TileBench: A Controlled Benchmark for Performance Evaluation and Bottleneck Diagnosis of Tile-Based Programming Models

Bowen Cui[1,*], Zhongchun Zhou[1,*], Hao Wu[1], Tejas Ramesh[1], Junyu Yin[1], Jialiang Gu[1], Keren Zhou[1]

[1]George Mason University

Correspondence: kzhou6@gmu.edu

Abstract

Tile-based programming models, such as Triton and cuTile, aim to simplify high-performance kernel development, but their practical performance, tuning behavior, and usability remain difficult to compare systematically. We present TileBench, a controlled benchmark for evaluating Triton and cuTile on NVIDIA B200 GPUs under matched operator semantics and comparable implementation structures. TileBench contains 45 operators covering diverse AI-kernel patterns and memory/computation behaviors. Each task provides a PyTorch reference, verified Triton and cuTile implementations, standardized data-types (dtype) and input-size sweeps, default and autotuned configurations, roofline-based metrics, and profiling-guided diagnosis. Our evaluation shows that performance gaps are workload-dependent: cuTile excels on a small cluster of Tensor-Core/TMA-friendly kernels, while Triton is stronger on many irregular, streaming, and bandwidth-bound operators. We further evaluate LLM-generated cuTile and Triton kernels and find that Triton is consistently more token-efficient than cuTile under the same iterative refinement protocol. TileBench is publicly available at `https://github.com/Deep-Learning-Profiling-Tools/Tilebench`.

1 Introduction

Modern AI workloads, such as Large Language Models (LLMs) (Brown et al., 2020), diffusion models (Ho et al., 2020), and multi-modal models (Radford et al., 2021), increasingly depend on custom kernels beyond standard library primitives, including attention variants (Dao, 2024), normalization (Zhang and Sennrich, 2019), Mixture-of-Experts (MOE) routing (Jiang et al., 2024), quantization and dequantization (Xiao et al., 2024), and fused point-wise operations (Lin et al., 2026), which involve diverse computation patterns. Although highly optimized implementations for common primitives are available in vendor libraries such as cuBLAS (NVIDIA, 2024a) and cuDNN (Chetlur et al., 2014), template libraries such as CUTLASS (NVIDIA, 2024b), and specialized kernels such as FlashAttention (Dao, 2024), many emerging operators still require custom kernel development involving custom fusion patterns, memory layouts, and advanced algorithms to fully exploit modern GPU hardware (Bentz, 2025; Lin et al., 2026).

Implementing high-performance GPU kernels remains difficult. CUDA (Nickolls et al., 2008) provides fine-grained control over the memory hierarchy, synchronization, and Tensor Cores, but this control comes with substantial engineering costs (Liao et al., 2025). High-level tile-based programming models such as cuTile (Bentz, 2025) and Triton (Tillet et al., 2019) aim to reduce this burden while offering efficient performance. Unlike the traditional Single Instruction Multiple Threads (SIMT) programming model (Lindholm et al., 2008), these domain-specific languages (DSLs) enable developers to program with a higher abstraction: they load and store chunks of data (Tiles) and apply certain computations on these tiles through tile-level primitives, facilitating the adoption of Tensor Cores (i.e., `mma`), Tensor Memory Accelerator (i.e., `cp.async.bulk.tensor`), and Tensor Memory instructions (i.e., `tcgen05`). The design philosophy of these DSLs enables developers to "focus on your algorithm" while leaving complex code optimizations to the compiler and runtime.

Though such abstractions hide low-level complexity, it is still difficult for developers to diagnose performance bottlenecks and optimize the kernels for diverse AI operators. Recent cuTile evaluation studies (Yadav et al., 2026) compare cuTile with cuBLAS, Triton (Tillet et al., 2019), Warp Matrix Multiply and Accumulate

[*]These authors contributed equally to this work.

Findings of the Association for Computational Linguistics: IJCNLP-AACL 2026, pages 2145–2167
November 6-10, 2026 ©2026 Association for Computational Linguistics

(WMMA), and raw SIMT on general matrix multiplication (GEMM) and attention across Hopper and Blackwell GPUs, showing that cuTile can be highly effective on selected B200 workloads but is also strongly workload- and architecture-dependent. On the other hand, Triton can yield different performance with regard to different versions of the compiler, autotuning space, Machine Learning (ML) algorithm, or implementation details when integrated into systems and libraries such as vLLM (Kwon et al., 2023), LightLLM (Hu et al., 2024), Liger-Kernel (Hsu et al., 2024), and Unsloth (Daniel Han and team, 2023). As a result, the community lacks a systematic understanding of performance bottlenecks, best practices for coding with these DSLs, and comparisons among different tile-centric DSLs.

To support fair evaluation and future tooling for tile-centric DSLs, we present **TileBench**, a controlled benchmark for comparing Triton and cuTile implementations of AI operators. TileBench contains 45 operators spanning attention and sequence primitives, GEMM variants, normalization, reductions, loss functions, elementwise and fused pointwise kernels, convolution and stencil computations, data movement, indexing, quantization, and sorting. Each task includes a PyTorch reference implementation for correctness checking and baseline measurement, together with matched Triton and cuTile implementations.

Our work makes the following contributions:

- We introduce TileBench, a controlled benchmark of 45 GPU operators on an NVIDIA B200 GPU, with manually written and verified Triton/cuTile implementations under matched semantics, comparable implementation structures, and standardized multi-dtype and input-size sweeps.
- We evaluate both default and autotuned configurations, quantifying how much performance is gained from systematic autotuning and how much performance headroom remains relative to hardware roofline limits.
- We build a profiling-guided analysis workflow to explain performance differences between Triton and cuTile using generated code, compiler behavior, and hardware-level metrics.
- We evaluate Triton and cuTile as target programming models for LLM-generated kernels, measuring not only correctness and runtime performance but also the token consumption required to achieve performant implementations.

Across the suite, both programming languages substantially outperform PyTorch, while Triton achieves stronger overall performance, and cuTile is competitive on a small set of Tensor Core and TMA-friendly workloads. Autotuning improves both backends but leaves substantial roofline headroom, suggesting that compiler lowering and memory behavior often dominate tile-parameter choices. Evaluation on LLM-generated cuTile and Triton kernels demonstrates that Triton is consistently more token-efficient than cuTile under the same iterative refinement protocol.

2 Related Work

2.1 Tile-Based GPU Programming Models

Triton (Tillet et al., 2019) and cuTile (Bentz, 2025) are two representative tile-centric interfaces that are directly relevant to NVIDIA GPUs. Triton exposes a Python-embedded block program in which users specify tile sizes, pointer arithmetic, and scheduling hints, while the compiler lowers tile-level tensor operations to GPU code. On NVIDIA GPUs with Tensor Memory Accelerator (TMA) support, Triton's tensor descriptor API can lower descriptor-based loads and stores to TMA-backed memory (Triton Document). cuTile raises a similar abstraction inside the CUDA ecosystem: kernels are expressed through tile operations such as `ct.load`, `ct.mma`, and `ct.store`, and the compiler and runtime map these operations to threads, memory movement, Tensor Cores, and TMA.

The broader ecosystem also includes several production and research systems, such as TileLang (Wang et al., 2025), ThunderKittens (Spector et al., 2025), Tilus (Ding et al., 2026), TLX (Guan et al., 2026), and AWS NKI (AWS, 2026). However, these systems are not all directly comparable in our setting: NKI targets non-GPU accelerators, TLX extends Triton rather than serving as an independent high-level backend, and TileLang, ThunderKittens, and Tilus expose low-level parallelism, scheduling, layouts, and runtime controls. TileBench, therefore, focuses on Triton and cuTile: both can be evaluated on the same NVIDIA B200 platform, both expose tile-centric programming interfaces, and both support a meaningful apples-to-apples comparison under matched semantics,

comparable tuning budgets, roofline analysis, and profiling-guided diagnosis.

2.2 Benchmarks for GPU Kernel Generation and Optimization

Recent benchmarks evaluate LLM-written or automatically optimized GPU kernels from different stages. KernelBench (Ouyang et al., 2025) and CUDABench (Zhu et al., 2026) focus on text-to-CUDA generation; BackendBench (Saroufim et al., 2025) and TritonBench (Li et al., 2025) study PyTorch backend or Triton operator generation; FlashInfer-Bench (Xing et al., 2026) emphasizes serving-oriented inference primitives; ComputeEval (NVIDIA, 2025) covers broader CUDA programming ability; and SOL-ExecBench (Lin et al., 2026) introduces hardware speedup-of-light evaluation. These benchmarks measure correctness, software-baseline speedup, GPU efficiency, or hardware headroom, but they do not isolate programming-model effects between tile-level DSLs. TileBench is complementary; its main track uses manually written and verified Triton/cuTile implementations under matched operator semantics and comparable autotuning ranges, while Section 3.3 separately measures how easy and token-efficient it is for LLMs to generate correct and performant kernels for each DSL.

3 TileBench

TileBench is a controlled benchmark for comparing Triton and cuTile under matched operator semantics and standardized evaluation. PyTorch serves as the semantic reference, correctness oracle, and software baseline, while Triton and cuTile are the primary programming models under comparison.

3.1 Overview and Pipeline

Figure 1 shows the TileBench workflow. The pipeline has three stages. First, the benchmark construction stage curates operators from TritonBench (Li et al., 2025) and LeetGPU (LeetGPU, 2025) and normalizes them into a unified TileBench task format. Second, the benchmark-execution stage evaluates each task through a shared harness: PyTorch produces reference outputs and baseline latency, while Triton and cuTile implementations are compiled, run through the configured default or autotuned path, verified, and timed on the same input cases. Third, the comparative analysis stage aggregates latency, speedup, throughput, effective bandwidth, and roofline utilization. For selected large-gap cases, we further use Nsight Compute (NVIDIA) together with PTX/SASS (NVIDIA, 2026) inspection to analyze instruction mix, memory behavior, TMA/`tcgen05` usage, and bank conflicts.

3.2 Benchmark Suite

Benchmark Sources and Coverage. TileBench contains 45 representative AI operator tasks derived from existing benchmark suites, including 26 from TRITONBENCH (Li et al., 2025) and 19 from LEETGPU (LeetGPU, 2025). We use these benchmarks as sources of operator semantics and task definitions, but all benchmark implementations are manually written and verified within the TileBench framework. For each operator, we provide controlled benchmark artifacts including `impl_torch.py`, `impl_triton.py`, `impl_cutile.py`, and `config.yaml`. This design ensures matched code shapes, algorithms, uniform data types (dtype) and input shapes, shared timing protocols, and consistent autotuning configurations across implementations. The main benchmark track evaluates manually implemented and verified Triton and cuTile kernels under the TileBench harness, while LLM-generated implementations are evaluated separately in Section 3.3.

The operators are grouped into five categories: *Point-wise*, *Reduction/Normalization*, *Matrix Multiplication/Attention*, *Stencil/Convolution*, and *Data Layout*. The category distribution is shown in Table 3. The full per-operator list is reported in Appendix A.1.

Dtype support and input-size sweeps are operator-specific. Each operator declares its supported dtypes and swept parameters in `config.yaml` (Appendix A.3), and TileBench evaluates 20 input configurations per supported dtype. This provides each task with standardized multi-dtype and multi-size coverage, designed to capture small-to-large scaling, B200 roofline behavior, and LLM-like regimes rather than inheriting the fixed cases from TritonBench or LeetGPU.

Standardization across Benchmarks. Standardization is necessary because the source benchmarks use different evaluation protocols: TritonBench provides Triton-focused operator-generation tasks and performance scripts, while LeetGPU follows a challenge-style format with per-problem functional tests and a single large performance case.

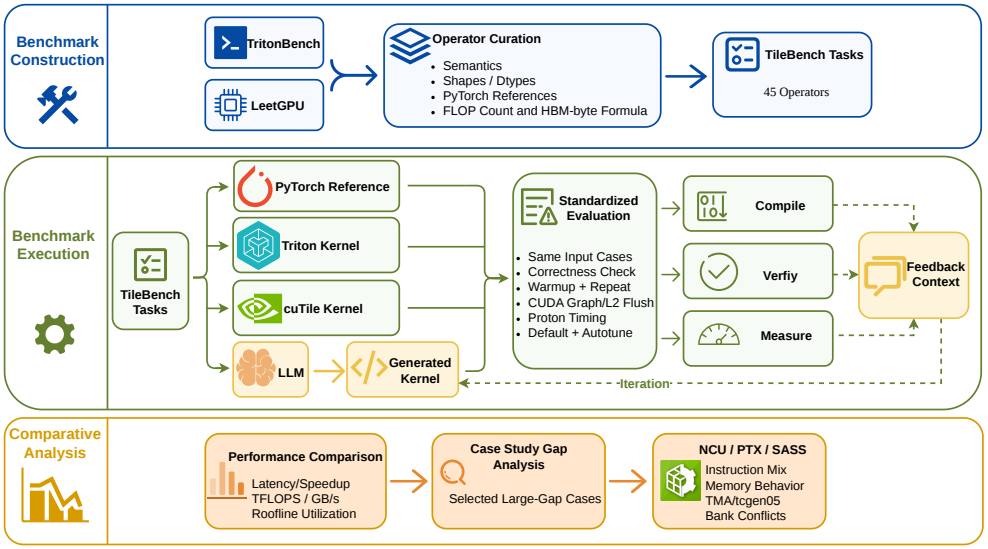


Figure 1: TileBench Overview.

TileBench reconstructs these operators into a unified benchmark with matched Triton/cuTile implementations, shared correctness checks, standardized multi-dtype and input-size sweeps, common timing controls, and comparable autotuning.

First, every operator has a PyTorch reference implementation that defines the semantic ground truth. Triton and cuTile outputs are compared against this reference using a shared correctness checker: floating-point outputs use `torch.testing.assert_close` with dtype- and operator-specific tolerances, while integer outputs use exact comparison (`atol = rtol =0`).

Second, each operator has a `config.yaml` file (Appendix A.3) that declares the dtype list, case grid, verification tolerances, timing options, and analytical FLOP/HBM-byte formulas. The same formulas are evaluated for every `(case, dtype)` pair to derive throughput, effective bandwidth, arithmetic intensity, and roofline utilization consistently across backends.

Third, all implementations are measured by the same timing protocol and report the mean kernel latency. Measurement uses the Proton profiler (Zhou et al., 2026) with CUDA graph capture-and-replay, L2-cache flushing, 20 warmup runs, and 100 timed runs; NVIDIA Nsight Compute (NCU) is reserved for selected post-hoc case studies.

Fourth, in the main benchmark track, the Triton and cuTile implementations provide both a default execution path and an autotuned execution path. The default path uses a single fixed, manually selected configuration, while the autotuned path searches over a manually defined set of candidate configurations and records the winning one for analysis. Since Triton and cuTile expose different tuning knobs, TileBench uses comparable tuning ranges rather than a one-to-one parameter mapping. This design allows each backend to use its native tuning interface while keeping the autotuning budget and search granularity comparable across programming models.

3.3 Iterative LLM Code-Generation

TileBench further defines an iterative LLM code-generation track over the same operator set. For each operator, the LLM generates Triton and cuTile implementations from the natural-language operator description and TileBench task specification; it does not see or optimize the human-written implementations used in the main benchmark track. Unlike the main benchmark track, LLM-generated code is evaluated without backend autotuning, and each iteration must commit to a single configuration. Each backend is evaluated for up to 10 refinement iterations on one representative large case per supported dtype, with verification and performance feedback used to guide the next implementation and configuration. A backend is frozen once its implementation passes verification and reaches at least 80% roofline utilization; otherwise, refinement continues until the 10-iteration budget is exhausted. The formal definition of the roofline utilization used for this stopping criterion is provided in Appendix C.1.

At iteration 0, the prompt (Appendix B.1) is

assembled from the TileBench framework guide (Appendix B.3), the Triton and cuTile API references (Appendix B.4), the natural-language operator description (Appendix B.5), the operator `config.yaml` (Appendix A.3), the PyTorch reference implementation (Appendix B.6), and the strict output-format instructions (Appendix B.7). The framework guide defines the implementation contract, multi-dtype handling, and hardware constraints. For iteration $N > 0$, the prompt (Appendix B.2) is rebuilt with the previous trajectory, per-backend configuration, verification status, roofline utilization, speedup over PyTorch, worst-first performance summary, previous generated source files, and, when the latest attempt regresses or fails verification, the best verify-clean source files as a rebuild reference.

The evaluator rejects three classes of reward hacking before timing: delegating computation to PyTorch, cuBLAS, cuDNN, or the PyTorch reference implementation; caching outputs using tensor identity, data pointer, or version-dependent keys; and bypassing the intended search process by importing backend autotuners. These cases are checked using static scans, two-pass anti-cache verification, and rotating-input timing. Any detected failures are returned to the LLM as structured feedback in the next iteration.

Triton and cuTile are tracked independently during refinement. The final result for each backend is selected as the best verify-clean iteration among all iterations executed before freezing or budget exhaustion.

3.4 Evaluation Metrics

Performance Metrics. For each operator o, backend b, dtype d, and input case c, the benchmark records mean kernel latency $T_{o,b,d,c}$. We report speedup over PyTorch as $T_{o,\mathrm{torch},d,c}/T_{o,b,d,c}$, along with achieved throughput, effective bandwidth, arithmetic intensity, and roofline utilization. Let $F_{o,d,c}$ and $B_{o,d,c}$ be the analytical FLOP count and HBM-byte count declared in `config.yaml`. Achieved throughput is computed as

$$\mathrm{TFLOPS}_{o,b,d,c} = \frac{F_{o,d,c}}{T_{o,b,d,c}} \cdot 10^{-12}.$$

Arithmetic intensity is $AI_{o,d,c} = F_{o,d,c}/B_{o,d,c}$. Roofline utilization normalizes achieved throughput by the dtype-specific roofline bound:

$$R_{o,b,d,c} = \frac{\mathrm{TFLOPS}_{o,b,d,c}}{\min(P_d^{\mathrm{peak}}, AI_{o,d,c} \cdot BW^{\mathrm{peak}})}.$$

Here P_d^{peak} denotes the B200 peak throughput for dtype d, and BW^{peak} denotes the B200 peak HBM bandwidth; both constants are specified in `B200.json` (Appendix A.4).

Token-aware Metrics. For the iterative LLM code-generation track, let o denote an operator, $b \in \{\mathrm{Triton}, \mathrm{cuTile}\}$ a backend, $i \in \{0, \ldots, 9\}$ a refinement iteration, and $\mathcal{D}_o$ the supported dtypes evaluated in this track. For each (o, b, i), it records the backend-level token usage $C^{\mathrm{tok}}_{o,b,i}$, counting failed iterations because compilation failures and incorrect generations are part of the search process.

We use correctness-gated speedup to measure useful performance. Let $\mathrm{correct}_{o,b,i}$ indicate that backend b at iteration i compiles and passes correctness checks for all evaluated dtypes. For dtype d, let $T_{o,\mathrm{torch},d}$ be the PyTorch reference latency and $T_{o,b,i,d}$ be the generated implementation latency on the representative large case. The correctness-gated speedup of iteration i is

$$G_{o,b,i} = \begin{cases} \dfrac{1}{|\mathcal{D}_o|} \displaystyle\sum_{d \in \mathcal{D}_o} \frac{T_{o,\mathrm{torch},d}}{T_{o,b,i,d}}, & \text{if } \mathrm{correct}_{o,b,i}, \\ 0, & \text{otherwise.} \end{cases}$$

Thus, incorrect or non-compiling iterations contribute zero speedup while still incurring token costs. We report:

$$\mathrm{BestSpeedup@10}_{o,b} = \max_{0 \le i < 10} G_{o,b,i}.$$

The token cost is accumulated over the $10-$iteration budget:

$$\mathrm{TokenCost@10}_{o,b} = \sum_{i=0}^{9} C^{\mathrm{tok}}_{o,b,i}.$$

Token efficiency is the best correctness-gated speedup per million tokens:

$$\mathrm{TokenEfficiency@10}_{o,b} = \frac{\mathrm{BestSpeedup@10}_{o,b}}{\mathrm{TokenCost@10}_{o,b}/10^6}.$$

When reporting aggregate results, we macro-average $\mathrm{BestSpeedup@10}$, $\mathrm{TokenCost@10}$, and $\mathrm{TokenEfficiency@10}$ across operators for each backend.

4 Experimental Setup

We run two evaluations on TileBench: a main benchmark track comparing the human-written Triton and cuTile implementations of all 45 operators, and the iterative LLM code-generation track described in Section 3.3.

Hardware. All measurements are taken on a single NVIDIA B200 GPU with 180 GB of HBM3e memory; the measured peak bandwidth used for roofline analysis is 6539.4 GB/s.

Software and Sampling. We use PyTorch 2.10 with CUDA 13.0, Triton 3.6.0, and the `cuda-tile` 1.3.0 release. Per-operator FLOP count and HBM-byte counts are evaluated from the `flops_expr` and `bytes_expr` declared in each `config.yaml`, ensuring throughput, bandwidth, and roofline utilization are derived consistently across backends. For aggregated metrics, unless stated otherwise, we use geometric mean over cases within each operator and over operators.

Models. For the iterative LLM track we evaluate two reasoning models: OpenAI GPT-5.5 (OpenAI, 2026) and Anthropic Claude Opus 4.7 (Anthropic, 2026), accessed through their respective APIs with default decoding settings.

5 Evaluations

- **RQ1: Autotuned Programming-Model Performance.** Under matched operator semantics and comparable implementation structures, how do the autotuned Triton and cuTile implementations compare with each other and with PyTorch baselines?
- **RQ2: Autotuning Sensitivity and Performance Headroom.** How much performance do the exposed configuration knobs recover beyond hand-selected defaults, and how much hardware headroom remains after tuning?
- **RQ3: Performance Gap Diagnosis.** Across TileBench, what compiler lowering and hardware execution behaviors explain the remaining gaps between aligned Triton and cuTile implementations?
- **RQ4: LLM Usability and Token Efficiency.** Under a matched generation and feedback protocol with a maximum budget of 10 refinement iterations and no backend autotuning, how do Triton and cuTile compare in correctness-gated speedup, token cost, and token efficiency for LLM-generated kernels?

5.1 reports the autotuned performance over the complete dtype and input-size sweeps. 5.2 directly compares the default and autotuned results over the same sweep. 5.3 profiles the NCU-cataloged

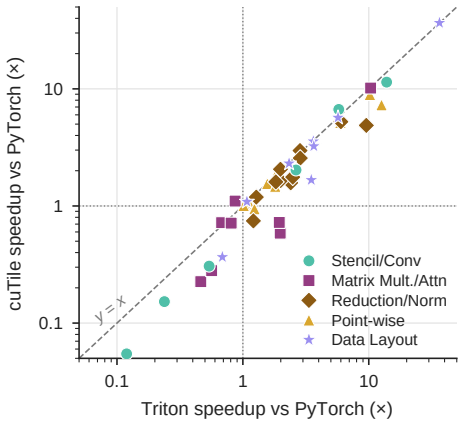


Figure 2: Triton and cuTile speedup over PyTorch across operators.

maximum input of each supported dtype using the autotuned winner selected for that exact case. 5.4 is evaluated separately: each model-backend pair is allowed up to ten refinement iterations and cannot invoke backend autotuners; once a backend satisfies the stopping criterion, it is frozen and incurs no additional token cost. Unless otherwise stated, all aggregation above the per-case latency uses geometric means.

5.1 RQ1: Autotuned Programming-Model Performance

Answer: Under autotuned configurations, Triton is stronger in aggregate and more robust across diverse workloads, while cuTile remains competitive and achieves its clearest advantages on regular, reuse-heavy tiled workloads, including Tensor-Core / TMA kernels.

Figure 2 compares the operator-level autotuned speedups of Triton and cuTile over PyTorch. Each point summarizes one operator using the geometric mean over all of its supported dtype and input-size cases. We make three observations:

(1) Both DSLs outperform PyTorch on most operators, but Triton is stronger overall. Triton achieves a geometric-mean speedup of $2.02\times$ over PyTorch, compared with $1.58\times$ for cuTile. Triton is faster than PyTorch on $36/45$ operators, while cuTile is faster on $33/45$. Their median speedups are $1.95\times$ and $1.58\times$, respectively, showing that the aggregate difference is not caused only by a few outliers.

(2) The direct Triton and cuTile comparison is workload-dependent. Triton is faster on $37/45$ operators, while cuTile is faster on $8/45$. However, the operator-level geometric-mean latency ratio be-

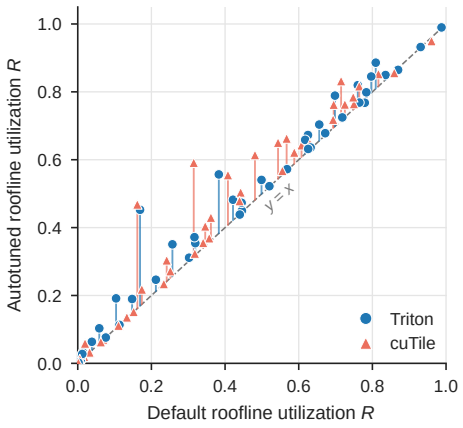


Figure 3: Default-to-autotuned roofline utilization across operators.

tween the two backends differs from parity by at most 5% on 14/45 operators and by at most 10% on 17/45, indicating that many head-to-head differences are small. The clearest cuTile advantages occur when regular tile movement, larger Tensor Core tiles, or two-dimensional data reuse can amortize tile-staging costs. Representative cases include `matmul_fp32_fp16_fp8`, `flash_attention`, and `jacobi_stencil_2d`. In contrast, Triton's largest margins occur on runtime-computed indexing, dynamic scatter, fine-grained reductions, sparse traversal, and low-parallelism loops, as observed in `destindex`, `weight_dequant`, `moe_topk_gating`, `flash_decode`, and the convolution family.

(3) Speedup over PyTorch does not always isolate a compiler-backend effect. PyTorch serves as a practical software baseline, but its internal execution path varies across operators. For eager compositions such as `rope`, `dropout`, `kl_divergence`, `fused_activation`, the DSL advantage mainly comes from kernel fusion and avoiding intermediate tensors. For reductions and indexing operators, such as `mean_reduction`, `argmax`, `destindex`, the difference reflects a shape-specialized DSL kernel compared with general ATen implementation. Conversely, PyTorch can be faster when it dispatches to highly optimized vendor implementation, including cuDNN convolution, cuBLAS GEMM, CUB radix sort, and warp-specialized attention kernels.

Additional distributions and category-level results are reported in Appendix D.1.

5.2 RQ2: Autotuning Sensitivity and Performance Headroom

Answer: Autotuning provides meaningful additional gains beyond hand-selected defaults for both backends, with a larger geometric-mean gain for cuTile, but substantial hardware headroom remains outside the exposed configuration spaces.

Figure 3 compares each operator's geometric-mean default and autotuned roofline utilization. The geometric-mean improvement is $1.18\times$ for Triton and $1.22\times$ for cuTile, while the median improvements are $1.07\times$ and $1.06\times$, respectively. These gains are measured relative to manually selected defaults rather than random or deliberately weak configurations. Autotuning therefore recovers meaningful performance from tile shape, thread organization, pipeline depth, and occupancy choices that are difficult to select uniformly across operators and dtypes.

The autotuned results should be interpreted as the best observed configurations within the declared search spaces. Triton exposes block dimension, warp counts, pipeline stages, and operator-specific scheduling parameters, while cuTile exposes tile dimensions, occupancy hints, and operator-specific launch choices. The search does not alter the source-level algorithm or explicitly explore alternative implementation structures, primitive choices, or lowering strategies. RQ2 therefore measures *configuration headroom*, not a global upper bound on either programming model.

After tuning, 7/45 Triton implementations and 5/45 cuTile implementations reach at least 80% operator-level geometric-mean roofline utilization. Only 2/45 Triton implementations and 1/45 cuTile implementation reach at least 90%. Thus, autotuning closes part of the performance gap while also revealing substantial remaining *backend headroom*. These factors generally cannot be corrected by selecting another tile size from the same implementation.

5.3 RQ3: Performance-Gap Diagnosis

Answer: The performance gaps between cuTile and Triton arise from recurring differences in index lowering, reduction granularity, state placement, and pipeline scheduling. cuTile's tensor core backend is not inherently weaker: it is highly effective on regular GEMMs, while its largest deficits occur when irregular indexing

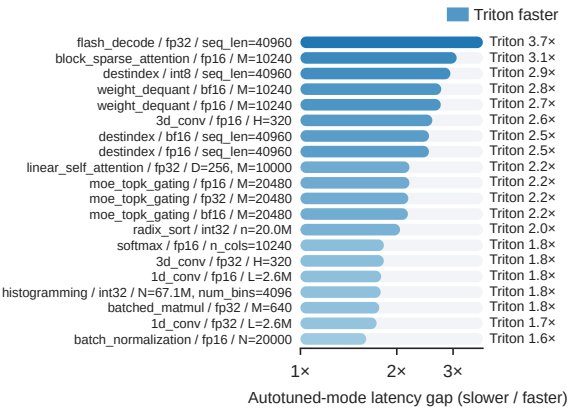


Figure 4: The 20 largest Triton-over-cuTile latency gaps under autotuned mode, evaluated at the NCU-profiled sweep-max input for each operator–dtype pair. Each bar represents one operator–dtype pair. The x-axis reports the ratio of the slower-backend latency to the faster-backend latency.

or fixed tile-lowering costs cannot be sufficiently amortized.

Aligned Triton versus cuTile. Figure 4 shows that all 20 of the largest sweep-max gaps favor Triton. The largest are `flash_decode`/FP32 at 3.72×, `block_sparse_attention`/FP16 at 3.08×, and `destindex`/INT8 at 2.94×. The largest cuTile advantage, 1.55× on FP32 `matmul_fp32_fp16_fp8`, falls below the twentieth Triton-faster gap and is analyzed separately below. Overall, the largest Triton advantages are concentrated in irregular indexing, dynamic scatter, small reductions, sparse or runtime-controlled loops, and cases where fixed tile-management costs are large relative to useful work. The following two cases expose the contrast between irregular indexing and regular tiled computation.

Case A: index lowering in `weight_dequant` and `destindex`. To investigate compiler-level performance discrepancies, we profile two memory-bound kernels—*BF16 Weight Dequantization* (`weight_dequant`) and *Int8 Destination Indexing* (`destindex`)—on B200. Despite their memory-bound nature, Triton substantially outperforms cuTile on both kernels. NCU profiling reveals an artificial bottleneck inversion in cuTile: in BF16 `weight_dequant`, cuTile saturates the Streaming Multiprocessor (SM) (87.8%) and Arithmetic Logic Unit (ALU) (89.7%) pipelines while Load/Store Unit (LSU) utilization drops to 12.8%, whereas Triton maintains a balanced profile (28.2% SM, 30.2% ALU, and 23.1% LSU). A similar pattern occurs in Int8 `destindex`, where cuTile sustains 78.5% ALU utilization versus 12.7% in Triton. While Triton warps predominantly encounter `stall_long_scoreboard` (hiding memory latency), cuTile warps are severely congested by `stall_not_selected` and `stall_math_pipe_throttle`, indicating that issue slots are choked by excessive integer arithmetic rather than memory stalls.

This compute saturation originates from severe instruction bloat during offset calculations rather than memory operations, and cuTile executes 6.2× more total instructions in BF16 `weight_dequant` and 17.3× more in Int8 `destindex` (86.99M vs. 5.04M). SASS disassembly traces this to multi-dimensional indexing (e.g., `offsets // N` and `offsets % N`): Triton lowers integer division/modulo via `arith.divsi/remsi` into constant reciprocal multiplications with magic constants (e.g., `0x66666667`). Conversely, cuTile lowers index arithmetic into defensive "safe-division" routines spanning much more SASS instructions (`ISETP`, `LOP3`, and `SEL`) to guard against signed inputs. Because tensor shapes are strictly non-negative compile-time constants (`tl.constexpr`), cuTile's index lowering leaves these value ranges unexploited, introducing entirely redundant arithmetic overhead.

Additionally, cuTile exhibits degraded memory coalescing in the `destindex` kernel, achieving only 1.78 sectors per load request (`sectors_sum/requests_sum`) compared to Triton's 8.50. This disparity shows that Triton vectorizes contiguous thread accesses into wide vectorized transactions, whereas cuTile fragments loads into disjoint, uncoalesced requests. In summary, the compounding effects of defensive arithmetic instruction bloat and memory transaction fragmentation fully account for cuTile's performance degradation.

Case B: regular tiled computation in `matmul_fp32_fp16_fp8`. Matrix multiplication provides a control case because the irregular indexing chain disappears. Both implementations use grouped tiled GEMM, FP32 accumulation, Tensor-Core MMA, and TMA-backed memory movement. The performance difference therefore cannot be explained by cuTile using TMA while Triton does not.

NCU instead shows that cuTile keeps the Ten-

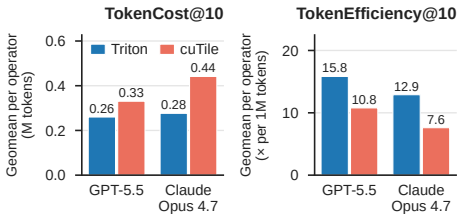


Figure 5: LLM token cost and efficiency by backend.

sor pipeline active for approximately 79%–81% of cycles, compared with roughly 52%–69% for Triton. At $M = N = 4096$ and $K = 20480$, cuTile is $1.55\times$ faster in FP32, $1.12\times$ faster in FP16, and $1.22\times$ faster in FP8. The FP32 difference is largest because cuTile sustains a $256 \times 256 \times 64$ tile, while Triton's TMA store staging and multi-stage buffers constrain its selected configuration to $128 \times 128 \times 32$. The smaller Triton tile nearly doubles the measured TMA load volume, 21.5 GB compared with 10.7 GB, while leaving scheduling and data-movement overhead amortized over less MMA work.

5.4 RQ4: LLM Usability and Token Efficiency

Answer: Under the same maximum 10-iteration protocol, Triton is a more effective and token-efficient target for both evaluated LLMs. It produces verify-clean kernels for more operators, reaches higher correctness-gated speedup, and consumes fewer tokens than cuTile.

Figure 5 reports the geometric-mean TokenCost@10 and TokenEfficiency@10 for each model-backend combination. We additionally report the geometric-mean BestSpeedup@10 over operators with at least one verify-clean result to separate final performance from token expenditure.

For GPT-5.5, Triton produces a faster-than-PyTorch verify-clean kernel on $38/45$ operators. cuTile does so on $36/43$ operators with at least one verify-clean result, with two operators failing to produce a verify-clean cuTile implementation. The geometric-mean BestSpeedup@10 is $4.12\times$ for Triton and $3.46\times$ for cuTile. The corresponding token costs are 0.260M and 0.331M tokens, while token efficiency is $15.83\times$ and $10.79\times$ per million tokens, respectively.

For Claude Opus 4.7, Triton produces a faster-than-PyTorch verify-clean kernel on $38/45$ operators. cuTile does so on $33/41$ operators with at least one verify-clean result, with four operators failing to produce a verify-clean cuTile implementation. The geometric-mean BestSpeedup@10 is $3.57\times$ for Triton and $3.16\times$ for cuTile. The token cost increases from 0.277M tokens on Triton to 0.441M on cuTile, while token efficiency decreases from $12.90\times$ to $7.61\times$ per million tokens.

The backend effect is consistent across both models. Relative to cuTile, Triton improves token efficiency by approximately 47% for GPT-5.5 and 70% for Claude Opus 4.7. The difference comes from both sides of the metric: Triton more often reaches a stronger verify-clean implementation, and the refinement process consumes fewer tokens while doing so. The model effect is less uniform. GPT-5.5 achieves the strongest Triton result, while the differences between the two models are smaller than the corresponding Triton and cuTile gaps.

These results establish an empirical usability difference under the TileBench protocol, but they do not isolate its cause. Possible explanations include differences in API expressibility, available primitives, compiler constraints, documentation, examples, and model exposure to each DSL. Because the experiment does not separately control these factors, we treat reduced model familiarity with cuTile as a possible explanation rather than a demonstrated causal mechanism.

6 Conclusions

We introduce TileBench, a controlled and reproducible benchmark for comparing tile-based GPU programming models and studying tile-centric DSLs. Our results show that both Triton and cuTile substantially outperform PyTorch, but neither abstraction is uniformly superior: Triton is more robust on irregular, streaming, and bandwidth-bound operators, while cuTile excels when workloads match static tile abstractions, Tensor Core computation, and reusable tile movement. Autotuning yields measurable gains, yet most kernels remain far from roofline limits, suggesting that backend lowering, memory staging, instruction selection, and shared-memory behavior matter more than tile-size search alone. The LLM-generated track further shows that programming-model design affects both final performance and the token cost of reaching correct, efficient kernels.

Limitations

TileBench currently targets a single NVIDIA B200 GPU and two tile-based DSLs. Its roofline metrics

use analytical FLOP/HBM-byte formulas, so they provide a standardized hardware-aware summary rather than a replacement for full NCU-counter analysis.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Award Nos. 2549919 and 2554012, and a donation from NVIDIA. We thank the anonymous reviewers for their valuable feedback. We acknowledge the use of Claude Code and OpenAI Codex for language editing and code implementation. No AI tools were used to generate research ideas, analyses, or experimental results. All code and text were reviewed, tested, and validated by the authors, who take full responsibility for the contents of this work.

References

Anthropic. 2026. What's new in Claude Opus 4.7.

AWS. 2026. Neuron Kernel Interface (NKI).

Jonathan Bentz. 2025. Simplify gpu programming with nvidia cuda tile in python.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and 1 others. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*.

Michael Han Daniel Han and Unsloth team. 2023. Unsloth.

Tri Dao. 2024. Flashattention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations*, volume 2024, pages 35549–35562.

Yaoyao Ding, Bohan Hou, Xiao Zhang, Allan Lin, Tianqi Chen, Cody Hao Yu, Yida Wang, and Gennady Pekhimenko. 2026. Tilus: A tile-level gpgpu programming language for low-precision computation. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 281–297.

Yue Guan, Hongtao Yu, Peng Chen, Daohang Shi, Karthik Manivannan, Nicholas J Riasanovsky, Manman Ren, Lei Wang, Shane Nay, Partha Kanuparthy, and 1 others. 2026. Tlx: Hardware-native, evolvable mimw gpu compiler for large-scale production environments. *arXiv preprint arXiv:2605.10905*.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. 2020. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851.

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

Jiawei Hu, Hong Jia, Mahbub Hassan, Lina Yao, Brano Kusy, and Wen Hu. 2024. Lightllm: A versatile large language model for predictive light sensing. *Preprint*, arXiv:2411.15211.

Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, and 7 others. 2024. Mixtral of experts. *Preprint*, arXiv:2401.04088.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626.

LeetGPU. 2025. Challenges.

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

Gang Liao, Hongsen Qin, Ying Wang, Alicia Golden, Michael Kuchnik, Yavuz Yetim, Jia Jiunn Ang, Chunli Fu, Yihan He, Samuel Hsia, and 1 others. 2025. Kernelevolve: Scaling agentic kernel coding for heterogeneous ai accelerators at meta. *arXiv preprint arXiv:2512.23236*.

Edward Lin, Sahil Modi, Siva Kumar Sastry Hari, Qijing Huang, Zhifan Ye, Nestor Qin, Fengzhe Zhou, Yuan Zhang, Jingquan Wang, Sana Damani, and 1 others. 2026. Sol-execbench: Speed-of-light benchmarking for real-world gpu kernels against hardware limits. *arXiv preprint arXiv:2603.19173*.

Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. 2008. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55.

John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable parallel programming with CUDA.

NVIDIA. NVIDIA Nsight compute.

NVIDIA. 2024a. cuBLAS library.

NVIDIA. 2024b. CUTLASS: CUDA templates for linear algebra subroutines.

NVIDIA. 2025. ComputeEval: Evaluating large language models for cuda code generation.

NVIDIA. 2026. PTX writer's guide to interoperability.

OpenAI. 2026. Introducing GPT-5.5.

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

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning transferable visual models from natural language supervision. *Preprint*, arXiv:2103.00020.

Mark Saroufim, Jiannan Wang, Bert Maher, Sahan Paliskara, Laura Wang, Shahin Sefati, and Manuel Candales. 2025. Backendbench: An evaluation suite for testing how well llms and humans can write pytorch backends.

Benjamin Spector, Simran Arora, Aaryan Singhal, Arjun Parthasarathy, Dan Fu, and Christopher Re. 2025. Thunderkittens: Simple, fast, and Adorable kernels. In *International Conference on Learning Representations*, volume 2025, pages 2097–2130.

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

Triton Document. triton.language.make_tensor_descriptor.

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

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2024. Smoothquant: Accurate and efficient post-training quantization for large language models. *Preprint*, arXiv:2211.10438.

Shanli Xing, Yiyan Zhai, Alexander Jiang, Yixin Dong, Yong Wu, Zihao Ye, Charlie Ruan, Yingyi Huang, Yineng Zhang, Liangsheng Yin, and 1 others. 2026. Flashinfer-bench: Building the virtuous cycle for ai-driven llm systems. *arXiv preprint arXiv:2601.00227*.

Divakar Kumar Yadav, Tian Zhao, and Deepak Kumar. 2026. Evaluating cuda tile for ai workloads on hopper and blackwell gpus. *arXiv preprint arXiv:2604.23466*.

Biao Zhang and Rico Sennrich. 2019. Root mean square layer normalization. *Preprint*, arXiv:1910.07467.

Keren Zhou, Tianle Zhong, Hao Wu, Jihyeong Lee, Yue Guan, Yufei Ding, Corbin Robeck, Yuanwei Fang, Jeff Niu, and Philippe Tillet. 2026. Proton: Towards multi-level, adaptive profiling for triton. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 493–506.

Jiace Zhu, Wentao Chen, Qi Fan, Zhixing Ren, Junying Wu, Xing Zhe Chai, Chotiwit Rungrueangwutthinon, Yehan Ma, and An Zou. 2026. Cudabench: Benchmarking llms for text-to-cuda generation. *Preprint*, arXiv:2603.02236.

Appendix

A Benchmark Information

A.1 Full Benchmark List

Table 1: TritonBench-derived operators in TileBench.

Operator Name	Category	Explanation
flash-attention	Matrix Mult./Attn.	Standard prefill/dense attention.
flash-decode	Matrix Mult./Attn.	Flash decoding attention.
block-sparse-attn	Matrix Mult./Attn.	Block-sparse attention.
softmax	Reduction/Norm.	Standard Softmax.
rope-embedding	Point-wise	Positional embedding.
matmul_fp32_fp16_fp8	Matrix Mult./Attn.	Standard GEMM (supports FP16 & FP8).
matmul-int8	Matrix Mult./Attn.	Quantized GEMM.
streamk-matmul	Matrix Mult./Attn.	Stream-K scheduling optimization.
layernorm	Reduction/Norm.	Standard layer normalization.
rmsnorm	Reduction/Norm.	RMS normalization.
vector-addition	Point-wise	Binary addition.
mul2	Point-wise	Elementwise multiplication.
relu	Point-wise	Activation function.
destindex	Data Layout	Indexing/scatter/gather operations.
quantize-global	Point-wise	Packing values.
dequantize-rowwise	Point-wise	Unpacking values.
cross-entropy	Reduction/Norm.	Elementwise loss calculation.
matrix-transpose	Data Layout	Transpose / permute.
dropout	Point-wise	Random mask generation + apply.
swiglu	Point-wise	Complex gated activation.
kl-divergence	Reduction/Norm.	Divergence metric calculation.
fused-activation	Point-wise	Generic fused kernel container.
mean-reduction	Reduction/Norm.	Sum/mean reduction.
argmax	Reduction/Norm.	Index finding.
l2-norm	Reduction/Norm.	Norm calculation.
2d-conv	Stencil/Conv.	2D convolution.

Table 2: LeetGPU-derived operators in TileBench.

Operator Name	Category	Explanation
top-k-selection	Data Layout	Select the top-K largest elements from an array.
histogramming	Reduction/Norm.	Count occurrences of each value into histogram bins.
linear-self-attn	Matrix Mult./Attn.	Compute linear attention using kernel feature maps.
jacobi-stencil-2d	Stencil/Conv.	Jacobi stencil 2D.
bitonic-sort	Data Layout	Sort an array of floats in ascending order.
radix-sort	Data Layout	Sort 32-bit unsigned integers in ascending order via radix sort.
batched-matmul	Matrix Mult./Attn.	Batched matrix multiplication.
batch-normalization	Reduction/Norm.	Batch normalization.
2d-max-pooling	Stencil/Conv.	2D max pooling.
gaussian-blur	Stencil/Conv.	Apply a Gaussian blur filter via 2D convolution.
1d-conv	Stencil/Conv.	1D convolution.
3d-conv	Stencil/Conv.	3D convolution.
matrix-copy	Data Layout	Copy an $N \times N$ matrix element-wise.
reverse-array	Data Layout	Reverse an array in-place.
interleave	Data Layout	Interleave two float arrays element-wise.
weight-dequant	Point-wise	Dequantize a tiled weight matrix using per-tile scale factors.
moe-topk-gating	Reduction/Norm.	Select the top-k experts per token and compute softmax gating.
leaky-relu	Point-wise	Leaky ReLU activation.
sigmoid	Point-wise	Sigmoid activation.

A.2 Operator Category

A.3 `Config.yaml` Example

Each operator ships with a `config.yaml` file declaring the case grid the harness will run (the

Category	#Ops	Representative operators
Point-wise	12	`vector_add`, `relu`, `swiglu`, `weight_dequant`
Reduction/Normalization	11	`softmax`, `layernorm`, `argmax`, `moe_topk_gating`
Matrix Multiplication/Attention	8	`matmul`, `matmul_int8`, `flash_attention`, `linear_self_attention`
Stencil/Convolution	6	`1d_conv`, `3d_conv`, `gaussian_blur`, `jacobi_stencil_2d`
Data Layout	8	`matrix_copy`, `matrix_transpose`, `bitonic_sort`, `radix_sort`
Total	45	

Table 3: TileBench operator categories. TileBench contains 45 operators: 26 adapted from TritonBench and 19 adapted from LeetGPU.

```
  "gpu": "NVIDIA B200",
  "peak_bw_GBs": 6539.4,
  "peak_tflops": {
    "fp64": 37,
    "fp32": 1100,
    "tf32": 1100,
    "fp16": 2250,
    "bf16": 2250,
    "fp8_e4m3fn": 4500,
    "fp8_e5m2": 4500,
    "int8": 4500
  },
```

Figure 6: B200.json Example

```
benchmark:
  warmup: 20
  repeat: 100
  use_cuda_graph: true
  flush_l2: true
  autotune: false
case_defaults:
  M: 2048
case_grid:
  N: [20480]
  dtype: ["fp16", "fp32"]
metrics:
  flops_expr: "M * N"
  bytes_expr: "M * N * dtype_size"
  plots:
    - latency_ms
    - bandwidth_GBs
    - speedup
```

Figure 7: `argmax` (`argmax.yaml`) Example

cross product of input sizes and dtypes) and the verification tolerances applied to the generated kernel's output against the PyTorch reference. The configuration is per-operator (not global) because different operators have meaningfully different shape-sensitivity profiles: matmuls are exercised at sizes where tensor-core paths kick in, reductions at sizes where occupancy starts saturating, and elementwise ops at sizes that span memory-bound and compute-bound regimes. The verify-tolerance defaults follow standard mixed-precision practice; per-operator overrides are used where the reference itself is non-associative across reduction order, in which case the tolerance is widened to absorb floating-point reassociation error.

A.4 B200 JSON Information

We use per-GPU dense peak compute ceilings derived from the NVIDIA HGX B200 / DGX B200 specifications, excluding 2:4 sparsity. For FP32 dot/MMA workloads, we use the TF32 Tensor Core dense ceiling because Triton `tl.dot` uses TF32 by default for FP32 inputs. The peak bandwidth value, 6539.4 GB/s, is measured on our B200 system rather than taken from the datasheet HBM bandwidth.

B Iterative LLM Code-Generation Prompts

This appendix documents the prompts used in the iterative LLM code-generation track described in Section 3.3. We show the iteration-0 and iteration-N prompt templates, followed by the building blocks they reference: the TileBench framework guide, the per-backend API references, the operator problem description, the PyTorch reference implementation, and the strict output-format instructions. The operator's `config.yaml` is shown separately in Appendix A.3. Throughout this section we use `vector_add` as the running example operator; for every other operator the same skeleton is filled in with the corresponding operator-specific files.

B.1 Iteration 0 Prompt Template

The iteration-0 prompt is assembled by concatenating the components below in the order shown. The model receives no prior context other than the system prompt; everything operator-specific is fed as the user message.

Iteration 0 Prompt Template

```
# Task: implement operator `<op>` for TileBench

Follow the framework conventions below. Then implement the operator
described. You will emit `impl_triton.py`, `impl_cutile.py`.

---

# TileBench Framework Conventions
<<< framework guide -- Cref{appsubsec:framework_guide} >>>

---

# Triton (triton / triton.language) API Reference
<<< Triton API reference -- Cref{appsubsec:api_reference} >>>

---

# cuTile (cuda.tile) API Reference
<<< cuTile API reference -- Cref{appsubsec:api_reference} >>>

---

# Operator description: `<op>`
<<< natural-language problem description --
    Cref{appsubsec:problem_description} >>>

---

# `config.yaml` for `<op>`
<<< config.yaml -- Cref{appsubsec:config.yaml_example} >>>

---

# PyTorch reference: `impl_torch.py`
<<< impl_torch.py -- Cref{appsubsec:torch_reference} >>>

---

<<< output-format instructions --
    Cref{appsubsec:output_format} >>>
```

B.2 Iteration N Prompt Template ($N > 0$)

For iterations after iteration 0, the prompt is rebuilt from scratch each iteration rather than appended to a running chat. It still ends with the framework guide, API references, problem description, `config.yaml`, PyTorch reference, and output-format instructions (identical to iteration 0), but is prefixed with a trajectory table, the previous iteration's source files, and, when the previous iteration regressed or failed verification, the best verify-clean source files seen so far as a rebuild reference.

Iteration N Prompt Template ($N > 0$)

```
# Task: improve operator `<op>` for TileBench (iteration N)

This is iteration N of the refinement loop. Review the trajectory
and feedback below. If the latest iteration regressed or failed
verification, consider rebuilding from the best verify-clean
implementation seen so far.
Then re-emit the implementation file(s) for the backend(s) that remain
active.

## Iteration trajectory so far

<<< per-iteration backend configuration, stop_score,
    speedup_vs_torch, and verification status >>>

<<< best verify-clean iteration for each backend and
    regression warnings, when applicable >>>

## Feedback from iteration N-1

<<< compile/timeout errors, verification failures,
    per-backend performance summary, and worst-first
    roofline feedback >>>

---

## Previous generated implementations

```python
<previous impl_triton.py source>
```

```python
<previous impl_cutile.py source>
```

Best verify-clean implementation

<<< included only when the latest implementation regressed
    or failed verification and an earlier verify-clean
    implementation is available >>>

```python
<best verify-clean impl_<backend>.py source>
```

```

```
# Reminders: TileBench Framework Conventions
<<< framework guide -- \Cref{appsubsec:framework_guide} >>>

---

# Triton (triton / triton.language) API Reference
<<< Triton API reference -- \Cref{appsubsec:api_reference} >>>

---

# cuTile (cuda.tile) API Reference
<<< cuTile API reference -- \Cref{appsubsec:api_reference} >>>

---

# Operator description: `<op>`
<<< problem description -- \Cref{appsubsec:problem_description} >>>

---

# `config.yaml` for `<op>`
<<< config.yaml -- \Cref{appsubsec:config.yaml_example} >>>

---

# PyTorch reference: `impl_torch.py`
<<< impl_torch.py -- \Cref{appsubsec:torch_reference} >>>

---

<<< output-format instructions -- \Cref{appsubsec:output_format} >>>
```

B.3 TileBench Framework Guide

The framework guide specifies the file structure each implementation must produce, the `run(*args, **kwargs)` and `get_last_config()` contract, the forbidden patterns (`triton.autotune`, `cutileAutotuner`, delegation to PyTorch / cuDNN / cuBLAS, output caching), recommended tile-size bounds for B200, multi-dtype handling, and a pre-return checklist. An abridged transcription is shown below: markdown emphasis and decorative emoji from the source file have been stripped, but section headings, rules, code templates, and numeric bounds are reproduced as in the original.

TileBench Framework Guide (`framework_guide.md`)

```
# TileBench Framework Guide

Read this carefully. Your generated kernels MUST follow this contract
or the benchmark harness will fail to load / run / evaluate them.

## Files you must produce

For operator `<op>`, write exactly two files:

1. `impl_triton.py` -- Triton kernel + Python `run()` wrapper
2. `impl_cutile.py` -- cuTile kernel + Python `run()` wrapper

You do NOT generate `impl_torch.py` -- it is copied from the
human-written reference implementation. Your `run()` function's
signature MUST match `impl_torch.run()` exactly (same positional
args, same keyword args).

## Required exports (both impls)

<<< `run(...)` and `get_last_config()` interface definitions >>>

## No autotune -- you pick one configuration per iteration

The harness does NOT call autotune.
Each active backend runs with a single hard-coded configuration that YOU choose.
The refinement loop, with a maximum budget of 10 iterations per backend, is the search mechanism:  at each executed iteration, the  model sees the previous iteration's (config -> roofline_pct, latency_ms, speedup_vs_torch) and proposes the next configuration.

The following are FORBIDDEN in generated  code and will be rejected at load:

- `triton.autotune(...)` decorator
- `from core.cutile_autotune import CutileAutotuner`
- `ct.tune.exhaustive_search` / `ct_experimental.autotune_launch`
- Any other in-kernel autotune-style search

## Triton template (no autotune)

<<< Triton template and configuration-recording rules >>>

## cuTile template (no autotune)

<<< cuTile template and backend-specific  implementation rules >>>

```

```
## Choosing tile / BLOCK sizes

<<< B200-specific tile-size bounds and recommended starting configurations >>>

## Multi-dtype support

If `config.yaml`'s `case_grid.dtype` lists multiple dtypes (e.g. ["fp16", "bf16", "fp32"]), the `run()` function must work for ALL of them in a single `run()` call with ONE configuration shared across all dtypes. Use a single dtype-polymorphic kernel:

<<< backend-specific multi-dtype implementation guidance >>>

## Hardware constraints to remember (B20 0, sm_100)

<<< B200-specific TMA, Tensor Core, and tile-shape constraints >>>

## FORBIDDEN: delegating computation to PyTorch / cuDNN / cuBLAS

The `run()` function in `impl_triton.py` MUST perform the operator's computation through the `@triton.jit` kernel, and `run()` in `impl_cutile.py` MUST perform it through the `@ct.kernel`. The following are NOT allowed inside `run()`:

<<< prohibited delegation patterns and permitted non-compute PyTorch operations >>>

## FORBIDDEN: caching outputs across `run()` calls

Do NOT add a Python-level cache that returns a previously-computed output when the inputs look identical.

<<< anti-cache constraints and harness checks >>>

## Common pitfalls (the harness rejects these)

<<< framework validation and common implementation pitfalls >>>

## What the engine does with generated code

<<< per-case input generation, verification, timing, and configuration-feedback procedure >>>

The refinement loop allows up to 10 iterations per backend.
Triton and cuTile are tracked independently.
After each executed iteration, the engine evaluates verification and roofline utilization.
A backend is frozen once its implementation passes verification and reaches at least 80% roofline utilization.
Otherwise, that backend remains active until the 10-iteration budget is exhausted.

Frozen backends are skipped in subsequent iterations and incur no additional token cost.
When refinement ends, the best verification-passing iteration among the executed iterations is promoted as the final result.
The `stop_score` recorded in the trajectory corresponds to the roofline utilization criterion used for stopping and is included in the feedback for subsequent iterations.

## TL;DR checklist before returning code

<<< pre-return checklist >>>
```

B.4 Triton and cuTile API References

For each active backend, the prompt prepends an authoritative API reference that scopes the model to the exact library versions used by the harness (Triton 3.6.0 and the cuTile pip packages shipped with TileBench). Each reference is a long document; reproducing either in full would dominate the appendix, so we describe their structure and show a representative excerpt.

Structure. Each API reference is organized as numbered topical sections that, at a high level, cover (i) import boilerplate and data types, (ii) the kernel programming model and tile creation, (iii) memory operations, (iv) compute primitives (arithmetic, reductions, matrix multiply), (v) control flow and synchronization, (vi) compiler hints, (vii) the autotune pattern (explicitly disabled in this pipeline; see the preamble below), (viii) the TileBench `run()` / `get_last_config()` contract and common kernel patterns, and (ix) critical constraints, gotchas, and a quick-reference cheatsheet. The exact section list differs between the two references because Triton and cuTile expose different abstractions.

Preamble (Triton). Each reference is prefixed by a short preamble that anchors the model to the

installed library version and explicitly invalidates the reference's autotune content (since the LLM track forbids autotune). The Triton preamble is:

Triton API Reference Preamble

```
Treat the API reference below as authoritative for the Triton version
installed in this repo (3.6.0); do NOT use APIs from later versions
you may have seen in training data. Every `tl.*` / `triton.*` symbol
you use must appear in this reference.

IMPORTANT -- autotune sections of this reference do NOT apply. Use
the reference for kernel-body syntax (tl.load, tl.store, tl.dot,
masking, make_block_ptr, make_tensor_descriptor, etc.); ignore
everything about cfg search / autotune wrappers.
```

The cuTile preamble is analogous.

Section excerpt. As a concrete example of the section-level granularity, the cuTile reference's memory-operations section opens as shown below. Each entry gives the function signature, a parameter list, and a few representative call examples; per-section content varies in length but follows this shape.

cuTile API Reference: §6 Memory Operations (Excerpt)

```
## 6. Memory Operations

### 6.1 ct.load()

Signature:
    tile = ct.load(array, index, shape, *,
                   order="C",
                   padding_mode=PaddingMode.UNDETERMINED,
                   latency=None,
                   allow_tma=None)

Parameters:
    array        : Array
                   Source array (torch tensor)
    index        : tuple[int | Tile, ...]
                   Tile-space coordinates
    shape        : tuple[const int, ...]
                   Tile shape (all dims must be power of 2);
                   () for 0D scalar tile
    order        : "C" / "F" / tuple
                   Axis mapping: "C"=identity, "F"=reversed,
                   tuple=explicit permutation
    padding_mode : PaddingMode
                   OOB element handling
    latency      : const int
                   DRAM traffic hint, 1 (low) to 10 (high)
    allow_tma    : const bool
                   Enable/disable TMA (default True)

Examples:
    tile   = ct.load(x, index=(bid,), shape=(TILE,))             # 1D
    tile   = ct.load(x, index=(row, j), shape=(1, TILE),
                     padding_mode=ct.PaddingMode.ZERO)           # 2D padded
    tile   = ct.load(Q, index=(b, h, i), shape=(1,1,M,D))     # 4D
    tile   = ct.load(K, index=(b, h, 0, j), shape=(1,1,D,N),
                     order=(0,1,3,2))                          # transposed
    scalar = ct.load(x, index=(i,), shape=())                    # 0D scalar

### 6.2 ct.store()

Signature:
    ct.store(array, index, tile, *,
             order="C",
             latency=None,
             allow_tma=None)

Parameters:
    array     : Array
                Destination array
    index     : tuple[int | Tile, ...]
                Tile-space coordinates
    tile      : Tile or scalar
                Data to store
    order     : "C" / "F" / tuple
                Axis mapping
    latency   : const int
                Latency hint
    allow_tma : const bool
                Enable/disable TMA

OOB writes are silently ignored (no-op).
```

B.5 Operator Problem Description

Each operator has a natural-language problem description at `benchmarks/problems/current/<op>_current.md` that gives the mathematical definition, the input contract, the output contract, and a small worked example. Below we show the file for the `vector_add` operator as a representative example.

Problem Description (`vector_add_current.md`)

```
# Vector Addition

Implement a program that performs element-wise addition of two 1D arrays of the same length.

The input consists of two arrays:

- `x`: A 1D array.
- `y`: A 1D array of the same length as `x`.

The output should be written to the `output` array, which has the same length as the inputs.

The operation is defined mathematically as:

    output[i] = x[i] + y[i]

where i ranges from 0 to n-1.

## Implementation Requirements

- Use only native features (external libraries are not permitted)
- The `solve` function signature must remain unchanged
- The final result must be stored in the array `output`

## Example 1

    Input:  x = [1, 2, 3, 4], y = [5, 6, 7, 8]
    Output: [6, 8, 10, 12]

## Example 2

    Input:  x = [-1.5, 2.0, 0.0], y = [0.5, -2.0, 3.0]
    Output: [-1.0, 0.0, 3.0]

## Constraints

- 1 <= n
- `x` and `y` have the same length
```

B.6 PyTorch Reference Implementation

The PyTorch reference (`impl_torch.py`) is included verbatim in the prompt. It serves both as a precise specification of the operator's input/output contract and as the correctness oracle the generated kernels are validated against. For `vector_add` the reference is:

PyTorch Reference (`vector_add/impl_torch.py`)

```
def run(x, y):
    return x + y
```

The harness checks that the generated kernels' `run()` has the same positional signature as the reference, and that its output matches the reference within the tolerances declared in `config.yaml`'s `verify:` block (or per-dtype defaults when absent).

B.7 Output Format Instructions

The final block of every prompt enforces a strict, machine-parseable output format. Generated text is parsed by a regular expression that matches "‘‘‘`python title="impl_<backend>.py"` fenced code blocks; anything outside the requested blocks is discarded. At each executed iteration, implementation files are emitted for the backend(s) that remain active, following the instructions below:

Output Format Instructions

````
## Output format (STRICT)

Return the fenced code block(s) for the active backend(s), in the order shown below, using the corresponding title(s):

    ```python title="impl_triton.py"
    # full Python file content here
    ```

    ```python title="impl_cutile.py"
    # full Python file content here
    ```
Emit ONLY the implementation file(s) for the active backend(s).
No other code blocks.
No prose between or after the blocks beyond a 1-2 sentence summary of your approach. Do NOT include test code, do NOT include PyTorch reference code (that file is provided by the framework).
````

C Additional Evaluation Metrics

C.1 Evaluation Modes and Aggregation Conventions

Per-case timing. For operator o, backend b, dtype d, and input case c, the benchmark records latency $T_{o,b,d,c}$. This is the arithmetic mean of 100 timed launches following 20 warmup launches. CUDA graph replay is enabled, and the L2 cache is evicted before each measured launch outside the timed Proton scope. The complete operator `run()` is timed as one unit, so a multi-kernel implementation retains any intended producer-to-consumer cache reuse inside the operator boundary.

Geometric-mean aggregation. Every paper-level aggregate above the per-case latency uses a geometric mean. The per-case speedup over PyTorch is

$$S_{o,b,d,c} = \frac{T_{o,\mathrm{torch},d,c}}{T_{o,b,d,c}}. \tag{1}$$

For the complete sweep of operator o, we assign equal weight to every valid dtype and case pair:

$$S_{o,b} = \mathrm{GM}_{(d,c)\in\mathcal{V}_o} S_{o,b,d,c}, \tag{2}$$

where $\mathcal{V}_o$ is the set of valid cases for operator o. The suite-level result gives equal weight to each of the 45 operators:

$$S_b^{\mathrm{suite}} = \mathrm{GM}_{o\in\mathcal{O}} S_{o,b}. \tag{3}$$

Category results use the same construction over operators in the category. The direct operator-level Triton and cuTile ratio is

$$Q_o = \frac{S_{o,\mathrm{Triton}}}{S_{o,\mathrm{cuTile}}}. \tag{4}$$

Thus, $Q_o > 1$ favors Triton and $Q_o < 1$ favors cuTile. Because the PyTorch factor cancels case by case, Q_o is also the geometric mean of $T_{o,\mathrm{cuTile},d,c}/T_{o,\mathrm{Triton},d,c}$ over the complete sweep. Medians reported in RQ1 and RQ2 are distribution statistics over the 45 operator-level values and are not used as averages.

Roofline aggregation and tuning gain. Let $R^m_{o,b,d,c}$ denote the roofline utilization of mode $m \in \{\mathrm{default}, \mathrm{autotuned}\}$. The operator-level value plotted in the paper is

$$\bar{R}^m_{o,b} = \mathrm{GM}_{(d,c)\in\mathcal{V}_o} R^m_{o,b,d,c}. \tag{5}$$

The operator-level autotuning gain is

$$G_{o,b} = \mathrm{GM}_{(d,c)\in\mathcal{V}_o} \frac{R^{\mathrm{autotuned}}_{o,b,d,c}}{R^{\mathrm{default}}_{o,b,d,c}}, \tag{6}$$

and the suite-level gain is $\mathrm{GM}_o\, G_{o,b}$. Threshold counts such as $R \geq 0.8$ are evaluated on $\bar{R}^{\mathrm{autotuned}}_{o,b}$.

RQ3 case selection. RQ3 does not average NCU counters across the sweep. For each supported operator–dtype pair, we select the maximum case recorded in the NCU catalogue and profile the exact autotuned winner used for that case. Performance ratios remain based on the corresponding Proton/CSV latency. NCU counters explain the execution mechanism but do not replace the benchmark latency.

RQ4 aggregation. For the LLM track, the correctness-gated speedup of an executed iteration is the PyTorch-to-generated-kernel latency ratio of the representative large verify-clean case saved by the evaluation pipeline. It is a single-case ratio rather than an average over dtypes. $\mathrm{BestSpeedup@10}$ is the maximum verify-clean speedup observed before the run ends or the backend freezes. $\mathrm{TokenCost@10}$ sums the tokens consumed by the executed iterations for that backend; skipped iterations after freezing contribute neither tokens nor speedup. $\mathrm{TokenEfficiency@10}$ divides the best speedup by token cost in millions.

C.2 NCU Profiling Protocol and Kernel Attribution

The systematic Triton and cuTile profiling set contains 220 profiles: 110 supported operator–dtype pairs for Triton and the same 110 pairs for cuTile. Each profile uses the NCU-catalogued maximum input and the same autotuned winning configuration evaluated by the main benchmark. PyTorch is profiled selectively when its internal dispatch path is needed to interpret the software-baseline comparison, for example cuDNN convolution, cuBLAS GEMM, CUB radix sort, compiled `flex_attention`, or a generic ATen reduction.

Latency source. All operator latency and speedup values are obtained from Proton measurements. NCU is used only for profiling and diagnosing performance phenomena, and its reported kernel duration is not used in our performance results.

Multi-kernel attribution. For a single-kernel implementation, the profile directly corresponds to the measured operator. For a multi-kernel implementation, the CSV latency covers the full kernel sequence, while counter-level observations are attributed to the stage that generates them. Examples include the two `destindex` scatter kernels; the five `linear_self_attention` launches grouped into activation, KV-GEMM, column-reduction, and output-GEMM stages; and the repeated histogram, scan, and scatter stages of `radix_sort`.

Diagnostic procedure. For each profiled case, we analyze the performance gap as follows:

1. Inspect the source implementation, including the algorithm, task decomposition, precision path, and selected autotuned configuration;

2. Inspect the generated PTX, SASS, and source-correlated hotspots;

3. Examine dynamic instruction counts, memory requests and sectors, TMA and Tensor Core usage, register and shared-memory usage, occupancy, pipeline activity, and stall behavior;

4. Relate these profiling observations to the corresponding Proton-measured latency.

Higher-level compiler IRs were not materialized in the finalized artifacts used for this analysis, and no compilation was rerun during validation. We therefore do not quote TTIR, TTGIR, or tile IR. Compiler-lowering claims are instead based on PTX, SASS, and source-correlated evidence preserved in the NCU reports. Opcode-family counts are extracted from NCU source tables. When source correlation undercounts instructions associated with a source line, the counter-level `smsp__inst_executed.sum` total remains authoritative, and the per-family executed count is treated as a lower bound.

A stall metric alone is not treated as causal evidence. For example, a long-scoreboard sample on an MMA line may indicate that the MMA is waiting for a preceding gather or staging chain rather than that Tensor Core arithmetic is intrinsically slow. We therefore interpret stalls together with instruction mix, dependency producers, resource use, memory transactions, and measured latency.

Scope of the diagnosis. The profiles describe the current source implementations, compiler versions, autotuned winners, and B200 architecture. They support recurring mechanisms across the suite, but they do not imply that every occurrence of a given source primitive must produce the same hardware path. The generated instruction sequence remains dependent on dtype, tile shape, layout, and compiler decisions.

C.3 PyTorch Baseline

PyTorch serves as the semantic reference, correctness oracle, and practical software baseline for reporting speedup. Its execution path is operator dependent and may involve eager tensor operations, generic ATen kernels, compiled kernels, or optimized vendor libraries. Therefore, speedup over PyTorch reflects the practical benefit of a custom kernel over the available software path, while RQ3 focuses primarily on explaining the performance differences between the matched Triton and cuTile implementations.

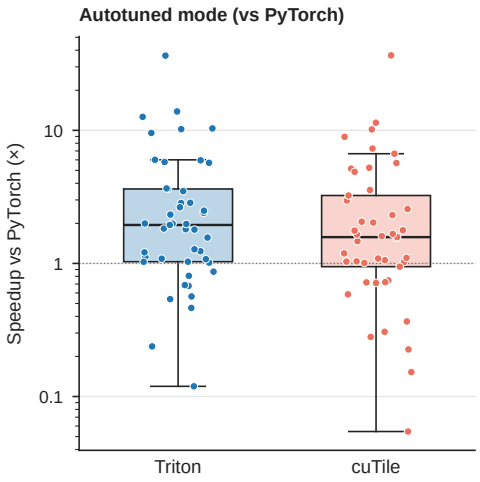
D Additional Evaluation Results

D.1 RQ1: Autotuned Speedup Distributions and Category Breakdowns

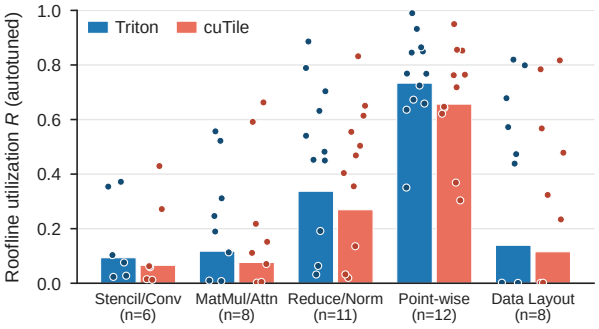
Figure 8 complements the operator scatter in Section 5.1. The median operator-level speedup is $1.95\times$ for Triton and $1.58\times$ for cuTile. 9 Triton implementations and 12 cuTile implementations remain below PyTorch. The largest operator-level speedups for both DSLs reach approximately $36\times$ on `bitonic_sort`, whose PyTorch reference is an eager multi-pass bitonic network rather than using `torch.sort`. The large speedup therefore reflects, in part, the overhead of the PyTorch reference execution path rather than only the efficiency of the DSL backends. This example illustrates why the speedup distribution should be interpreted together with the corresponding PyTorch baseline implementation.

Table 4 shows that the software and hardware utilization views are complementary. Point-wise operators obtain the highest category-level roofline utilization and exhibit many near-parity Triton and cuTile results. Reduction / Normalization kernels also achieve strong speedups over PyTorch, often because the DSL kernels fuse operation or specialize a generation reduction template. Data Layout operators show the largest speedups over PyTorch but much lower roofline utilization, because the category contains multi-pass sorting and selection workloads whose PyTorch references and DSL algorithms have different launch and traffic structures. Stencil / Convolution and Matrix Multiplication / Attention remain the least utilized categories. Their arithmetic work is substantial, but performance is frequently limited by vendor-library gaps, dynamic indexing, state placement, or pipeline formation rather than peak Tensor Core throughput alone.

Triton has the highest category-level speedup in all five groups, but this does not imply that cuTile is uniformly weaker inside a category. cuTile wins the most informative regular tiled cases, including `matmul_fp32_fp16_fp8`, `flash_attention`, and `jacobi_stencil_2d`. Its category aggregates are instead reduced by a smaller number of large deficits on irregular indexing, small reductions, sparse traversal, and runtime-controlled loops.



(a) Autotune-mode speedup distribution across operators.



(b) Per-category roofline utilization R (Autotune-mode).

Figure 8: RQ1 extension: speedup distribution and per-category roofline utilization.

Category	Speedup over PyTorch Triton	cuTile	Roofline Utilization Triton	cuTile	Triton wins	cuTile wins
Point-wise	2.07×	1.84×	0.733	0.657	11	1
Reduction/Norm	2.56×	2.05×	0.337	0.269	9	2
Data Layout	3.37×	2.79×	0.139	0.115	6	2
Stencil/Conv	1.22×	0.86×	0.093	0.066	5	1
Matrix Mult./Attn	1.22×	0.79×	0.118	0.076	6	2

Table 4: Category-level geometric means under autotuned mode. The last two columns count direct Triton and cuTile wins inside each category.

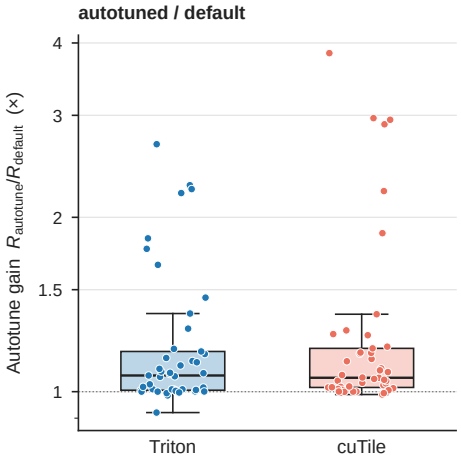


Figure 9: Triton and cuTile autotune gain $R_{\text{autotune}}/R_{\text{default}}$ per operator.

Metric	Triton	cuTile
Default suite geometric-mean R	0.209	0.160
Autotuned suite geometric-mean R	0.248	0.194
Geometric-mean tuning gain	1.18×	1.22×
Median tuning gain	1.07×	1.06×
Autotuned $R \geq 0.8$	7/45	5/45
Autotuned $R \geq 0.9$	2/45	1/45

Table 5: Aggregate RQ2 results.

D.2 RQ2: Per-operator Autotuning Gains and Roofline Headroom

The suite-level geometric-mean tuning gain is 1.18× for Triton and 1.22× for cuTile, while the median gains are 1.07× and 1.06×, respectively. The larger cuTile geometric mean is driven by several high-gain operators rather than a uniformly larger shift. The strongest Triton gains are `argmax` (2.67×), `histogramming` (2.27×), `3d_conv` (2.24×), and `2d_conv` (2.20×). The strongest cuTile gains are `linear_self_attention` (3.84×), `1d_conv` (2.97×), `2d_conv` (2.95×), `argmax` (2.90×), and `3d_conv` (2.22×).

The autotuned result is the best measured candidate in the declared search space and is not clamped to the manually selected default. 4 Triton operator-level results and 4 cuTile results are therefore slightly below their defaults. 7 of these 8 regressions are within approximately 1.5% of parity.

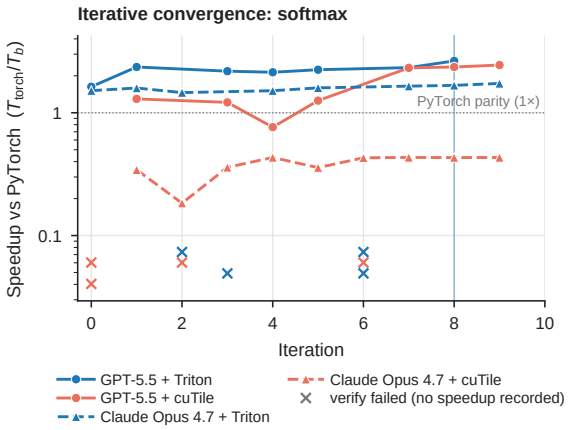


Figure 10: Iteration-level correctness-gated speedup trajectory for `softmax`. Frozen or failed backend iterations do not contribute a new valid speedup point.

The main exception is Triton `top_k_selection`, whose tuned aggregate is $0.92\times$ its default. We retain these values rather than replacing them with the default because they accurately describe the implemented tuning protocol.

The results separate two kinds of headroom. Configuration headroom is recovered by changing tile shape, warp count, pipeline stages, or occupancy. Backend headroom remains when the source implementation and compiler continue to generate an expensive instruction path, an unsuitable reduction organization, excessive register or shared-memory state, or an ineffective software pipeline. The small number of operators above 80% utilization after tuning shows that the second kind of headroom dominates much of the suite.

D.3 RQ4: Iterative Refinement Trajectories

The LLM track allows at most 10 refinement iterations. A backend freezes once it is verify-clean and reaches the stopping threshold, after which it incurs no additional token cost. Failed executed iterations remain part of the token total because compilation and correctness failures are part of the search process.

For Triton, both GPT-5.5 and Claude Opus 4.7 produce at least one verify-clean implementation for all 45 operators. With cuTile, GPT-5.5 achieves verify-clean coverage on 43/45 operators and Claude Opus 4.7 on 41/45; for the remaining 2 and 4 operators, respectively, none of the generated implementations within the 10-iteration budget passes verification.

Figure 10 illustrates why a single final score does not fully describe the refinement process. GPT-5.5 + Triton reaches a valid implementation early and peaks at approximately $2.64\times$ in iteration 8. Claude Opus 4.7 + Triton reaches approximately $1.73\times$ by iteration 9. GPT-5.5 + cuTile fails verification at iteration 0, later drops to approximately $0.76\times$, and reaches approximately $2.45\times$ by iteration 9. Claude Opus 4.7 + cuTile remains near $0.43\times$. The trajectories show that later refinement can improve both backends, but the final experiment does not vary the iteration budget and therefore does not establish whether a larger budget would eliminate the backend gap.

The LLM results quantify an empirical usability difference under the TileBench protocol. They do not isolate whether the cause is API expressibility, primitive availability, compiler constraints, documentation, public examples, or prior model exposure. We therefore avoid treating model familiarity as a demonstrated causal explanation.

E Computational Budget

All experiments were run on a single NVIDIA B200 GPU with 180GB HBM3e memory. The main benchmark, autotuning, NCU report generation experiments consumed approximately 100 GPU-hours in total.

F LLM API Usage

For the iterative LLM code-generation track, GPT-5.5 consumed 18.68M API tokens with a total cost of $279.05, while Claude Opus 4.7 consumed 21.09M API tokens with a total cost of $191.66. In total, the LLM track consumed 39.77M API tokens and cost $470.71. Because both evaluated LLMs are closed-source API models, their parameter counts are not publicly available.

G Artifact Sources and Licenses

TileBench uses TritonBench and LeetGPU as sources of operator coverage and task semantics. TritonBench is distributed under the Apache License 2.0. For TritonBench-derived components, we preserve the original attribution and license notices and document our modifications when applicable.

LeetGPU is distributed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International license (CC BY-NC-ND 4.0). This license permits non-commercial sharing with attribution, but does not permit redistribution of adapted material. Accordingly, TileBench does

Model and backend	Verify-clean coverage	Faster than PyTorch	TokenCost@10	TokenEfficiency@10
GPT-5.5 + Triton	45/45	38/45	0.260M	15.83
GPT-5.5 + cuTile	43/45	36/43	0.331M	10.79
Claude Opus 4.7 + Triton	45/45	38/45	0.277M	12.90
Claude Opus 4.7 + cuTile	41/45	33/41	0.441M	7.61

Table 6: Final LLM-track coverage and geometric-mean token metrics. The faster-than-PyTorch denominator in the cuTile rows is the number of operators with at least one verify-clean implementation.

not redistribute LeetGPU source code, test files, or modified problem statements. For operators selected from LeetGPU challenge topics, we provide independently written PyTorch references, Triton implementations, cuTile implementations, and TileBench configuration files, and we cite the original LeetGPU artifact.

TileBench artifacts are newly written by the authors. Third-party artifacts, tools, and dependencies remain governed by their original licenses and terms of use.