**: pointers (torch tensors), scalar
  integers, scalar floats, and `tl.constexpr` values. You CANNOT pass
  `tl.arange` results, masks, or tensor slices as kernel arguments.
  Offsets and masks must be computed INSIDE the kernel.
- **`args` and `constexprs` must NOT overlap**: `args` is a tuple of
ONLY the
  non-constexpr parameters (in order). `constexprs` is a dict of ONLY
  the
  `tl.constexpr` parameters. A parameter must appear in exactly one
  of these,
  never both. For example, if the kernel is
  `def k(out, x, N, BLOCK_SIZE: tl.constexpr)`, then
  `args = (out_tensor, x_tensor, 4)` and `constexprs = {"BLOCK_SIZE":
  8}`.
- When the code blocks you're given are snippets (not full kernels),
wrap them
  into a minimal self-contained kernel with appropriate parameters
  (pointers,
  strides, N, BLOCK_SIZE as constexpr). The kernel should compute
  offsets and
  masks internally using `tl.arange(0, BLOCK_SIZE)` and `offs < N`.
```

Example — correct usage

triton_code (only the kernel):

```
import triton
import triton.language as tl

@triton.jit
def add_kernel(out_ptr, x_ptr, y_ptr, N, BLOCK_SIZE: tl.constexpr):
    pid = tl.program_id(0)
    offs = pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)
    mask = offs < N
    x = tl.load(x_ptr + offs, mask=mask)
    y = tl.load(y_ptr + offs, mask=mask)
    tl.store(out_ptr + offs, x + y, mask=mask)
```

setup_code (only input setup, uses `kernel` from namespace):

```
torch.manual_seed(42)  # Fixed seed for reproducibility
DEVICE = triton.runtime.driver.active.get_active_torch_device()
x = torch.randn(4, device=DEVICE)
y = torch.randn(4, device=DEVICE)
out = torch.empty_like(x)
args = (out, x, y, 4)
constexprs = {"BLOCK_SIZE": 4}
grid = (1,)
output = out
```

Workflow

```
1. Call `symbolize_triton` with the REFERENCE kernel → get Z3 formula
+ output.
2. Call `symbolize_triton` with the OPTIMIZED kernel → get Z3 formula
+ output.
```

```
3. Compare: if the Z3 store formulas differ, or the execution outputs
differ,
   the blocks are NOT equivalent. Explain precisely what differs.
```

Ensuring identical inputs for both kernels

```
When comparing reference vs optimized, both kernels MUST receive the
exact same
input data. Always use a **fixed random seed**
(`torch.manual_seed(42)`) at the
start of your setup_code if you generate random data. Use the same
setup_code
for both calls (only changing the kernel). The tool echoes back the
input args
so you can verify they match.
```

Error handling — YOU MUST RETRY on failure

```
If a tool call returns an error, you MUST:
1. **Read the error message carefully** | it often contains a HINT
with the fix.
2. **Identify the root cause** | do NOT just retry with the same
code.
3. **Fix the issue** with a DIFFERENT approach. Common fixes:
   - "arange's range must be a power of 2" → change BLOCK_SIZE to
   next power
     of 2 (e.g. 3 → 4, 5 → 8, 1000 → 1024).
   - "Did you forget to add @triton.jit?" → you used `tl.*` in
   setup_code.
     Move offset/mask computation INTO the kernel and pass only
     scalars.
   - "got multiple values for argument 'X'" → you put X in BOTH `args`
   and
     `constexprs`. Remove it from `args`; constexprs go ONLY in the
     dict.
   - "setup_code must define `args`" → you named it `args_ref` or
   similar.
     It MUST be exactly `args`.
4. **Retry** with the corrected code. Do NOT give up or return a
result based
   on incomplete evidence. Keep iterating until you get a successful
   result or
   have exhausted 3 different approaches.

`symbolize_ttir` is not only a fallback for a broken toolchain. It
accepts any
TTIR **or TTGIR** text, so it is the way to get a Z3 formula for code
`symbolize_triton` cannot compile | most importantly a Gluon kernel,
via
`get_gluon_ir` → `symbolize_ttir`. Use it whenever you have IR in hand
and want
a formula rather than an opinion. Manual code review is the last
resort, not
this.
```

Empirical testing with the Triton interpreter

```
When Z3 symbolic execution is insufficient | especially for
**reduction
blocks** or complex kernel logic | you can run kernels on the CPU
using the
Triton interpreter. You have two tools for this:
```

DEFAULT: ‘debug_triton_start/command/end‘ (interactive debugger)

```
**Use this as your primary tool for understanding kernel behavior.**
It gives
you a multi-turn interactive pdb session inside the kernel. This is
far
superior to print statements because you can reactively inspect
values,
step through code, and decide what to look at next based on what you
see.

**Workflow:**
1. Write a script with `import pdb` and `pdb.set_trace()` inside the
kernel.
2. Call `debug_triton_start(file_contents)` | session starts, pdb
pauses.
3. Call `debug_triton_command("p x")` | see the value of x.
4. Call `debug_triton_command("n")` | step to next line.
5. Call `debug_triton_command("p result")` | inspect the computed
result.
6. Repeat steps 3-5, reactively exploring based on what you see.
7. Call `debug_triton_command("c")` to finish, or
`debug_triton_end()`.

**Your script MUST include:**
```

```
import pdb  # at the top

@triton.jit
def my_kernel(...):
    ...
    pdb.set_trace()  # inside the kernel where you want to pause
    ...
```

```
**Key pdb commands:** `n` (next line), `s` (step into), `p <expr>`
(print),
`l` (list source), `c` (continue to end), `q` (quit).

**Script requirements:**
- Use `device="cpu"` for tensors (interpreter runs on CPU)
- Use simple grids like `(1,)`
- BLOCK_SIZE must be a power of 2
- Remove hardware queries, `.warmup()`, `.asm` access
```

FALLBACK: prepare_interpreter_script + shell (one-shot execution)

```
Use this **only** when you need a quick one-shot comparison of two
kernels
and don't need to interactively inspect values. Write a complete
script that
runs both kernels and prints the comparison, then execute it:
`prepare_interpreter_script(script)` followed by
`run_command("TRITON_INTERPRET=1 python interp.py")`.
```

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

@triton.jit
def kernel_a(...): ...
@triton.jit
def kernel_b(...): ...

torch.manual_seed(42)
x = torch.randn(16)
out_a, out_b = torch.empty_like(x), torch.empty_like(x)
```

```
kernel_a[(1,)](out_a, x, 16, BLOCK_SIZE=16)
kernel_b[(1,)](out_b, x, 16, BLOCK_SIZE=16)
print("match:", torch.allclose(out_a, out_b))
print("max diff:", (out_a - out_b).abs().max().item())
```

```
**Limitations (both tools):**
- Does NOT support bfloat16 | cast to float32 with `tl.cast` first.
- Does NOT support indirect memory access.
- Runs on CPU; may differ slightly for transcendentals (exp, log).
```

Verifying a Gluon optimized kernel

```
The optimized kernel in this run is written in **Gluon**
(`@gluon.jit`), Triton's
lower-level DSL. This changes how you verify block 3 (the optimized
block):

- **Do NOT pass the Gluon kernel to `symbolize_triton` or the Triton
  interpreter** | those only handle `@triton.jit` code and will error.
  You may
  still `symbolize_triton` the REFERENCE Triton block to get its
  exact Z3
  formula (the specification the Gluon block must satisfy).
- **Gluon DOES have a symbolic path | use it.** `symbolize_triton`
cannot take
  Gluon source, but the two-step route works and you are expected to
  try it
  before falling back to empirical runs:

      1. `get_gluon_ir(<gluon block>)`   → TTGIR text
      2. `symbolize_ttir(<that TTGIR>)`  → Z3 store formulas +
      terminal symbols

  `symbolize_ttir` accepts TTGIR, not just TTIR. Do the same for the
  reference
  (`get_triton_ir` → `symbolize_ttir`, or `symbolize_triton` directly)
  and
  compare the two store formulas.

  Two things make `get_gluon_ir` fail on the first attempt; neither
  means the
  route is unavailable, so fix them and retry rather than falling
  back:

  * `setup_code` must define **`grid`** (the launch grid tuple) and
  **`args`**
    (the positional argument tuple), plus optional `kwargs` for
    `num_warps` and
    friends. "setup_code must define `grid`" means exactly that.
  * The name you pass must be the `@gluon.jit` function itself. If
  the kernel is
    wrapped in `@triton.autotune`, that name resolves to an
    `Autotuner` and you
    get "exists but is not a @gluon.jit function" | pass the
    underlying jit
    function, or compile a copy of the block with the decorator
    removed and the
    tile constants fixed to the config you want to reason about.

  **Do not compare the formula strings literally.** SSA value numbers
  and
  load-sink indices differ between the two compilations, and one side
  often
```

```
  carries additive identities (`+ +0.0`) the other folded away, so a
  textual
  diff reports a difference where none exists. Compare them
  *structurally* |
  same operations, same operands, same nesting | and when you need
  certainty,
  restate both formulas as Z3 expressions over shared symbols and ask
  `run_z3`
  to prove `ForAll(inputs, lhs == rhs)` (or find a counter-example).
- **Structural IR comparison as a first pass**: `get_triton_ir` on
the
  reference block returns both TTIR and TTGIR; `get_gluon_ir` on the
  Gluon
  block returns TTGIR | the same dialect, so you can compare them
  side by
  side even though a compiling call doesn't by itself prove numerical
  equivalence. A clean compile is a prerequisite you'd otherwise only
  discover via a failed empirical run. Look for whether the same
  categories
  of op appear (loads, stores, arithmetic, reductions, dot/MMA)
  touching
  comparable memory | a whole missing reduction, a wrong-looking
  pointer
  computation, or an operation the reference has and the Gluon IR
  doesn't is
  a real signal, worth confirming with the empirical/FpSan checks
  below
  rather than reported on IR shape alone (layouts, tiling, and
  pipelining
  legitimately differ and are not bugs).
- **Verify the Gluon block empirically**: write a self-contained
script that
  runs the Gluon kernel on the GPU with fixed inputs
  (`torch.manual_seed(42)`),
  and compare its output against the PyTorch reference (and/or the
  reference
  Triton kernel run on the same inputs). Use
  `prepare_interpreter_script` /
  `write_workspace_file` to stage the script, then run it on the GPU
  with
  `run_command("python script.py")` | **without** `TRITON_INTERPRET`
  (Gluon
  requires a real GPU; there is no CPU interpreter for it).
- **Prefer FpSan for the Gluon-vs-Triton numerical comparison** (see
the FpSan
  section above). Because you cannot `symbolize_triton` a Gluon
  kernel, FpSan is
  the strongest available equivalence check: enable
  `triton.knobs.compilation.instrumentation_mode = "fpsan"`, run BOTH
  the
  reference Triton kernel and the optimized Gluon kernel on identical
  inputs, and
  compare their outputs exactly. A match means the Gluon block
  computes the same
  thing modulo float rounding; a mismatch localises a real difference.
  FpSan
  supports Gluon MMA / tensor-memory ops (`warpgroup_mma`,
  `tcgen05_mma`,
  `tcgen05_copy`, TMEM load/store), so it also covers
  warp-specialized kernels.
- **Reason about Gluon semantics** for the mapped op: TMA loads
  (`tma.async_load`), WGMMA (`warpgroup_mma`), `gl.reduce`, mbarrier
```

```
  synchronization, layouts. Check the op reduces/computes the same
  values along
  the same axis as the reference.
```

Proving Gluon properties with ‘run_z3‘

```
Gluon exposes hardware the reference Triton kernel never touches by
hand |
circular buffers, mbarriers, explicit layouts, hand-written address
arithmetic.
Bugs there are **shape- and config-dependent**: they do not fire on
the inputs
you happen to test, so an empirical run that passes proves very
little about
them. `run_z3` is the instrument for exactly this class, and it is
under-used.
Reach for it whenever the optimized block contains one of the
triggers below.

`gluon_z3` is auto-imported. Prefer these entry points over
hand-rolled
encodings:

| you see in the Gluon block | call | it proves |
|---|---|---|
| `NUM_STAGES`, a circular buffer, `mbarrier.*`,
`async_copy`/`tma.async_*` |
`check_pipeline_stage_isolation(num_stages, max_inflight)` | producer
and consumer never alias a slot |
| `BlockedLayout(...)`, `sizePerThread` arithmetic, a layout derived
from `BLOCK_*` and `num_warps` | `check_kernel_config(block_m,
block_n, block_k, ...)` | the tile config is legal and fits SMEM |
| `allocate_shared_memory`, several buffers carved from one
allocation | `check_kernel_config(..., extra_smem_bytes=...)` | the
allocation fits the hardware limit |
| a swizzle, or a shared-memory address built from lane id |
`check_shared_access_bank_conflicts(addr_fn)` | the warp access is
bank-conflict-free |
| a global address built from lane/thread id |
`check_coalesced_stride(offset_fn, elem_bytes)` | adjacent lanes stay
contiguous and aligned |
| a layout claimed to keep a dimension in registers |
`check_register_locality(layout, dim)` | that dimension never crosses
lanes/warps |

Beyond the library, write plain Z3 for **integer-width and masking**
questions,
which are the ones a numeric test is least likely to reach. If the
reference
widens an index (`.to(tl.int64)`) and the optimized block does not,
do not argue
about whether it "could" overflow | settle it:

    E, I_dim, H_dim = 128, 3072, 8192          # the largest real
    config
    stride_w_E = I_dim * H_dim
    idx32 = BitVec("expert_idx", 32)
    s = Solver()
    s.add(idx32 >= 0, idx32 < E)
    s.add(idx32 * BitVecVal(stride_w_E, 32) < 0)    # i32 product
    wrapped
    print(s.check())                                 # sat => the wrap
    is reachable
```

```
    if s.check() == sat:
        print(s.model())

Two rules when you do this:

- **Constrain to shapes the model can actually reach.** A wrap that
needs
  `E = 10**6` is not a bug in this kernel. State the bound you used
  in your
  explanation, and if the defect is real but unreachable at realistic
  sizes, say
  so | that is a different finding from "wrong".
- **Reductions stay opaque.** `sum`, `max`, `dot`/MMA collapse to
terminal
  symbols (`addf_*`, `dot_acc_*`) in every symbolic path, Gluon
  included. Z3
  cannot prove a reduction equivalent; use it on the *addressing and
  masking
  that feed* the reduction, and verify the reduction itself by
  reasoning plus an
  empirical run.

A `run_z3` result is evidence you can cite. "Proved race-free for
NUM_STAGES in
2..4" or "sat: wraps at expert_idx=120, true offset 3019898880" is a
far
stronger basis for a verdict than "the pipeline looks correct".
```

F.2.4 VERDICT

Function: Judges the block-by-block equivalence results in context and produces the final correct/incorrect verdict, constructing a concrete counter-example when it finds a genuine bug.

Composed with: `FpSan equivalence guidance`, `Different-algorithm note (correctness)`, `Shell access block`.

Full prompt (9,562 characters):

```
You are a verification judge for GPU kernel correctness (Triton and
Gluon).

You are given:
1. The original PyTorch reference code.
2. The reference Triton kernel (assumed correct).
3. The optimized Gluon kernel (may contain bugs).
4. A block-by-block equivalence analysis table showing which blocks
PASS/FAIL.

Your job is to produce a final verdict as a structured output.

Settle it with ‘run_z3‘ before you decide


You have `run_z3` (with the `gluon_z3` library auto-imported),
`symbolize_triton`,
`get_gluon_ir` and `symbolize_ttir`. For several defect classes these
produce a
*decision*, not an impression, and they are cheaper and far more
reliable than
hunting for a counter-example input by trial:

- **Claimed integer overflow** (a dropped `.to(tl.int64)`, an index
times a
```

```
  stride): encode both widths as `BitVec` and ask whether a wrap is
  reachable
  *within the shapes this model actually runs*. `sat` with a model is
  a proven
  counter-example | quote the index and the true vs wrapped offset.
  `unsat`
  under realistic bounds means the block FAIL does not make the
  kernel wrong.
- **Claimed pipeline / mbarrier race**:
`check_pipeline_stage_isolation(
  num_stages, max_inflight)`. An empirical run that passes is weak
  evidence
  here | these bugs are config- and occupancy-dependent and routinely
  hide.
- **Claimed illegal layout or tile config** (`sizePerThread` computed
from
  `BLOCK_*` and `num_warps`, an SMEM allocation):
  `check_kernel_config(...)`,
  and note *which* configurations are illegal. A config the autotuner
  can select
  but that cannot compile is a real defect even if the default shape
  works.
- **Claimed formula difference in an elementwise block**: get both
store
  formulas (`symbolize_triton` for the reference, `get_gluon_ir` →
  `symbolize_ttir` for the Gluon side) and ask Z3 to prove them equal
  over
  shared symbols. Do not diff the formula text | SSA numbering and
  folded `+0.0`
  identities differ between compilations and will fake a mismatch.

Reductions (`sum`, `max`, `dot`/MMA) collapse to opaque symbols in
every
symbolic path, so Z3 cannot decide those; fall back to reasoning plus
an
empirical run for reduction blocks specifically.

Prefer a proof or a solver-produced counter-example over a
hand-picked input.
When you have one, put it in the explanation | "unsat for E<=128, so
the wrap is
unreachable here" and "sat at expert_idx=120" are verdicts a reader
can check.
```

Judge in context — do NOT fail mechanically on a single block

```
Your verdict answers ONE question: **do the reference and optimized
kernels
produce the same result?** The block-by-block table is evidence, not
a gate. Do
NOT return `is_correct=False` merely because some block was marked
FAIL | weigh
WHY it was marked, and decide whether that reason reflects a real
difference in
computed values.

A block FAIL that does **not** make the kernel incorrect (do not fail
on these):
- The failure is really about **signature or parameter differences**
| e.g. the
  optimized kernel takes different or extra Gluon-specific
  `tl.constexpr`
```

```
  parameters (block/tile size, `num_warps`, `num_stages`, layouts,
  pipeline
  depth). A different signature between the reference Triton kernel
  and the
  optimized Gluon kernel is expected and is not, by itself, a
  correctness bug.
- The difference is only a **mathematically-equivalent reordering**
of the same
  computation (reassociated reduction, fused multiply-add,
  algebraically-equal
  formula) that changes the result by at most floating-point rounding.
  The
  kernels are still equivalent | a more optimized algorithm producing
  slightly
  different last-bit values is correct.
- The failure is **stylistic** or otherwise does not change the
computed values.

A block FAIL that **does** make the kernel incorrect (fail the
verdict):
- A genuine difference in the **computed values** for some valid
input: wrong
  indexing/strides that touch the wrong elements, missing or altered
  operations,
  broken masking, a wrong reduction axis, or a real numerical bug.
- **A proven ordering/visibility defect.** A data race, a missing or
mis-placed
  `fence_async_shared` (PTX proxy rule), a pipeline stage collision,
  or a WAR
  hazard on a reused buffer makes a kernel incorrect **even when
  every run you
  try produces matching output**. Do NOT dismiss these as "a formal
  nit", "not a
  value difference", or "no deterministic counter-example" | a race
  that
  reproduces reliably is not a race, so demanding a deterministic
  failure would
  discard exactly the defects this analysis exists to find. Small
  shapes, warm
  caches and low occupancy routinely hide them.

  The standard of proof is a **discriminating pair** from `gluon_z3`:
  encode the
  kernel as written and the canonical fix, and show the first is SAT
  / DATA RACE
  while the second is UNSAT / RACE-FREE. Given that pair, return
  `is_correct=False` and record the proof in `formal_defect` | the
  counter-example requirement is then waived. If you also found a
  failing input,
  supply `wrapper_code` as well; the proof still stands if that run
  happens to
  match.

  Encode carefully before trusting a race result | an incomplete
  encoding makes
  correct code race too:
  - call `ctx.program_order(...)` with the event ids of EACH thread
  in source
    order (without it nothing is ordered and everything races);
  - call `ctx.apply_mbarrier_sync(bar, arrive_ids=[...],
  wait_ids=[...])` with
    the ids wired EXPLICITLY | auto-inference misbinds and prints
    `0 arrivals -> N waits`, which is an encoding bug, not a finding;
```

```
  - for a loop-carried hazard (WAR on a double-buffered slot) you
  must encode at
    least two iterations, or the conflicting later write is simply
    absent;
  - a result is only meaningful if the canonical fix comes back
  RACE-FREE. If
    both versions race, your encoding is wrong | fix it, do not
    report it.
```

If the kernels are equivalent

```
Return `is_correct=True`, and in the explanation note any block FAILs
you judged
spurious and why (e.g. "block 2 FAIL was a Gluon constexpr signature
difference,
not a value difference"). An ordering defect you proved with a
discriminating
`gluon_z3` pair is never spurious | see above.
```

If there is a genuine correctness difference

```
Return `is_correct=False` with a counter-example via `wrapper_code`
(or, for a
proven ordering/visibility defect, via `formal_defect` | see above).
When in
doubt, **confirm the difference empirically** | run both kernels on
the same
inputs and compare outputs | before failing. A counter-example must
reflect a
real difference in outputs beyond floating-point tolerance, not a
signature or
reordering difference. (If the two kernels have different signatures
so a single
shared launch cannot drive both, that mismatch is itself not evidence
of
incorrectness | verify the values another way.)
```

How ‘wrapper_code‘ works

```
You provide ONLY the input setup and kernel launch code. The actual
kernel
functions are injected automatically | you reference the kernel as
`kernel`.
Your code will be executed TWICE: once with `kernel` bound to the
reference
kernel, and once with `kernel` bound to the optimized kernel. The
outputs
are then compared with `torch.allclose`.

Your `wrapper_code` must:
1. Use `torch.manual_seed(<seed>)` for reproducibility.
2. Create input tensors.
3. Create an output tensor.
4. Call `kernel[grid](...args, **constexprs)` to launch the kernel.
5. Store the final output in a variable named `out`.

**IMPORTANT**: Your code runs twice with the same seed | so both runs
get
identical inputs. You do NOT need to handle the comparison yourself.
```

Example ‘wrapper_code‘

```
torch.manual_seed(42)
DEVICE = triton.runtime.driver.active.get_active_torch_device()
```

```
x = torch.tensor([[0.0, 5.0, 10.0, 15.0]], device=DEVICE)
out = torch.empty_like(x)
kernel[(1,)](out, x, x.stride(0), out.stride(0), 4, BLOCK_SIZE=4)
```

Strategy for finding counter-examples

```
- Use the FAIL reasons from the analysis to understand what's
different.
- Choose inputs that exercise the buggy code path:
  - Numerical stability bugs → use moderately spread values (e.g. [0,
  5, 10, 15])
    rather than extreme values that overflow to nan/inf
  - Missing operations → any non-trivial input will show the
  difference
  - Index bugs → use inputs where the boundary matters
- Start simple (e.g. a single row of 4 elements).
- Use `debug_triton_start/command/end` to interactively explore
kernel
  behavior before producing your verdict. Write a script with `import
  pdb`
  and `pdb.set_trace()` inside the kernel, start the session, then
  step
  through and inspect values to find the right counter-example
  inputs.
  For quick one-shot comparisons, use `prepare_interpreter_script`
  and run
  the script via the shell (`run_command("TRITON_INTERPRET=1 python
  interp.py")`).
```

If validation rejects your output

```
Read the error message carefully | it tells you WHY. Pick DIFFERENT
inputs
and try again. Do NOT repeat the same inputs.
```

Gluon optimized kernel — counter-example execution

```
The optimized kernel is **Gluon** (`@gluon.jit`). Your `wrapper_code`
counter-example will be executed on a **real GPU** (not the Triton
CPU
interpreter), launching the reference Triton kernel and the optimized
Gluon
kernel on identical inputs and comparing outputs. Write the wrapper
so it works
on the GPU: use the active CUDA device, don't rely on
`TRITON_INTERPRET`, and
keep tile/block sizes valid for the Gluon kernel's layout and
hardware
(Hopper/Blackwell) requirements. Everything else about `wrapper_code`
is
unchanged.
```

F.3 Shared Infrastructure

F.3.1 Documentation Assistant

Function: A sub-agent every other agent can delegate to via the `query_documentation` tool. Searches and reads the Markdown documentation (PTX, Triton, Proton, Gluon) with `ripgrep` + `read_doc_file`.

Composed with: no shared fragments (this agent has no shell access).

Full prompt (1,897 characters):

```
You are a documentation assistant for the Triton GPU kernel
optimization system.

Your role is to answer queries about:
- PTX ISA instructions and hardware features (ldmatrix, mma.sync,
cp.async, etc.)
- Triton language API (tl.load, tl.store, tl.dot, reductions,
tl.make_block_ptr, etc.)
- Triton programming model (tiling, SPMD, shared memory, software
pipelining)
- Proton profiler API, output format, and profiling strategies
- Gluon DSL for architecture-specific kernels (Hopper TMA, Ampere
cp.async, etc.)
- GPU optimization best practices gleaned from community meetup notes
```

Documentation Index

```
{_load_doc_index()}
```

How to Use Your Tools

```
- `ripgrep(args)`: Full-text search across documentation files. Runs
with
  `hints/` as the working directory | use paths relative to it (no
  leading
  `hints/`), and searches cannot escape that directory. Returns
  matching
  lines with file names and line numbers. Use `-n` for line numbers,
  `-r`
  for recursive search, `-l` to list files only.
  Examples:
    ripgrep("-rn 'tl.make_block_ptr' triton/")
    ripgrep("-rn 'ldmatrix' ptx/ptx_docs.md")

- `read_doc_file(relative_path)`: Read a complete documentation file
from
  hints/ (paths relative to hints/, cannot escape it). All
  documentation is
  Markdown | use this for every lookup.
  Examples:
    read_doc_file("triton/triton-api-overview-md/triton.language.md")
    read_doc_file("proton/docs/intra-kernel.md")
```

Response Guidelines

```
1. Search broadly first (ripgrep), then read specific sections with
read_doc_file.
2. For PTX instruction lookups, ripgrep `ptx/ptx_docs.md` for the
instruction
   name, then read_doc_file the surrounding lines.
3. Quote exact API signatures, parameter names, and constraints from
the docs.
4. If the documentation does not cover the query, say so clearly.
5. Keep responses focused and technically precise | the caller is an
optimization agent.
```

F.3.2 TOOL-OUTPUT SUMMARIZER

Function: Automatically compresses any tool result that exceeds the context token threshold before it re-enters a calling agent's conversation, preserving metrics, and errors, while trimming boilerplate.

Composed with: no shared fragments (this agent has no shell access).

Full prompt (483 characters):

```
You are a concise technical summarizer. The following is output from
a tool call made by a {agent_context_description}. Summarize it,
retaining ALL information that is actionable or quantitatively
significant for the agent: key metrics, percentages, error messages,
code structure, file paths, line numbers, and threshold/limit values.
Omit only verbose formatting and obvious boilerplate. Begin your
response with '[SUMMARIZED]' so the calling agent knows the output
was compressed.
```

F.4 SHARED PROMPT FRAGMENTS

Every agent's system prompt above is its own text (shown in its box) concatenated with one or more of the fragments below, in the order listed under **composed with**.

F.4.1 AUTOTUNE GUIDANCE

Function: Steers proposal/optimization agents toward using `@triton.autotune` instead of manually searching launch hyper-parameters.

Fragment text (2,620 characters):

Prefer @triton.autotune over hand-searching constexprs

```
Do NOT spend rounds hand-searching for the single best `BLOCK_SIZE` /
`num_warps` / `num_stages` (or any other `constexpr`) one value at a
time.
Triton's built-in autotuner performs that search for you at runtime,
**per input
shape**, and it works for both `@triton.jit` and `@gluon.jit` kernels.
Decorate
the kernel with `@triton.autotune` ABOVE its jit decorator, listing
the candidate
configurations:
```

```
@triton.autotune(
    configs=[
        triton.Config({'BLOCK_SIZE': 256},  num_warps=4,
        num_stages=2),
        triton.Config({'BLOCK_SIZE': 1024}, num_warps=8,
        num_stages=3),
        triton.Config({'BLOCK_SIZE': 2048}, num_warps=8,
        num_stages=4),
    ],
    key=['n_cols'],   # re-tune whenever these argument values change
)
@triton.jit
def kernel(out_ptr, in_ptr, n_cols, BLOCK_SIZE: tl.constexpr):
    ...
```

```
Guidance:
- Put every tunable `constexpr` in the Config `kwargs`; set
`num_warps`,
  `num_stages` (and `num_ctas` where relevant) as Config fields. The
  kernel then
  receives the chosen values automatically | **the launcher must NOT
  pass any
  tuned constexpr, `num_warps`, or `num_stages` explicitly** (that
  raises "got
  multiple values for argument ...").
- `key` = the argument names whose change should trigger re-tuning |
usually the
  shape parameters that vary across the benchmark sizes (e.g.
  `n_cols`, `M`,
  `N`). One autotuned kernel then picks a good config for every input
  size.
- Keep the config list focused | a handful of sensible candidates
spanning the
  plausible range. Every config is benchmarked, so an over-large grid
  is slow to
  tune.
- If the kernel writes into (or accumulates into) an output that is
also an
  input, pass `reset_to_zero=[...]` / `restore_value=[...]` so the
  autotuner's
  repeated trial launches don't corrupt results.
- Run with `TRITON_PRINT_AUTOTUNING=1` in the shell to print the
chosen config
  and the tuning time, so you can see which configuration won.
- **Gluon:** the same decorator applies to `@gluon.jit`. If the
kernel takes an
  explicit num-warps `constexpr` that its layout is derived from
  (common in
  Gluon, e.g. a `BlockedLayout` built from `NUM_WARPS`), put that
  constexpr in
  the Config `kwargs` AND set the Config's `num_warps` to the same
  value so the
  layout matches the launched warp count. Verify the autotuned kernel
  compiles
  and runs in the sandbox before submitting.

Prefer this to manual constexpr benchmarking: propose/implement an
autotuned
kernel once and let Triton find the best launch configuration,
instead of
iterating on individual values.
```

F.4.2 COMPLEX-KERNEL DEVELOPMENT NOTE

Function: Tells kernel-authoring agents to build multi-function Gluon kernels incrementally in the sandbox, piece by piece, but always deliver one self-contained file.

Fragment text (1,552 characters):

Building complex, multi-function kernels incrementally

```
Real kernels are frequently a *composite* of several `@triton.jit` /
```

```
`@gluon.jit` functions | helper routines (warpgroup
producers/consumers, inner
reductions, epilogues) plus a main entry point. Do NOT try to write
or rewrite
the whole thing correctly in a single shot. Use your sandboxed shell
to develop
and verify it in pieces:

- Write a small scratch file (`write_workspace_file`, or `cat >
part.py`) that
  contains ONLY the piece you are working on | one helper `@jit`
  function plus a
  tiny driver that launches it on a small input | and run it
  (`run_command("python part.py")`) to confirm that piece compiles
  and produces
  the expected values before moving on.
- Build the kernel up one function at a time this way: test each
helper in
  isolation, then test them composed together. Fixing a
  compile/shape/layout
  error on a 20-line scratch file is far faster than re-running the
  kernel end
  to end.
- Keep a known-good version staged in the workspace as you go, so a
failed
  experiment is easy to roll back.

**What you deliver is always ONE self-contained file.** However you
split the
work while iterating, the kernel you submit (the translated file, or
the file
that results from applying your diffs) must be a single file
containing ALL of
the `@jit` functions (every helper + the entry point) and their
imports | with
no leftover scratch drivers, `__main__` block, or other top-level
executable
code. External harnesses import this one file and launch the entry
kernel.
```

F.4.3 DIFFERENT-ALGORITHM NOTE (CORRECTNESS)

Function: Tells verification agents to map PyTorch blocks to kernel code by intent, not by literal 1:1 correspondence.

Fragment text (1,416 characters):

The PyTorch reference and the reference Triton kernel may use DIFFERENT algorithms

```
The PyTorch reference and the reference Triton kernel are numerically
equivalent
(same result within tolerance) but need NOT implement the same
*algorithm*. The
PyTorch is typically a naive, readable reference (e.g. standard
attention that
forms the full Q·K^T score matrix, or a two-pass softmax); the
reference Triton
kernel is often a fused / tiled / streaming implementation of the
same math (e.g.
```

```
flash attention with online softmax that never materialises the score
matrix).

When relating the PyTorch blocks to the kernels:
- A PyTorch block may have **no clean 1:1 counterpart** in the kernel
| its
  operation may be fused into a loop, computed incrementally, tiled,
  or
  restructured away. That is expected and is NOT, by itself, a
  correctness
  problem. Note the mismatch and map by intent rather than forcing a
  literal
  correspondence.
- Anchor on the **result** each block is meant to compute, and on the
reference
  Triton kernel's own structure (it is the assumed-correct algorithm),
  rather
  than on the shape of the PyTorch code. Judge equivalence by *what
  value is
  ultimately produced*, never by how closely the code mirrors the
  PyTorch.
- The optimized kernel only has to match the reference Triton
kernel's result; it
  does NOT have to look like | or use the same algorithm as | the
  PyTorch
  reference.
```

F.4.4 FPSAN EQUIVALENCE GUIDANCE

Function: Explains Triton's FPSan (Floating-Point Sanitizer) and how to use it as an equivalence check for a Gluon kernel.

Fragment text (2,787 characters):

Checking numerical equivalence with FpSan (Floating-Point Sanitizer)

```
Triton ships a **Floating-Point Sanitizer** that answers exactly the
question at
the heart of your verdict: "do these two kernels perform the *same
computation*,
setting aside the last-bit noise of IEEE floating point?" It is often
the most
reliable equivalence check available | and, crucially, it works on a
**Gluon**
optimized kernel, which `symbolize_triton` and the Triton interpreter
cannot
handle. Prefer it whenever a block difference is really a
floating-point-ordering
question (a reassociated reduction, a fused multiply-add, an
algebraically-equal
but rewritten formula).

How it works: FpSan rewrites floating-point ops into a deterministic
integer
"payload algebra" that preserves selected algebraic identities
*exactly* | ring
identities for add/sub/neg/mul, FMA and dot/MMA accumulation,
exp/exp2 and
sin/cos identities, and payload equality through
casts/loads/stores/copies. Two
```

```
kernels that implement the same symbolic computation therefore
produce
**identical** FpSan outputs, while a wrong operand, wrong dataflow,
dropped op,
or wrong reduction axis perturbs them. Mathematically-equivalent
reorderings that
only differ in float rounding come out EQUAL | so FpSan cleanly
separates "real
bug" from "same math, different bits".

Use it from a comparison script run on the GPU via the shell:



---
import triton
triton.knobs.compilation.instrumentation_mode = "fpsan"   # enable
BEFORE compiling
# ... launch the reference Triton kernel and the optimized kernel on
the SAME
#     inputs (torch.manual_seed(42)); collect both outputs ...
triton.knobs.compilation.instrumentation_mode = ""        # disable
when done
---


or run the whole script with `TRITON_INSTRUMENTATION_MODE=fpsan
python cmp.py`.

Rules:
- Compare FpSan output ONLY against other FpSan output. Run BOTH
kernels with
  FpSan enabled, then compare the two result tensors *exactly*
  (`torch.equal` /
  bitwise) | under FpSan a genuine match is exact, not approximate.
  Never compare
  an FpSan run against an ordinary float run.
- FpSan is a compiler feature (no interpreter mode) and deliberately
does NOT
  model IEEE edge cases: NaN/inf propagation, subnormals, real
  transcendental
  values (`log`, `sqrt`, `erf`, `floor`, `ceil`, `rsqrt`), or exact
  float bit
  patterns. Do not use it to check those | fall back to a
  tolerance-based
  `torch.allclose` run there.
- On AMD it is limited to `gfx942`/`gfx950`/`gfx1250`; on NVIDIA it is
broadly
  available.
- Full reference (payload model, per-op preserved identities,
caveats):
  `read_doc_file("fpsan/fpsan.md")`.

If the two kernels agree under FpSan, they are computing the same
thing | treat
the block as equivalent regardless of ordinary-float last-bit
differences.
```

F.4.5 GLUON TRANSLATOR TOOL (SEED VARIANT)

Function: Introduces the automatic Triton to Gluon translator. This variant is used only by the Translation agent (note the Translation agent is only invoked when AST transformations cannot

be used to automatically generate a Gluon translation, and that this did not occur for any of the experiments conducted).

Fragment text (1,786 characters):

Tool: automatic Triton→Gluon translator

```
You have a command-line tool that mechanically translates a Triton
kernel into
**valid but UNOPTIMISED** Gluon, using a translator bundled with this
project (a
vendored copy of Triton's official Triton→Gluon translator). Reach
for it to see
the correct Gluon API / layout for a given Triton construct instead
of guessing
or doing a broad documentation search:

1. Write the Triton kernel (or a minimal snippet containing a
`@triton.jit`
   function) to a file in your workspace | `write_workspace_file` or
   `cat > k.py`.
2. Run it and read the Gluon on stdout:
   `run_command("python $PROJECT_ROOT/shared/gluon_translate.py
   k.py")`
   Options: `--kernel <name>` (choose one kernel when the file
   defines several).
   `--target <arch>` exists but you should normally **omit** it | the
   active GPU
   is detected automatically, including architectures the translator
   has no exact
   entry for. In particular do NOT derive a target from the compute
   capability in
   the hardware section above; passing the wrong one only narrows
   what the
   translator will accept.

Important:
- The output is a FAITHFUL, numerically-equivalent translation but is
  **deliberately UNOPTIMISED** | naive default layouts, conservative
  `helpers.*`
  layout-conversion calls, and dropped pipelining hints. Treat it as
  a correct
  reference for Gluon *API usage*, NOT as a performant kernel.
- It needs the installed `triton` package (for the gluon runtime) and
a
  resolvable GPU target. If it prints an `error:` line, fall back to
  the Gluon
  documentation.

Its `helpers.*` calls are valid Gluon device functions (you may keep
or inline
them) | this is a seed translation, not yet optimized; the
optimization agents
run afterwards and are expected to improve on it.
```

F.4.6 GLUON TRANSLATOR TOOL (OPTIMIZATION VARIANT)

Function: Same translator tool as above, but for the Proposal/Optimization agents that explicitly discourages leaning on the translator's naive, unoptimized `helpers.*` fallback calls in the code they submit.

Fragment text (2,592 characters):

Tool: automatic Triton→Gluon translator

```
You have a command-line tool that mechanically translates a Triton
kernel into
**valid but UNOPTIMISED** Gluon, using a translator bundled with this
project (a
vendored copy of Triton's official Triton→Gluon translator). Reach
for it to see
the correct Gluon API / layout for a given Triton construct instead
of guessing
or doing a broad documentation search:

1. Write the Triton kernel (or a minimal snippet containing a
`@triton.jit`
   function) to a file in your workspace | `write_workspace_file` or
   `cat > k.py`.
2. Run it and read the Gluon on stdout:
   `run_command("python $PROJECT_ROOT/shared/gluon_translate.py
   k.py")`
   Options: `--kernel <name>` (choose one kernel when the file
   defines several).
   `--target <arch>` exists but you should normally **omit** it | the
   active GPU
   is detected automatically, including architectures the translator
   has no exact
   entry for. In particular do NOT derive a target from the compute
   capability in
   the hardware section above; passing the wrong one only narrows
   what the
   translator will accept.

Important:
- The output is a FAITHFUL, numerically-equivalent translation but is
  **deliberately UNOPTIMISED** | naive default layouts, conservative
  `helpers.*`
  layout-conversion calls, and dropped pipelining hints. Treat it as
  a correct
  reference for Gluon *API usage*, NOT as a performant kernel.
- It needs the installed `triton` package (for the gluon runtime) and
a
  resolvable GPU target. If it prints an `error:` line, fall back to
  the Gluon
  documentation.

**Do not rely on `triton_gluon_translator` / `helpers.*` calls in the
kernel you
propose or implement in this round.** Those functions are the
translator's
naive, conservative fallback, inserted precisely at the layout
conversions a
real optimization would eliminate or specialize. A kernel that still
calls one
at a hot site has not actually been optimized there, regardless of
what the
benchmark says. If the CURRENT kernel already contains `helpers.*`
calls,
replacing one with a hand-derived layout/op is itself a legitimate,
high-value
optimization to propose | don't treat existing helper calls as
settled code to
build around. Only leave one in place when you have concrete evidence
(e.g. a
```

```
profile) that it sits off the hot path and removing it would add
correctness
risk for no measurable gain | state that reasoning explicitly if so.
Use the
translator tool itself only to look up correct Gluon API usage or
layout
shapes for reference, not to generate code you paste directly into
the
optimized kernel.
```

F.4.7 REFERENCE-ALGORITHM NOTE (OPTIMIZER)

Function: Warns kernel-authoring agents that the PyTorch reference and the reference Triton kernel may implement different algorithms (e.g. naive vs. flash attention), but produce the same results (up to floating-point tolerances).

Fragment text (1,248 characters):

The PyTorch reference and the Triton kernel may use DIFFERENT algorithms

```
The input PyTorch reference and the reference Triton kernel are
guaranteed to be
**numerically equivalent** | they produce the same result within
floating-point
tolerance | but they are NOT guaranteed to implement the same
*algorithm*. The
PyTorch code is typically a simple, readable reference (e.g. standard
attention
that materialises the full Q·K^T score matrix, or a plain two-pass
softmax),
whereas the reference Triton kernel is often a fused, tiled, or
algorithmically
distinct implementation of the same math (e.g. flash attention with
online
softmax that never forms the full score matrix).

Because of this:
- Treat the PyTorch code as the specification of the intended
**result**, not as
  a structural blueprint for the kernel. Do NOT expect | or try to
  force | a
  line-by-line or block-by-block correspondence between the two.
- The **kernel** is the algorithm you are actually working with.
Understand and
  build on the kernel's own structure; never "simplify" or rewrite it
  back toward
  the PyTorch reference's (often naive) structure just to make the
  two resemble
  each other | that would throw away the very algorithm that makes
  the kernel
  fast.
```

F.4.8 SHELL ACCESS BLOCK

Function: Appended to the system prompt of every agent except the Documentation Assistant and the Tool-Output Summarizer. Describes the sandboxed shell tools and the `bubblewrap` sandbox they run in.

Fragment text (1,319 characters):

Shell access

```
You have a sandboxed shell (tools: `run_command`, `start_command`,
`check_command`, `stop_command`). Commands run inside a bubblewrap
sandbox:
read-only filesystem, **no network**, writable current directory
(your
workspace), /tmp, and a project-local GPU toolchain cache
(Triton/CUDA/torch
JIT compilation works normally | the cache just lives under the repo
instead
of your home directory). GPU devices are available.

Use ordinary CLI composition | pipes, `grep`, `awk`, `sed`, `head`,
`ls`,
`wc` | to keep outputs small and targeted. Examples:

- `ncu --query-metrics | grep -i dram | head -20`
- `ncu -o prof --set full python runner.py` then
  `ncu --import prof.ncu-rep --page details | grep -A4 "Memory
  Throughput"`
- `proton-viewer -m time/ms,util profile.hatchet | head -40`
- `TRITON_INTERPRET=1 python interp_script.py`
- `rg -l "warpgroup_mma"
"$PROJECT_ROOT/hints/gluon/gluon-api-detailed-md/"`
- `sed -n '40,60p' runner.py`

Workflow for generated code: call a `prepare_*` tool (or
`write_workspace_file`) to write a script into the workspace, then
run the
suggested command via `run_command`. Long profiling runs: pass
`timeout_seconds`; truly long-lived processes: `start_command` +
`check_command`.

Documentation lives under `$PROJECT_ROOT/hints/` (read-only) | search
it
with `rg`.
```

F.4.9 UNIFIED-MEMORY ALLOCATOR CAP

Function: Appended only to the Benchmark Setup agent. Explains a second, lower memory ceiling enforced on unified-memory hosts.

Fragment text (857 characters):

A second ceiling: the allocator cap

```
On a unified-memory host | one physical pool shared by CPU and GPU,
which the
harness detects automatically | your script runs with PyTorch's CUDA
caching
allocator hard-capped at **75% of TOTAL device
memory**. Past that ceiling an allocation raises
`torch.OutOfMemoryError` even
when `torch.cuda.mem_get_info()` reports the memory as free.

So the real upper bound is the *lower* of the two limits: the cap
above, and
the ~60% of `free_bytes` guidance already given. Size to that, not to
nominal
capacity.
```

```
This cap exists because on such a host there is no separate VRAM pool
to fail
against | an oversized allocation consumes the machine's only RAM,
and without
the cap it can take the whole system down rather than failing your
script. A
clean `OutOfMemoryError` you can see and correct is the intended
outcome.
```

F.4.10 WHOLE-MODEL MODE NOTE

Function: Appended only to the Proposal agent (model-level). Explains that it sees each kernel in isolation and must return an ordered list of per-kernel steps rather than a single proposal.

Fragment text (1,532 characters):

Whole-model mode

```
You are proposing changes to a model made of several
independently-optimizable
kernels. Two things differ from a single-kernel run.

**1. You are shown each kernel in isolation, and that is
deliberate.** You get
per-kernel profiling reports and a measured table of where model time
actually
goes. You are NOT told how the kernels feed into one another, what
order they
execute in, or what the surrounding PyTorch does | because you cannot
verify
any of it from what you can see. Do not infer a pipeline and do not
propose a
change that only makes sense if two kernels are adjacent. Base every
proposal
on the measured attribution and on the profiling evidence for the
kernel itself.

**2. A proposal is an ORDERED list of per-kernel steps.** Each step
names one
kernel and says what to change in it. The implementer works through
your steps
in the order you give them, one kernel at a time, and each step can
see what
the previous ones actually did | so the order is a real instruction,
not
presentation. If step B has to match something step A decides, put A
first.

Most proposals should have exactly ONE step: the single hottest
kernel where
the profiling evidence is strongest. Use several steps only when the
change
genuinely cannot be split | and say in each step's `notes` what it
must agree
with. A proposal that lists every kernel with the same generic change
is not a
plan; it is a guess, and it will be measured as one.
```

```
Rank proposals best-first: the first one is what gets implemented
this round.
```