

000 001 002 003 004 005 006 007 008 009 010 011 012 013 014 015 016 017 018 019 020 021 022 023 024 025 026 027 028 029 030 031 032 033 034 035 036 037 038 039 040 041 042 043 044 045 046 047 048 049 050 051 052 053 TILELANG: BRIDGE PROGRAMMABILITY AND PERFORMANCE IN MODERN NEURAL KERNELS

Anonymous authors

Paper under double-blind review

ABSTRACT

Achieving high performance in modern AI increasingly requires kernels co-designed with underlying hardware, but writing efficient kernels remains challenging due to hardware-level complexity and limited fine-grained control in compilers like Triton. In this paper, we introduce TILELANG, a programmable tile-level system that provides explicit primitives for memory placement, data movement, and parallel scheduling. Using a unified fused tile-level dataflow graph (FTG), TILELANG streamlines kernel development by unifying tile recommendation, which guides developers with hardware-aware defaults, and tile inference, which automates completion through constraint propagation. TILELANG enables concise expression of a wide range of AI algorithms in fewer than 70 lines of Python, reducing code size by up to 85.5% compared with manual implementations. Our evaluation shows that TILELANG delivers $1.08\times$ – $10.58\times$ speedups over Triton on NVIDIA H100 (3.02 \times on average) and $1.01\times$ – $11.56\times$ on AMD GPUs (2.65 \times on average), effectively bridging programmability and performance.

1 INTRODUCTION

The rapid progress of modern neural networks has driven a growing demand for highly optimized compute kernels, particularly for memory-bound operations such as attention. In recent years, modern attention algorithms such as Multi-Head Attention(MHA) (Vaswani et al., 2017), Multi-Head Latent Attention (MLA) (Liu et al., 2024), Gated Query Attention (GQA) (Ainslie et al., 2023), and Linear Attention (Gu & Dao, 2023; Dao & Gu, 2024; Sun et al., 2023; Yang et al., 2024), increasingly demand fine-grained control over memory hierarchy, scheduling, and data movement to fully utilize hardware capabilities. However, existing systems like Triton (Tillet et al., 2019) lack programmable abstractions to support this level of control. For instance, FlashMLA relies on carefully pipelined computations and shared memory reuse, but Triton gives programmers no direct control over tile reuse or pipeline scheduling, restricting performance optimization.

As a result, developers often face a steep trade-off between achieving peak performance and maintaining programmability: *they must either manually write complex CUDA kernels or sacrifice significant performance due to abstraction mismatches*. As illustrated in Figure 1, the Triton implementation of MLA requires only 130 lines of code, whose convenience comes at a steep cost—its performance reaches only 14.2% of the hand-written CUDA version (DeepSeek, 2025) (~ 500 lines) on NVIDIA H100 GPUs. Bridging the gap between programmability and performance requires addressing two key challenges. First, a programming model must give developers precise control over data movement and computation, enabling direct interaction with hardware resources. Second, a compiler must efficiently lower these high-level programs

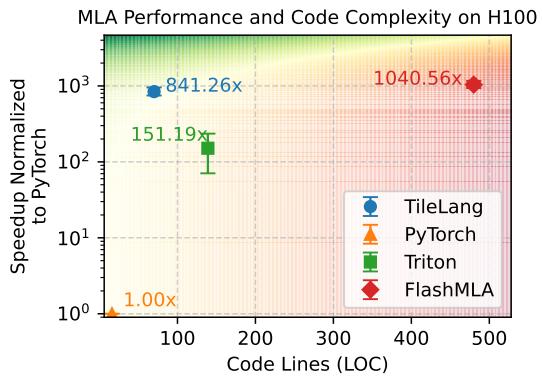


Figure 1: Performance vs. code size trade-off for MLA kernels on NVIDIA H100. Points closer to the top-left indicate better balance between performance and implementation simplicity. The annotated speedup values indicate performance gains over the PyTorch implementation.

054 to GPU code, mapping abstractions onto hard-
 055 ware resources without adding programming complexity. Solving both challenges is essential to
 056 balance developer productivity with near-peak hardware performance on modern accelerators.
 057

058 We introduce TILELANG, a *controllable programming system* for modern neural workloads. TILE-
 059 LANG provides programmable tile abstractions that let developers express and optimize low-level
 060 kernel behaviors in a high-level, composable way. Unlike existing compilers such as Triton, which
 061 rely on opaque optimization passes, TILELANG gives developers explicit control over memory, data
 062 movement, layout, and parallel execution. Specifically, developers can allocate buffers in differ-
 063 ent hardware memory levels (`alloc_shared`, `alloc_fragment`), orchestrate data transfers
 064 (`copy`), define custom memory layouts (`annotate_layout`, `use_swizzle`), and fine-tune
 065 parallelism and pipelining strategies (`Parallel`, `Pipelined`).

066 Under programmable tile abstractions, TILELANG programs can be represented as a unified fused
 067 tile-level dataflow graph (FTG). By operating on this FTG, TILELANG enables fine-grained reasoning
 068 and optimization of AI kernels, guiding developers from high-level design choices to fully specified,
 069 hardware-efficient kernel configurations. It introduces two complementary techniques. First, *tile*
 070 *recommendation* analyzes the FTG along with partially specified configurations to provide hardware-
 071 aware defaults for tile shapes, memory placement, and warp partitions, offering developers high-
 072 quality starting points that can be accepted, adjusted, or further tuned. Second, tile inference
 073 propagates shape and layout constraints across the FTG to complete the remaining configurations
 074 based on the partially annotated operators. It also automatically aligns buffer shapes, layouts, and
 075 memory allocations both downstream and upstream. This design blends flexible user control with
 076 automated optimization, yielding efficient kernels with far less manual effort.

077 As shown in Figure 1, TILELANG achieves on average $5.56\times$ the performance of Triton and
 078 approaches the hand-written CUDA version in performance, while requiring less than 16% of the code
 079 size of the manual kernel and even fewer LOCs than Triton. This highlights TILELANG’s ability
 080 to attain a more favorable balance between programmability and performance, offering both high
 081 efficiency and low development effort. We also implement other modern AI kernels—including
 082 Dequantize Matmul (Wang et al., 2024), Multi-Head Attention (MHA) (Vaswani et al., 2017), and
 083 Block-Sparse Attention (BSA) (Guo et al., 2024). Despite its deliberately streamlined interface,
 084 TILELANG achieves state-of-the-art throughput across heterogeneous GPUs, delivering speed-ups of
 085 up to $10.59\times$ over Triton on an NVIDIA H100 and $11.56\times$ on an AMD MI300X (AMD, 2024).

086 Our contributions are twofold: (1) programmable tile abstractions that let developers directly control
 087 and interact with hardware; and (2) tile recommendation and inference that guide developers with
 088 hardware-aware defaults and automatically complete configurations over a unified FTG graph. We
 089 believe TILELANG improve both the productivity and performance of modern AI kernel development.

2 RELATED WORK

090 **AI kernel programming and optimizations.** To simplify the development of AI kernels, libraries
 091 like FlashAttention-3 (Shah et al., 2024), CUTLASS (NVIDIA, 2019), and ThunderKittens (Spector
 092 et al., 2025) rely on manual or template-driven designs. Triton (Tillet et al., 2019) provides a
 093 high-level Python DSL but restricts control over critical performance paths. Gluon (OpenAI, 2025)
 094 is built on Triton DSL and exposes lower memory hierarchies like shared memory and registers.
 095 Helion (PyTorch, 2025) works as a higher-level DSL and is designed to compile down to Triton.
 096 Cypress (Yadav et al., 2025) introduces a task-based programming model with sequential semantics.
 097 Tilos (Ding et al., 2025) is another Python DSL for GPU programming, designed with thread-
 098 block-level granularity and tensors as the core data type. Mojo (Godoy et al., 2025) combines
 099 Python’s interoperability and CUDA-like syntax to build performance-portable HPC science kernels.
 100 Frameworks such as PyTorch (Paszke et al., 2019), Graphene (Hagedorn et al., 2023), MLIR (Lattner
 101 et al., 2021), and Welder (Shi et al., 2023) take a compiler-centric approach. Unlike these works,
 102 TILELANG is a tile-level programmable language that automates layout and low-level configuration
 103 while giving users fine-grained control. Its flexible tile programming abstraction can help researchers
 104 obtain kernels for a broad range of AI operations, and enable advanced optimizations like software
 105 pipelining (Cheng et al., 2025) and warp specialization (Huang et al., 2023).

106 **Cost modeling.** TANGRAM (Gao et al., 2019) optimizes dataflow across scheduling layers, along
 107 with a performance modeling tool extended by SET (Cai et al., 2023) with Resource Allocation Trees.

108 KPerfIR (Guan et al., 2025) adds instrumentation for profiling and pipeline reordering in Triton.
 109 ML-based predictors like Path Forward (Li et al., 2023) and NEUSIGHT (Lee et al., 2025) also exist.
 110 In contrast, TILELANG’s tile-level analytical cost model uniquely captures both computation and data
 111 movement at tile granularity, supporting fusion-aware scheduling with high accuracy and usability.
 112

113 An extended discussion of related work is provided in Appendix A, covering classic tensor-level
 114 IRs (e.g., XLA (Google, 2019)), polyhedral compilation (Griebel et al., 1998; Zhao et al., 2021),
 115 loop-scheduling systems such as Halide (Ragan-Kelley et al., 2013), the TVM stack (Chen et al.,
 116 2018), CUTLASS (NVIDIA, 2019), and [TaichiLang](#) (Hu et al., 2019; 2020; 2021).
 117

3 PROGRAMMING MODEL

3.1 TILE LANGUAGE

121 **Tile declarations.** TILELANG elevates a *tile*—a hyper-rectangular slice of a tensor—to a first-class
 122 citizen. A tile may be owned by a warp, a thread block, or any programmer-defined parallel unit, and
 123 can be reshaped or re-partitioned at compile time. In the FlashMLA kernel, the global matrices are
 124 consumed in tiles whose extents are parameterized by `block_H`, `block_N`, and related symbolic
 125 sizes. The `T.Kernel` structure establishes the kernel’s launch configuration (e.g. `bx`, `by`, and the
 126 thread count), enabling both index derivation for each thread block and subsequent compiler analyses
 127 such as memory-access coalescing and loop tiling.

128 **Tile placement.** A distinguishing feature of TILELANG is the ability to map every tile buffer
 129 to a concrete level of the target accelerator’s memory hierarchy via user-visible intrinsics, rather
 130 than relying on opaque compiler heuristics. `T.alloc_shared` reserves storage in low-latency,
 131 software-managed shared memory on NVIDIA GPUs (or an architecturally analogous space on other
 132 devices). `T.alloc_fragment` places accumulator tiles in the register file. Although registers
 133 are scarcer than shared memory, their single-cycle latency is indispensable for performance-critical
 134 reductions. During compilation, a layout-inference pass distributes these register tiles across threads
 135 while respecting register-pressure constraints and bank conflicts.

136 **Tile operators and schedulable primitives.** Table 1 in Appendix C showcases the representative sub-
 137 set of core building blocks that orchestrate computation and movement among tiles. Fundamental op-
 138 erators (`T.copy`, `T.gemm`, `T.reduce`) act on tile operands directly, allowing the programmer to ex-
 139 press dense linear algebra, pointwise transforms, and reductions without resorting to scalarized loops.
 140 Orthogonal *scheduling primitives* expose fine-grained control over parallelism (`T.Parallel`),
 141 pipelining (`T.Pipelined`), and memory layout (`T.annotate_layout`, `T.use_swizzle`).
 142

3.2 A FLASH MULTI-HEAD LATENT ATTENTION EXAMPLE

144 By fusing high-level expressiveness with
 145 architecture-aware orchestration, TILELANG
 146 succinctly captures sophisticated AI algorithms
 147 such as FlashMLA (Liu et al., 2024) while
 148 fully harnessing the performance envelope
 149 of modern GPU architectures. Figure 2
 150 illustrates TILELANG’s developer–compiler
 151 co-optimization model: the *developer*
 152 specifies key decisions—such as tile config-
 153 uration, launch grid (`block_H`, `block_N`),
 154 buffer placement (`T.alloc_shared`,
 155 `T.alloc_fragment`), swizzled layouts
 156 (`T.annotate_layout`), and warp-level
 157 collaboration (`T.Parallel`). The *compiler*
 158 then infers the remaining low-level details,
 159 including latency-hiding pipelines, conflict-free
 160 memory layouts, and instruction selection
 161 for peak hardware performance. To balance
 162 flexibility with automation, TILELANG offers
 163 two developer-facing facilities. First, *tile recom-*

```

1 Decide tile configuration
2 with T.Kernel(batch, heads // min(block_H, kv_group_num), threads=280) as (bx, by):
3   ❶ Define buffer on desired memory layer
4     Q_shared = T.alloc_shared([block_H, dim], dtype)
5     S_shared = T.alloc_shared([block_H, block_N], dtype)
6     Q_pe_shared = T.alloc_shared([block_H, pe_dim], dtype)
7     acc_s = T.alloc_fragment([block_H, block_N], accum_dtype)
8     # static initialization statements
9   ❷ Define desired data layout
10   T.annotate_layout({|O_shared: T.make_swizzled_layout(O_shared)|})
11
12   T.copy(Q[bx, by * block_H:(by + 1) * block_H, :, :] Q.shared)
13   T.copy(Q.pe[bx, by * block_H:(by + 1) * block_H, :, :] Q_pe_shared)
14
15   ❸ Auto pipeline scheduling
16   for k in T.Pipelined(T.ceildiv(seqlen_kv, block_N), num_stages=2):
17     T.copy(KV[bx, k * block_N:(k + 1) * block_N, cur_kv_head, :, :] KV_shared)
18     T.copy(KV[pe, k * block_N:(k + 1) * block_N, cur_kv_head, :, :] K_pe_shared)
19     T.clear(acc_s)
20
21   ❹ Warp partitioning
22   T.gemm(Q_shared, KV_shared, acc_s, transpose_B=True,
23           policy=T.GemmWarpPolicy.FullCol)
24   T.gemm(Q_pe_shared, K_pe_shared, acc_s, transpose_B=True,
25           policy=T.GemmWarpPolicy.FullCol)
26   # skip flash operations on scaling and softmax
27   T.copy(acc_s, S_shared)
28
29   ❺ Auto utilize high performance instruction
30   T.gemm(S_shared, KV_shared, acc_o, policy=T.GemmWarpPolicy.FullCol)
31
32   for i, j in T.Parallel(block_H, dim):
33     acc_o[i, j] = logsum[i]
34   T.copy(acc_o, O_shared)
35   T.copy(O_shared, Output[bx, by * block_H:(by + 1) * block_H, :, :])

```

Figure 2: FlashMLA TILELANG kernel example

162 *mendation* (Sec. 4.2) supplies hardware-aware
 163 defaults that serve as high-quality starting points.
 164 Second, *tile inference* (Sec. 4.3) analytically
 165 propagates user-provided or recommended hints to complete the schedule and guarantee consistency.
 166 Working in concert, these facilities deliver near-optimal performance with limited manual tuning.
 167

168 3.3 TILELANG PHILOSOPHY

170 **Tile-level tradeoff.** The system adopts tiles as the central abstraction because this granularity
 171 provides a practical balance between portability and performance. TILELANG models the GPU
 172 memory hierarchy and the major compute and data-movement units, exposing tile size, memory
 173 placement, warp partitioning, memory layout, and software pipelining as tunable dimensions. This
 174 design enables hardware-aware specialization on both NVIDIA and AMD while preserving a unified
 175 programming model. Remaining tradeoffs lie below the tile level, where extremely fine-grained
 176 hardware behavior cannot be captured through a stable and portable API.

177 **Novel tile abstraction.** Unlike prior systems where a “tile” is essentially a manually managed
 178 shared-memory buffer, TILELANG treats tiles as first-class IR constructs with explicit semantics
 179 for indexing, data movement, reuse, and pipelining. This makes tile behavior compiler-visible and
 180 supports systematic analysis and transformation. Consequently, TILELANG differs not only in surface
 181 syntax but also in the underlying IR, which enables principled optimization at tile granularity.

183 4 SCHEDULING GUIDANCE AND AUTOMATION

185 4.1 TWO-STAGE FRAMEWORK

187 **Optimization space.** High-performance kernel design in TILELANG begins with a tile-level program,
 188 represented as a fused tile-level graph (FTG) capturing dataflow and tiling structure—each node
 189 represents a tile operator and each edge encodes a data dependency. By operating on this unified graph,
 190 TILELANG exposes and reasons about hardware-aware optimizations across six key dimensions: *tile*
 191 *size* (affecting shared memory and register usage), *memory placement* (selecting appropriate memory
 192 scope), *warp partitioning* (how threads collaborate and bind within a block), *memory layout* (how tile
 193 data is organized across memory levels), *software pipelining* (overlapping compute and data transfer,
 194 e.g., via TMA), and *tensorization* (mapping operations to CUDA or Tensor Cores).

195 **Tile recommendation and inference.** To efficiently explore the optimization space, TILELANG
 196 adopts a unified two-stage workflow over the FTG. In the first stage, *tile recommendation* analyzes the
 197 FTG to provide hardware-aware defaults for partially annotated operators, covering dimensions such
 198 as initial tile shapes, memory placement, and warp partitioning (Section 4.2). These recommendations
 199 shape the memory footprint, compute partitioning, and thread collaboration, providing high-quality
 200 starting points. In the second stage, leveraging the context from recommendation, *tile inference* prop-
 201 agates constraints through the FTG, automatically inferring the remaining configuration, including
 202 tile size, memory layout, software pipelining, and tensorization. It ensures consistency, compatibility,
 203 and hardware efficiency (Section 4.3). Together, these stages unify developer guidance and automated
 204 completion: recommendation narrows the design space with informed hints, while inference finalizes
 205 fully specified, hardware-efficient kernels with minimal manual effort.

206 In TileLang, the FTG defines the division of labor between developer control and system automation.
 207 Developers specify the FTG by composing tile operators; tile-level annotations such as tile sizes,
 208 memory placement, warp partitioning, and tensorization are optional. Given an FTG, TILELANG’s
 209 optimization pipeline performs tile recommendation and inference, propagating shape, layout, and
 210 memory constraints to complete missing details. This design lets developers concentrate on describ-
 211 ing computation while the system automatically finalizes and optimizes low-level configuration,
 212 supporting both fully automated and hint-guided usage.

213 **Running example.** Taking MLA as an example (Figure 2), TILELANG first performs tile recom-
 214 mendation as illustrated in Figure 3. Tile operators in the FTG expose tunable parameters—such as
 215 tile size, memory placement, and warp-partitioning strategies—serving as the user interface for these
 optimization knobs. For instance, in the first `T.gemm` operator (Figure 3 1), memory placement

216 annotations specify Q and KV tiles in shared memory, while S resides in registers. The S tile is
 217 further partitioned across columns using the “`policy=FullCol`” warp-partitioning strategy. These
 218 decisions directly shape the memory footprint and influence data access patterns across the FTG. The
 219 cost model analyzes the FTG to estimate memory traffic, guiding the search toward configurations
 220 that minimize data movement.

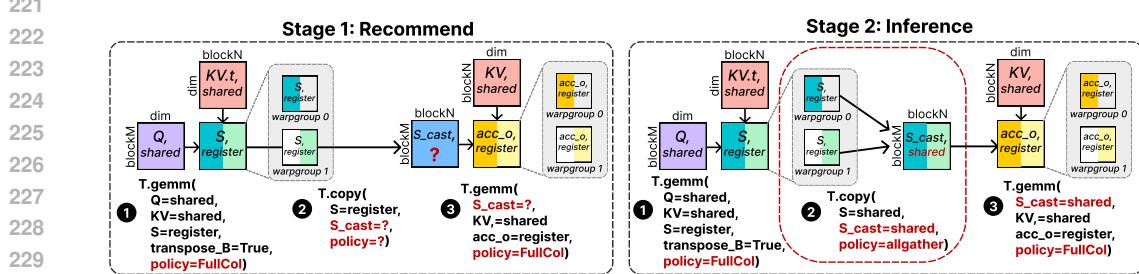
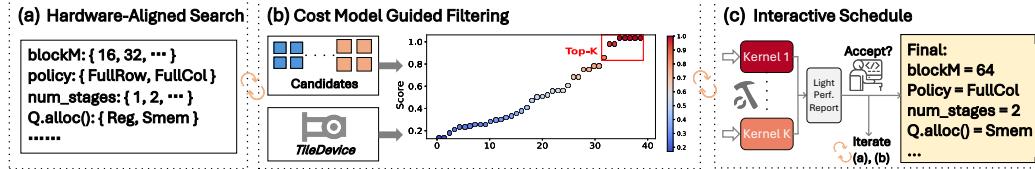


Figure 3: Two-stage workflow of optimizing MLA example.

230
 231
 232 Tile inference completes the configuration by operating over the FTG. For example, once S (output
 233 of the first `T.gemm` in Figure 3(1) and S_{cast} (input of the second `T.gemm` in Figure 3(3)) are
 234 fixed in location, shape, and partitioning in the first step, inference automatically determines the
 235 tile placement and partitioning (e.g., all-gather or scatter) of `copy` (Figure 3(2)) in the second step,
 236 ensuring consistency without manual effort. Beyond copy decisions, inference also derives memory
 237 layouts by mapping multi-dimensional indices to physical addresses, explicitly considering vector-
 238 ization, coalescing, and bank conflicts. Finally, it automates software pipelining and tensorization,
 239 ensuring that the resulting kernel configuration is efficient on the underlying hardware. TILELANG
 240 also provides platform-specific recommendations and inference (see Appendix D).

4.2 TILE RECOMMENDATION



241
 242
 243
 244 Figure 4: Tile recommendation with cost model. In (b), we show a scatter plot of candidate schedules.
 245 The x-axis orders candidates by their $\frac{1}{\text{predicted latency}}$, and the y-axis shows their normalized scores
 246 defined as $\frac{\text{latency of best candidate}}{\text{latency of current candidate}}$ which lies in the range (0,1].
 247

248
 249
 250
 251 **Roofline-based cost model.** As outlined in Figure 4, TILELANG uses a static roofline-based cost
 252 model to evaluate candidate configurations, which include tile shapes, memory placement strategies,
 253 and warp partitioning. The cost model operates directly on the fused tile graph (FTG): each FTG
 254 under a given configuration is lowered into an intermediate representation (IR), a structured, tile-
 255 oriented compute plan that explicitly encodes compute and memory access patterns per tile. From
 256 this IR, the model statically extracts two key quantities: total memory traffic at each memory level,
 257 and total floating-point operations for each compute type. These quantities are used in a roofline
 258 formulation that assumes perfect overlap between computation and memory transfers, ignoring
 259 pipeline prologues/epilogues. The execution time is estimated as: where i indexes levels of the
 260 memory hierarchy (e.g., HBM, L2, L1), and j indexes compute unit types (e.g., tensor cores, vector
 261 cuda cores, special function units (SFUs)). The term $t_{\text{intrinsic}}$ accounts for inherent overheads such as
 262 kernel launch latency and loop prologue and epilogue costs. This model provides a tight performance
 263 upper bound and allows rapid evaluation across large configuration spaces without actual execution
 264 or runtime profiling.

265
 266 Based on the cost model, TILELANG generates actionable recommendations for kernel tuning,
 267 including tile shapes, memory placement, and warp partitioning. These recommendations form an
 268 interactive baseline: developers can accept, adjust, or iteratively refine them across multiple rounds.

This human-in-the-loop workflow balances automation with expert insight, slashing tuning effort while preserving full design control.

Tile size. TILELANG presents a ranked shortlist of tile shapes that are multiples of the device’s native tensor-core fragments and respect register and shared-memory limits. Each candidate shows predicted arithmetic intensity, memory traffic, and roofline utilisation. Developers can accept the top choice, pin alternatives for later benchmarking, or adjust dimensions manually.

Memory placement. Given a chosen tile shape, TILELANG enumerates legal bindings of operands and temporaries to registers or shared memory, flagging options that exceed capacity. Each binding includes estimated pipeline stalls and effective bandwidth, letting developers quickly explore trade-offs and commit or refine placements.

Warp partition. To ensure sufficient thread-level parallelism, TILELANG proposes warp partitions that evenly cover the output tile and match the SM topology. With predicted occupancy and compute–memory overlap, developers can select, benchmark, or override, retaining full control while benefiting from data-driven guidance.

4.3 TILE INFERENCE

Layout inference. While memory placement and computation partitioning in Section 4.2 decide where tensors reside and how computation is split, layout inference determines how multi-dimensional indices are converted into physical memory addresses—taking into account vectorization, memory coalescing, and bank conflict avoidance. In other words, layout is not about which memory scope is used, but how data is accessed within that scope. Once placement and partitioning are fixed, the system can then infer an appropriate layout to ensure efficient low-level memory access.

TILELANG supports high-level indexing into multi-dimensional arrays (e.g., $A[i, k]$), which is eventually lowered to physical memory addresses through a hierarchy of abstractions. At the physical level, layouts are modeled as linear address expressions of the form $\sum_i y_i s_i$, where y_i is the index along dimension i , and s_i is its stride. To capture such mappings, TILELANG introduces a composable **Layout** algebra based on *IterVar*—a loop iterator that carries range and stride information. This allows layout transformations (e.g., transposes) to be expressed as algebraic mappings, such as **lambda** $i, j: (j, i)$. Formally, a layout becomes a function $f : \mathbb{K}^n \rightarrow \mathbb{K}^m$, converting high-level indices into memory addresses. Additionally, TILELANG defines **Fragment** layouts—a specialized extension where $f : \mathbb{K}^n \rightarrow \mathbb{K}^2$, mapping each index to a thread’s register ID and its local offset. This enables precise modeling of intra-thread register allocation. Although a buffer of size N theoretically allows $O(N!)$ memory layouts, the set of feasible layouts is significantly constrained by hardware. Global memory prefers coalesced access, shared memory requires bank conflict avoidance, and Tensor Core instructions impose strict layout requirements. To explore these constraints, TILELANG employs a greedy strategy that derives valid layouts by enforcing layout rules on selected tile operators.

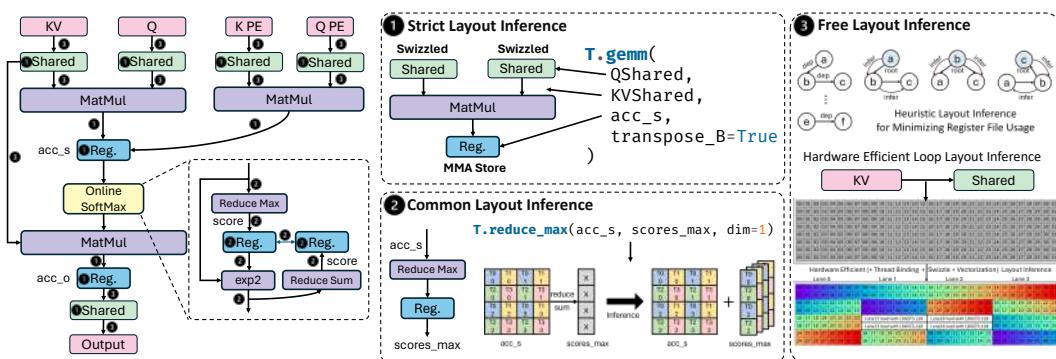


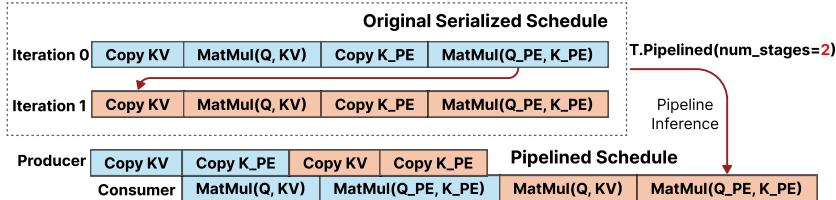
Figure 5: Layout Inference mechanism in TILELANG

We propose a hierarchical layout inference algorithm that operates over an FTG. As illustrated in Figure 5, FlashMLA can be represented as a FTG, where nodes are tile-level operators (e.g., `matmul`,

324 softmax) and edges encode data dependencies. The graph captures how Q, K, and V tiles are
 325 loaded into shared memory, attention scores computed and normalized in registers, and final outputs
 326 written back. This structure makes memory movement and parallelism explicit, enabling layout
 327 inference and efficient scheduling.

328 Our goal is to synthesize memory layouts that optimize low-level execution efficiency while preserving
 329 high-level tensor semantics. **The inference process is modeled as a constraint propagation algorithm**
 330 **(Algorithm 1 in Appendix E)** that **iteratively traverses the FTG and incrementally refines the layout**
 331 **mapping \mathcal{L} until convergence**. As illustrated in Figure 5, the algorithm integrates three complementary
 332 inference strategies: (1) Strict Layout Inference (Fig. 5①) enforces operator-specific constraints for
 333 hardware-sensitive primitives such as tensor core GEMM, including swizzled shared memory layouts
 334 and MMA-aligned register allocations; (2) Common Layout Inference (Fig. 5②) propagates layout
 335 decisions through structurally aligned operators (e.g., reductions), ensuring consistent thread bindings
 336 and register reuse; and (3) Free Layout Inference (Fig. 5③) handles the remaining unconstrained
 337 layouts by partitioning them into subgraphs via connected component analysis. For each subgraph,
 338 the partitioning scheme with the lowest register usage is selected. This step also determines the loop
 339 layout using the hardware cost model, which specifies thread binding and vectorization length to
 340 maximize memory coalescing and minimize bank conflicts. This unified inference pipeline supports
 341 composable, performance-portable layout generation and seamlessly bridges high-level loop indexing
 342 with low-level memory organization.

343 **Pipeline inference.** TILELANG automatically infers a pipelined schedule from a sequential program.
 344 As shown in Figure 6 (a), operations like `copy` and `gemm` are overlapped to increase parallelism.
 345 The system analyzes dependencies in the FTG and generates a structured pipeline that preserves
 346 execution correctness, exposing only a single `num_stages` parameter to users. Additionally,
 347 TILELANG applies Warp Specialization to fully exploit asynchronous copy instructions on Hopper
 348 GPUs, inserting synchronization barriers where necessary to maintain correct data dependencies. **The**
 349 **detailed inference procedure is described in Appendix F.**



350
 351
 352
 353
 354
 355
 356
 357
 358 Figure 6: Pipeline Inference mechanism in TILELANG
 359

360 **Instruction inference.** In TILELANG, while low-level hardware instructions such as `dp4a` or `mma`
 361 can be manually invoked via source injection or inline PTX (NVIDIA, 2021), choosing the most
 362 appropriate instruction based on input shapes and data types can be challenging. To address this,
 363 TILELANG integrates with high-level Tile Libraries like NVIDIA’s `cute` (NVIDIA, 2019) and
 364 AMD’s `ck` (AMD, 2025), which abstract hardware-specific details and automatically choose efficient
 365 instructions based on input configurations. These libraries expose standardized tile-based APIs
 366 (e.g., `t1::gemm_ss`), and TILELANG supports their invocation via a unified `T.call_extern`
 367 interface, simplifying development while ensuring performance portability.

368 5 EVALUATION

371 TILELANG is realised as a Pythonic DSL whose compiler lowers high-level tile programs to hardware-
 372 specialized kernels through a modular IR and code-generation pipeline. **TileLang is implemented on**
 373 **top of the TVM backend, but our main contributions sit above TVM.** TILELANG provides the tile
 374 abstraction, the FTG IR, and its own optimization passes, while TVM supplies the low-level code
 375 generation backend. As illustrated in Fig. 7, TILELANG adopts a five-stage compilation workflow: (1)
 376 tile-level code is written in a Python-based DSL; (2) the compiler translates the AST into the TileLang
 377 AST; (3) a FTG is constructed from the TensorIR; (4) a series of optimization passes in TileLang and
 TVM is applied; and (5) the optimized IR is finally lowered to CUDA, or other backends.

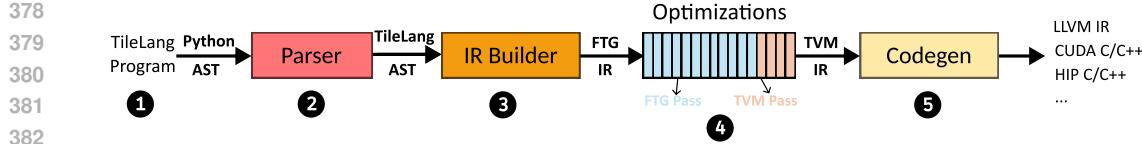


Figure 7: Overall TILELANG workflow.

385 5.1 EXPERIMENTAL SETUP

387 **Hardware platforms.** We assess the performance of TILELANG on two leading GPU architectures:
 388 NVIDIA and AMD, which dominate contemporary accelerator ecosystems. Our evaluation employs
 389 state-of-the-art hardware, including the NVIDIA H100 (80GB) (NVIDIA, 2023) and the AMD
 390 Instinct MI300X (192GB) (AMD, 2023). The NVIDIA H100 leverages CUDA 12.8, while the
 391 AMD MI300X utilizes ROCm 6.2.0. Both GPUs are benchmarked under the Ubuntu 20.04 operating
 392 system to ensure consistency in environmental configurations.

393 **AI kernels.** To evaluate system performance, we analyze nine representative operators: (1) GEMM,
 394 (2) fused dequantized GEMM ($W_{\text{INT4}}A_{\text{FP16}}$), (3) Attention, (4) Multi-Head Latent Attention,
 395 (5) Block Sparse Attention, (6) 2D Convolution, (7) Chunk Gated Delta Net, (8) Vertical Slash Sparse
 396 Attention, and (9) Attention Sink. Shape configurations are provided in Appendix N.

397 **Baselines.** Our comparative analysis considers the following baselines: (1) PyTorch Inductor—`torch.matmul` for GEMM, SDPA (PyTorch, 2023) for attention, and other operators com-
 398 piled via Inductor; (2) Triton implementations, including GemLite (Mobius ML, 2024) and MLA
 399 from SGLang (Zheng et al., 2024); (3) ThunderKittens (TK) (Spector et al., 2025)—a template-
 400 based framework for high-performance AI kernels on NVIDIA GPUs; and (4) Highly optimized
 401 libraries, including CUTLASS (NVIDIA, 2019) and Composable Kernel (AMD, 2025) for GEMM,
 402 Marlin (Frantar et al., 2025) for dequantized GEMM, FlashAttention-V3 (Dao, 2023) for MHA,
 403 AITER (AMD, 2025) for MLA, and Block Sparse Attention (Guo et al., 2024) for sparse attention.

404 We evaluate kernel performance versus code complexity (Section 5.2) and present ablation results
 405 (Section 5.3), with cost model and tuning time analyses in Appendices G and H.

408 5.2 KERNEL PERFORMANCE

409 **Matrix Multiplication.** TILELANG achieves high performance with low code complexity across
 410 diverse GEMM configurations, demonstrating 1.18–1.40× speedup over PyTorch on NVIDIA
 411 H100, while maintaining competitive performance (0.94–1.05×) on AMD MI300X. It also de-
 412 liveres 1.08–1.43× speedup over Triton with minimal kernel code, enabled by automated inference
 413 that abstracts low-level hardware details such as TMA and pipeline scheduling. Compared with TK,
 414 TILELANG achieves 0.99–1.11× speedups while reducing code complexity by 77%. Its cost-model
 415 guidance and automated tile inference eliminate manual tuning. TK depends on curated CUDA
 416 templates, limiting it to NVIDIA GPUs, whereas TILELANG supports multiple hardware backends.

417 **Low-Bit Matmul.** For $W_{\text{INT4}}A_{\text{FP16}}$ GEMM, TILELANG achieves 1.35–3.81× speedups over PyTorch
 418 and up to 1.55× over Triton on H100, while outperforming the specialized Marlin kernel with far
 419 simpler code. On MI300X, it delivers on average 0.96 × over Triton. These gains arise because
 420 TILELANG exposes low-level memory, dequantization, and layout controls that Triton hides.

422 **Convolution.** On H100, TILELANG achieves 1.24–1.79× and 1.10–1.97× speedups over PyTorch
 423 and Triton, respectively, with reduced code complexity. These gains come from its instruction
 424 inference mechanism, which maps data movement efficiently to TMA `im2col`. On MI300X, the
 425 improvements are even larger, reaching 1.29–6.80× over PyTorch and 1.02–3.10× over Triton.

426 **Flash Attention.** TILELANG achieves efficient attention computation with concise code across
 427 sequence lengths. On H100 and MI300X, it delivers 1.08–1.58× and 1.22–1.37× speedups over
 428 Triton, while matching the performance of FlashAttention-V3 (0.98× and 0.96× on average). These
 429 results stem from TILELANG’s ability to infer and apply platform-specific partitioning and pipelining
 430 strategies that exploit specialized compute units. TILELANG achieves up to 1.10× speedup over TK
 431 while significantly reducing code complexity (from 185 lines to 66), highlighting its programmability.
 By combining tile-level guidance with automated inference, TILELANG streamlines kernel

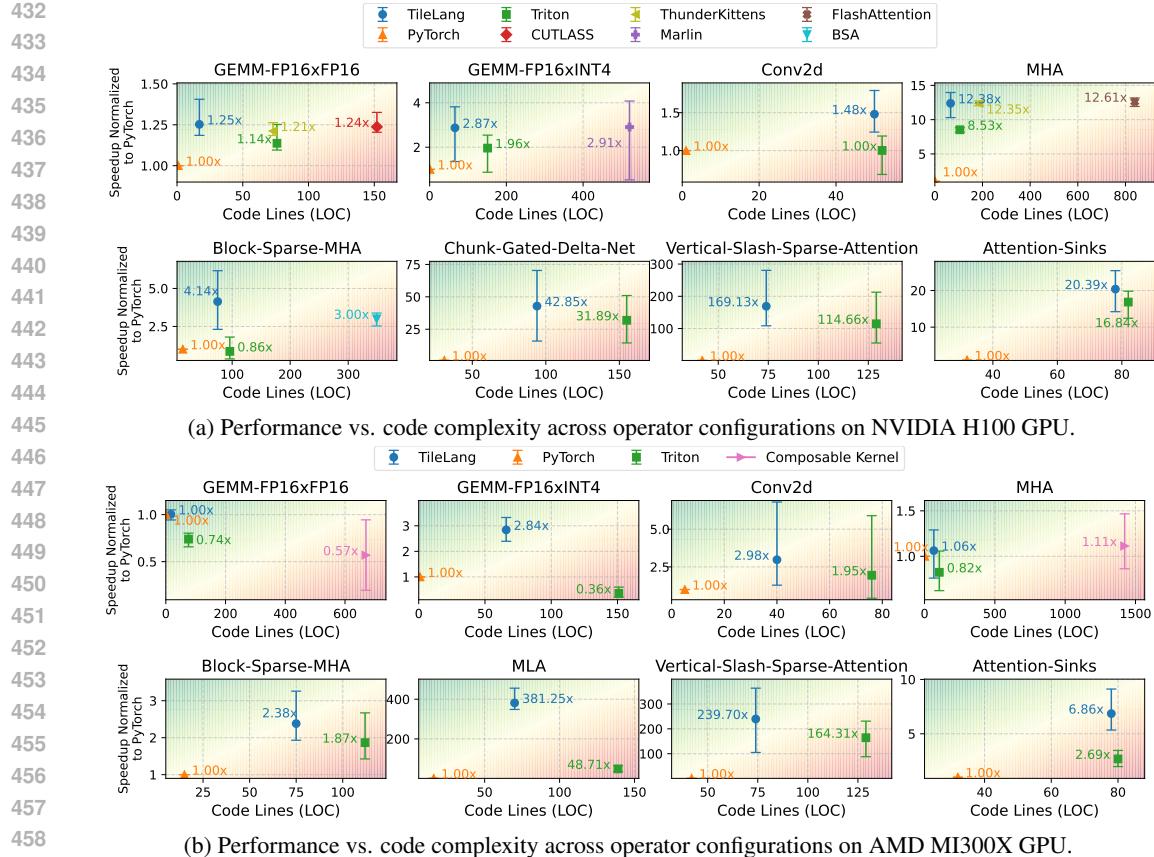


Figure 8: Performance and code complexity on an NVIDIA H100 GPU and AMD MI300X GPU. The y-axis denotes the speedup relative to PyTorch, while the x-axis indicates lines of code (LOC). Ideal solutions appear toward the top-left corner.

development. This is particularly valuable for complex attention operators, where TK often requires extensive manual tuning of tiling, warp partitioning, layout, and pipelines.

Flash MLA. As shown in Figure 1, TILELANG achieves 4.06–10.59× speedups over Triton on H100, with substantially reduced code complexity. It matches the latency of the specialized FlashMLA kernel while reducing code complexity by 6.86×. On MI300X, TILELANG delivers 5.64–12.97× gains over Triton and slightly outperforms the hand-tuned ROCm library AITER (1.05×). These improvements arise from warp specialization and automated TMA mapping.

Block Sparse Attention. TILELANG achieves acceleration of 3.42–7.87× and 1.22–1.37× over Triton with less code on H100 and MI300X, respectively. On H100, it matches BlockSparse (BSA) latency (0.91–1.82×) while greatly reducing complexity. Implementing block-sparse MHA requires only adding two lines to the standard MHA code (Appendix O.5).

Chunk Gated Delta Net. On H100, TILELANG achieves 15.88–70.35× speedups over PyTorch by fusing complex operations into a single kernel. Compared to Triton, it attains 1.10–1.45× speedups with 39% fewer lines of code. These gains come from automated tile recommendation and inference, which optimize memory placement and partitioning for efficient hardware utilization.

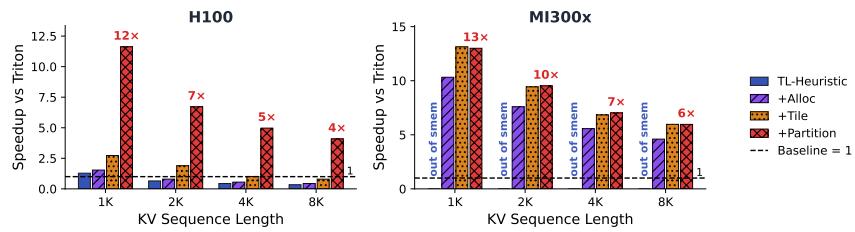
Vertical Slash Sparse Attention. TILELANG delivers 108.55–280.41× and 105.01–363.53× speedups over PyTorch on H100 and MI300X, largely by fusing the sparse attention operation into a single efficient kernel. Compared to Triton, it achieves 1.16–1.97× and 1.19–1.60× speedups on H100 and MI300X, respectively, while cutting the code size by roughly half.

Attention Sinks. For attention with the sinks mechanism, TILELANG achieves 14.21–25.57× and 5.35–9.11× speedups over PyTorch on H100 and MI300X, respectively, enabled by TILELANG’s

486 FTG-based fusion into a single optimized kernel. Against Triton, it reaches 1.13–1.30 \times on H100
 487 and 2.32–2.69 \times on MI300X. The attention-sink variant differs only slightly from standard MHA,
 488 showing that TILELANG readily supports diverse attention patterns with minimal effort.
 489

490 5.3 ABLATION STUDIES

492 To help clarify what contributes to the speedups over the baseline, we perform an ablation study
 493 on FlashMLA as a representative example. Starting from a TILELANG version that uses manually
 494 crafted scheduling heuristics (TL-Heuristic), we progressively enable three components: (i) cost-
 495 model-guided tiling (+Tile), which improves the compute–memory ratio and cache use; (ii) cost-
 496 model-guided memory placement (+Alloc), which chooses efficient buffer locations and reduces
 497 register spilling; and (iii) warp partitioning (+Partition), which improves intra-warp load balance.
 498 Performance is measured at each stage relative to the [Triton](#) baseline.



500 501 502 503 504 505 506 507 Figure 9: Ablation study for FlashMLA on both H100 and MI300X GPUs.

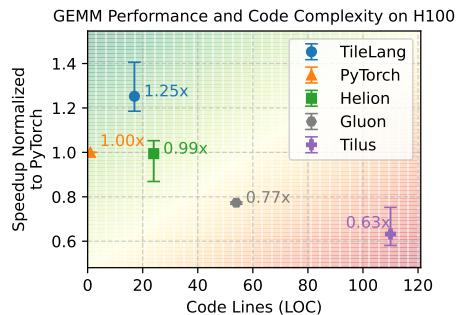
508 As shown in Figure 9, each of the evaluated optimizations provides measurable performance gains
 509 over the [Triton baseline](#), validating their effectiveness. Our analysis also highlights architecture-
 510 specific behaviors (illustrated in Appendix D). On H100, tiling (+Tile) yields a modest speedup of
 511 1.31 \times . Building on this, warp partitioning (+Partition) provides the dominant contribution, delivering
 512 an additional 4.34 \times improvement. On MI300X, allocation placement (+Alloc) serves as the primary
 513 optimization, achieving a 6.56 \times speedup. When further combined with tiling (+Tile), the overall
 514 gain increases by another 1.75 \times improvement.

515 5.4 COMPARE WITH MORE RECENT DSLS

517 We further compare TILELANG with recent
 518 DSLs including Helion, Gluon, and Tilus. Our
 519 evaluation covers the latest Hopper-supported
 520 implementations across GEMM, MHA, and
 521 Mamba-chunk-scan. Figure 10 reports the
 522 GEMM results, with additional results in Ap-
 523 pendix L. On GEMM, TILELANG achieves
 524 1.15 \sim 1.62 \times , 1.52 \sim 1.83 \times , and 1.87 \sim
 525 2.12 \times speedups over Helion, Gluon, and Tilus,
 526 respectively, while using fewer lines of code.
 527 These improvements largely stem from TILE-
 528 LANG’s compiler-visible tile abstraction, which
 529 enables more structured optimization than ex-
 530 isting DSLs. Note that Tilus is not yet fully op-
 531 timized for Hopper features such as WGMMA
 532 and TMA, and on Ampere/Ada TILELANG is
 533 still slightly better (see Appendix L).

534 6 CONCLUSION

535 TILELANG offers a controllable tile-level programming model with graph-based optimizations
 536 via tile recommendation and inference. By combining automated configuration with fine-grained
 537 developer control, it streamlines kernel development and delivers significant speedups. It enables rapid
 538 experimentation with emerging AI algorithms, such as custom attention, sparsity, and quantization.
 539 TILELANG also lowers barriers for systems-aware research across diverse hardware platforms.



536 537 538 539 Figure 10: Performance and code-size comparison
 540 of GEMM kernels on H100 across recent systems.

540 7 REPRODUCIBILITY STATEMENT
541

542 We provide a detailed description of our experimental setup in Section 5.1. Operator shapes used in
543 our benchmarks are drawn from widely adopted, real-world AI models (e.g., GPT-OSS, DeepSeek
544 V3, Qwen3-Next). A list of these operator configurations is included in Appendix N, and the
545 corresponding TILELANG code of kernels used in evaluation is provided in Appendix O. The system
546 implementation and scripts for reproducing our experiments will be made publicly available after the
547 review process, ensuring full reproducibility while maintaining anonymity.

548
549 REFERENCES
550

551 Joshua Ainslie, James Lee-Thorp, Michiel De Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit
552 Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints.
553 *arXiv preprint arXiv:2305.13245*, 2023.

554 AMD. Amd cdna™ 3 architecture. Technical report, AMD, 2023. URL <https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-3-white-paper.pdf>.

557 AMD. White paper: Introducing amd cdna™ 3 architecture, 2024. URL <https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-3-white-paper.pdf>.

561 AMD. ROCm aiter, 2025. <https://github.com/ROCm/aiter>.

562 AMD. AMD Composable Kernel, 2025. https://github.com/ROCm/composable_kernel.

565 Jingwei Cai, Yuchen Wei, Zuotong Wu, Sen Peng, and Kaisheng Ma. Inter-layer scheduling space
566 definition and exploration for tiled accelerators. In *Proceedings of the 50th Annual International
567 Symposium on Computer Architecture*, pp. 1–17, 2023.

568 Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan
569 Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End}
570 optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design
571 and Implementation (OSDI 18)*, pp. 578–594, 2018.

572 Yu Cheng, Lei Wang, Yining Shi, Yuqing Xia, Lingxiao Ma, Jilong Xue, Yang Wang, Zhiwen
573 Mo, Feiyang Chen, Fan Yang, et al. Pipethreader: Software-defined pipelining for efficient dnn
574 execution. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI
575 25)*, 2025.

577 Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv
578 preprint arXiv:2307.08691*, 2023.

579 Tri Dao and Albert Gu. Transformers are ssms: Generalized models and efficient algorithms through
580 structured state space duality. *arXiv preprint arXiv:2405.21060*, 2024.

582 DeepSeek. FlashMLA, 2025. <https://github.com/deepseek-ai/FlashMLA>.

583 Yaoyao Ding, Bohan Hou, Xiao Zhang, Allan Lin, Tianqi Chen, Cody Yu Hao, Yida Wang, and Gen-
584 nady Pekhimenko. Tilus: A tile-level gppu programming language for low-precision computation.
585 *arXiv preprint arXiv:2504.12984*, 2025.

587 Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin
588 Zheng, Cody Hao Yu, Yong Yu, et al. Tensorir: An abstraction for automatic tensorized program
589 optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support
590 for Programming Languages and Operating Systems, Volume 2*, pp. 804–817, 2023.

591 Elias Frantar, Roberto L Castro, Jiale Chen, Torsten Hoefer, and Dan Alistarh. Marlin: Mixed-
592 precision auto-regressive parallel inference on large language models. In *Proceedings of the 30th
593 ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pp.
239–251, 2025.

594 Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. Tangram: Optimized
 595 coarse-grained dataflow for scalable nn accelerators. In *Proceedings of the Twenty-Fourth Interna-*
 596 *tional Conference on Architectural Support for Programming Languages and Operating Systems*,
 597 pp. 807–820, 2019.

598 William F Godoy, Tatiana Melnichenko, Pedro Valero-Lara, Wael Elwasif, Philip Fackler, Rafael Fer-
 599 reira Da Silva, Keita Teranishi, and Jeffrey S Vetter. Mojo: Mlir-based performance-portable hpc
 600 science kernels on gpus for the python ecosystem. *arXiv preprint arXiv:2509.21039*, 2025.

602 Google. XLA, 2019. <https://www.tensorflow.org/xla>.

603 Martin Griebl, Christian Lengauer, and Sabine Wetzel. Code generation in the polytope model. In
 604 *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques*
 605 (*Cat. No. 98EX192*), pp. 106–111. IEEE, 1998.

607 Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv*
 608 *preprint arXiv:2312.00752*, 2023.

610 Yue Guan, Yuanwei Fang, Keren Zhou, Corbin Robeck, Manman Ren, Zhongkai Yu, Yufei Ding, and
 611 Adnan Aziz. Kperfir: Towards a open and compiler-centric ecosystem for gpu kernel performance
 612 tooling on modern ai workloads. 2025.

613 Junxian Guo, Haotian Tang, Shang Yang, Zhekai Zhang, Zhijian Liu, and Song Han. Block Sparse At-
 614 tention. <https://github.com/mit-han-lab/Block-Sparse-Attention>, 2024.

616 Bastian Hagedorn, Bin Fan, Hanfeng Chen, Cris Cecka, Michael Garland, and Vinod Grover.
 617 Graphene: An ir for optimized tensor computations on gpus. In *Proceedings of the 28th ACM*
 618 *International Conference on Architectural Support for Programming Languages and Operating*
 619 *Systems, Volume 3*, pp. 302–313, 2023.

620 Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. Taichi: a
 621 language for high-performance computation on spatially sparse data structures. *ACM Transactions*
 622 *on Graphics (TOG)*, 38(6):201, 2019.

624 Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo
 625 Durand. Difftaichi: Differentiable programming for physical simulation. *ICLR*, 2020.

626 Yuanming Hu, Jiafeng Liu, Xuanda Yang, Mingkuan Xu, Ye Kuang, Weiwei Xu, Qiang Dai,
 627 William T. Freeman, and Frédo Durand. Quantaichi: A compiler for quantized simulations.
 628 *ACM Transactions on Graphics (TOG)*, 40(4), 2021.

630 Guyue Huang, Yang Bai, Liu Liu, Yuke Wang, Bei Yu, Yufei Ding, and Yuan Xie. Alcop: Automatic
 631 load-compute pipelining in deep learning compiler for ai-gpus. *Proceedings of Machine Learning*
 632 *and Systems*, 5:680–694, 2023.

633 Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman
 634 Parashar. Maestro: A data-centric approach to understand reuse, performance, and hardware
 635 cost of dnn mappings. *IEEE micro*, 40(3):20–29, 2020.

637 Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River
 638 Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler
 639 infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium*
 640 *on Code Generation and Optimization (CGO)*, pp. 2–14. IEEE, 2021. <https://mlir.llvm.org/>.

642 Seonho Lee, Amar Phanishayee, and Divya Mahajan. Forecasting gpu performance for deep learning
 643 training and inference. In *Proceedings of the 30th ACM International Conference on Architectural*
 644 *Support for Programming Languages and Operating Systems, Volume 1*, pp. 493–508, 2025.

646 Ying Li, Yifan Sun, and Adwait Jog. Path forward beyond simulators: Fast and accurate gpu execution
 647 time prediction for dnn workloads. In *Proceedings of the 56th Annual IEEE/ACM International*
 648 *Symposium on Microarchitecture*, pp. 380–394, 2023.

648 Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao,
 649 Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint*
 650 *arXiv:2412.19437*, 2024.

651 652 Mobius ML. Gemlite: Lightweight generative machine learning engine. <https://github.com/mobiusml/gemlite>, 2024. Accessed: [Insert Access Date].

653 654 NVIDIA. NVIDIA cutlass, 2019. <https://github.com/NVIDIA/cutlass>.

655 656 NVIDIA. PTX ISA, 2021. <https://docs.nvidia.com/cuda/parallel-thread-execution/>.

657 658 NVIDIA. Nvidia h100 tensor core gpu architecture. Technical report, NVIDIA Corporation, 2023.
 659 URL <https://resources.nvidia.com/en-us-tensor-core>.

660 661 OpenAI. Gluon, 2025. <https://github.com/triton-lang/triton/tree/main/python/examples/gluon>.

662 663 Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag
 664 Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer.
 665 Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE international
 666 symposium on performance analysis of systems and software (ISPASS)*, pp. 304–315. IEEE, 2019.

667 668 Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor
 669 Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style,
 670 high-performance deep learning library. *Advances in neural information processing systems*, 32,
 2019. <https://pytorch.org/>.

671 672 PyTorch. torch.nn.functional.scaled_dot_product_attention, 2023. URL https://docs.pytorch.org/docs/stable/generated/torch.nn.functional.scaled_dot_product_attention.html.

673 674 PyTorch. Helion: A High-Level DSL for Performant and Portable ML Kernels, 2025. <https://github.com/pytorch/helion>.

675 676 Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman
 677 Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recom-
 678 putation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference
 679 on Programming Language Design and Implementation, PLDI '13*, pp. 519–530, New York,
 680 NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462176. URL
 681 <http://doi.acm.org/10.1145/2491956.2462176>.

682 683 Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao.
 684 Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *arXiv preprint*
 685 *arXiv:2407.08608*, 2024.

686 687 Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang,
 688 and Lidong Zhou. Welder: Scheduling deep learning memory access via tile-graph. In *17th
 689 USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pp. 701–718,
 2023.

690 691 Benjamin F Spector, Simran Arora, Aaryan Singhal, Daniel Y Fu, and Christopher Ré. Thunderkittens:
 692 Simple, fast, and adorable ai kernels. In *Proceedings of the 13th International Conference on Learn-
 693 ing Representations*, 2025. URL <https://openreview.net/forum?id=0fJfVOSUra>.
 694 <https://github.com/HazyResearch/ThunderKittens>.

695 696 Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and
 697 Furu Wei. Retentive network: A successor to transformer for large language models. *arXiv preprint*
 698 *arXiv:2307.08621*, 2023.

699 700 Philippe Tillet, H. T. Kung, and David Cox. *Triton: An Intermediate Language and Compiler for
 701 Tiled Neural Network Computations*, pp. 10–19. Association for Computing Machinery, New York,
 NY, USA, 2019. ISBN 9781450367196. URL <https://doi.org/10.1145/3315508.3329973>.

702 Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito,
 703 William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions:
 704 Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730,
 705 2018. URL <http://arxiv.org/abs/1802.04730>.

706 Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz
 707 Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing
 708 systems*, 30, 2017.

710 Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li,
 711 Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: Optimizing tensor programs with par-
 712 tially equivalent transformations and automated corrections. In *15th USENIX Symposium on
 713 Operating Systems Design and Implementation (OSDI 21)*, pp. 37–54. USENIX Association,
 714 July 2021. ISBN 978-1-939133-22-9. URL [https://www.usenix.org/conference/
 715 osdi21/presentation/wang](https://www.usenix.org/conference/osdi21/presentation/wang).

716 Lei Wang, Lingxiao Ma, Shijie Cao, Quanlu Zhang, Jilong Xue, Yining Shi, Ningxin Zheng,
 717 Ziming Miao, Fan Yang, Ting Cao, et al. Ladder: Enabling efficient {Low-Precision} deep
 718 learning computing through hardware-aware tensor transformation. In *18th USENIX Symposium
 719 on Operating Systems Design and Implementation (OSDI 24)*, pp. 307–323, 2024.

720 Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Kit Ao, Praveen Velliengiri, Xupeng
 721 Miao, Oded Padon, and Zhihao Jia. Mirage: A multi-level superoptimizer for tensor programs. In
 722 *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, Boston,
 723 MA, July 2025. USENIX Association.

724 Rohan Yadav, Michael Garland, Alex Aiken, and Michael Bauer. Task-based tensor computations on
 725 modern gpus. *Proceedings of the ACM on Programming Languages*, 9(PLDI):396–420, 2025.

727 Songlin Yang, Bailin Wang, Yu Zhang, Yikang Shen, and Yoon Kim. Parallelizing linear transformers
 728 with the delta rule over sequence length. *Advances in neural information processing systems*, 37:
 729 115491–115522, 2024.

731 Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun
 732 Cheng, Zheng Li, Peng Di, Kun Zhang, and Xuefeng Jin. Akg: Automatic kernel generation
 733 for neural processing units using polyhedral transformations. In *Proceedings of the 42nd ACM
 734 SIGPLAN International Conference on Programming Language Design and Implementation*,
 735 PLDI 2021, pp. 1233–1248, New York, NY, USA, 2021. Association for Computing Machinery.
 736 ISBN 9781450383912. doi: 10.1145/3453483.3454106. URL [https://doi.org/10.1145/
 737 3453483.3454106](https://doi.org/10.1145/3453483.3454106).

738 Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun
 739 Yang, Danyang Zhuo, Koushik Sen, et al. Ansol: Generating {High-Performance} tensor programs
 740 for deep learning. In *14th USENIX symposium on operating systems design and implementation
 741 (OSDI 20)*, pp. 863–879, 2020.

742 Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi
 743 Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of
 744 structured language model programs. *Advances in Neural Information Processing Systems*, 37:
 745 62557–62583, 2024.

746
 747
 748
 749
 750
 751
 752
 753
 754
 755

756 Our appendix is organized as follows:
 757
 758 Appendix A: Extended discussion of related work.
 759 Appendix B: Details of the MLA algorithm.
 760 Appendix C: Semantics of a partial list of TILELANG primitives.
 761 Appendix D: Platform-specific scheduling
 762 Appendix E: Layout inference algorithm of TILELANG.
 763 Appendix F: Pipeline inference algorithm of TILELANG.
 764 Appendix G: Evaluation of the cost model.
 765 Appendix H: Tuning time measurements.
 766 Appendix I: Matmul implementation difference between TVM, Triton, and TILELANG.
 767 Appendix J: Comparison of FTG IR and TVM IR.
 768 Appendix K: Comparison of FlashMLA implementations on different architectures.
 769 Appendix L: Comparison with recent DSLs.
 770 Appendix M: Commit hashes of each baseline.
 771 Appendix N: Operator shapes used in our benchmark.
 772 Appendix O: TILELANG code of kernels used in the evaluation.
 773
 774
 775
 776
 777
 778
 779
 780 **A EXTENDED DISCUSSION OF RELATED WORK**
 781
 782 **AI kernel generators** For kernel generating and optimization, Ansor (Zheng et al., 2020) builds
 783 many kernel combinations by sampling programs from a hierarchical search space. PET (Wang et al.,
 784 2021) moves forward to partially equivalent transformations and automated correction for newly
 785 discovered kernels. TensorIR (Feng et al., 2023) generates new kernels by generalizing the loop nest
 786 representation used in existing machine learning compilers. Mirage (Wu et al., 2025) proposes a
 787 uniform representation of a tensor program at each level of the latest GPU compute hierarchy to
 788 find custom kernels. AKG (Zhao et al., 2021) leverages polyhedral schedulers to perform a much
 789 wider class of transformations to automatically generate kernels on NPUs. TileLang also utilizes
 790 multi-level programming interfaces and autotuning techniques based on tile-level cost model for
 791 kernel generation and optimization.
 792
 793 **Tensor-level IRs, The polyhedral model, and Loop synthesizers.** Traditional approaches ad-
 794 dress program optimization at different abstraction levels: Tensor-level IRs (e.g., XLA (Google,
 795 2019)) lower tensor programs via pattern-matched templates (e.g., LLVM, CUDA). Polyhedral mod-
 796 els (Griebel et al., 1998) (e.g., TC (Vasilache et al., 2018)) automate affine loop transforms, mainly for
 797 DNN layers. Loop synthesizers (e.g., Halide (Ragan-Kelley et al., 2013)) generate loop nests guided
 798 by user-defined schedules. TILELANG targets a distinct programming model and control granularity.
 799 It differs fundamentally by introducing tiles as first-class programming units. It offers programmable
 800 control over fusion strategies, memory hierarchy, and parallelism. This enables developers to design
 801 fused kernels with both high performance and portability across hardware.
 802
 803 **TVM (Chen et al., 2018).** TILELANG builds upon TVM’s IR and arithmetic passes. However,
 804 unlike TVM’s schedule-driven loop generation from high-level compute definitions, TILELANG
 805 offers explicit, tile-level programmability and control over memory, fusion, and parallelism. This
 806 enables much finer-grained kernel customization beyond what TVM can achieve. For instance, TVM
 807 cannot fully express advanced algorithms like FlashAttention (FA) or Multi-Level Attention (MLA),
 808 which demand precise management of memory hierarchy and execution order—capabilities that
 809 TileLang supports.
 810
 811 **Warp Partition.** Warp Partition (WP) is a key component of TILELANG’s execution model, building
 812 directly on the tile abstraction. Given a specified tile size, WP allows further partitioning of the tile
 813 along each dimension across multiple warps. For example, consider a GEMM operation $C = A @ B$,

810 where $A \in \mathbb{R}^{M \times K}$, $B \in \mathbb{R}^{K \times N}$, and $C \in \mathbb{R}^{M \times N}$. The output tile C can be partitioned along either
 811 the M or N axis, corresponding to *full-row* or *full-column* warp-partitioning strategies, respectively.
 812 By giving users explicit control over warp partitioning, TILELANG enables fine-grained management
 813 of resources such as register usage within each warp. This, in turn, allows users to better control
 814 the performance of operations. Such flexibility is crucial for mapping computations efficiently to
 815 hardware, especially when optimizing diverse and performance-sensitive kernel workloads.

816 **CuTe library.** While both TILELANG and CuTe (NVIDIA, 2019) share this high-level goal, their
 817 underlying mechanisms differ: CuTe relies on shape/stride pairs, whereas TILELANG encodes the
 818 mappings using explicit arithmetic expressions. This arithmetic formulation offers advantages by
 819 more directly capturing index transformations and enabling more flexible, composable manipulations,
 820 allowing for clear definition and description in the DSL frontend.

821 **The roofline-guided cost model.** Several analytical modeling approaches have been proposed, such
 822 as the nested-loop-based modeling in Timeloop (Parashar et al., 2019), the data-centric representation
 823 in Maestro (Kwon et al., 2020). In contrast to these methods, our work leverages a tile-level program-
 824 ming abstraction, which naturally lends itself to a tile-centric cost model. This enables us to accurately
 825 capture both computation and data movement at the tile granularity, while maintaining simplicity and
 826 enhanced support for modeling operator fusion. This design strikes a balance between accuracy and
 827 usability, making it effective for guiding schedule selection without introducing excessive complexity.

828 **TaichiLang.** Although both Taichi (Hu et al., 2019; 2020; 2021) and TileLang target GPU workloads,
 829 they differ fundamentally in abstraction and intended use. Taichi provides a high-level, scalar-loop
 830 DSL with automatic parallelization, SNode-based data layouts, and strong autodiff support, making
 831 it well-suited for scientific computing and simulation. TileLang, by contrast, is designed for deep-
 832 learning kernels such as attention and GEMM, where peak performance requires explicit control
 833 of tile shapes, memory placement across shared/LDS and registers, multi-stage pipelining, warp
 834 specialization, and instruction selection (Tensor Core MMA or AMD MFMA). These capabilities are
 835 not directly expressible in Taichi’s fully automatic model.

837 B MLA ALGORITHM

839 Instead of storing full-sized key and value matrices, MLA projects input token embeddings into a
 840 lower-dimensional latent space using a down-projection matrix:

$$842 \mathbf{z}_t = \mathbf{x}_t \mathbf{W}_{\text{down}}.$$

843 The latent vector \mathbf{z}_t is then used to reconstruct the key and value representations:

$$844 \mathbf{k}_t = \mathbf{z}_t \mathbf{W}_{\text{up}}^K, \quad \mathbf{v}_t = \mathbf{z}_t \mathbf{W}_{\text{up}}^V.$$

845 To incorporate positional information, Rotary Positional Embedding (RoPE) is applied to the recon-
 846 structed keys and queries:

$$847 \mathbf{k}_t^{\text{rot}} = \text{RoPE}(\mathbf{k}_t).$$

848 Queries are also compressed using a similar process to reduce activation memory:

$$849 \mathbf{q}_t = \mathbf{z}_t^Q \mathbf{W}_{\text{up}}^Q.$$

850 MLA further enhances computational efficiency through a technique known as *matrix absorption*,
 851 which reorders matrix multiplications to optimize performance. This approach enables the key and
 852 value inputs to share the same latent representation \mathbf{z}_t , thereby reducing redundancy and memory
 853 usage. In the adopted configuration, MLA employs a single shared key-value (KV) head, with a head
 854 dimension of 512.

856 C PARTIAL LIST OF TILELANG PRIMITIVES

858 Table 1 illustrates the expressiveness of the TILELANG intermediate representation. To support
 859 efficient code generation across diverse hardware backends, TILELANG decouples the definition of
 860 algorithms from their optimization. The *Dataflow Centric Tile Operators* define the functional seman-
 861 tics of the workload (e.g., matrix multiplication, atomic updates), while the *Scheduling Primitives*
 862 expose critical optimization handles—such as loop pipelining (‘Pipelined’) and layout transformation
 863 (‘annotate_layout’)—to maximize hardware utilization and locality without altering the algorithmic
 864 correctness.

864 Table 1: A partial list of primitives supported by TILELANG.
865

866 Dataflow Centric Tile Operators		867 Scheduling Primitives	
868 copy	869 data movement among hierarchy memory.	870 Parallel	871 Parallelization of loop iterations over threads.
872 gemm	873 matrix multiplication on different GPUs.	874 Pipelined	875 Enables pipelining to overlap data transfers with computation.
876 reduce	877 reduction operator (e.g., sum, min, max) exploiting warp/block-level parallelism.	878 annotate_layout	879 Definition of custom memory layouts to minimize bank conflicts and optimize thread binding.
880 atomic	881 atomic operations to ensure thread-safe updates in shared or global memory.	882 use_swizzle	883 Improves L2 cache locality via swizzled access patterns.
884 Warp Specialization			
885 barrier_arrive		886 Signals the arrival at a synchronization point (mbarrier) for producer/consumer coordination.	
887 barrier_wait		888 Blocks execution until specific barrier conditions (e.g. transaction counts) are met.	

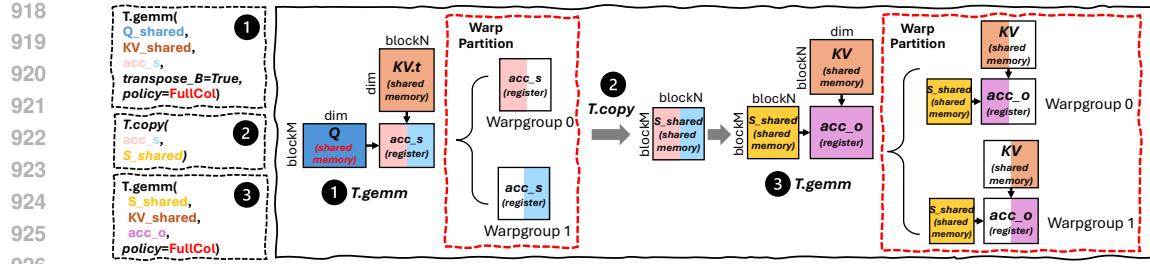
889

D PLATFORM-SPECIFIC SCHEDULING

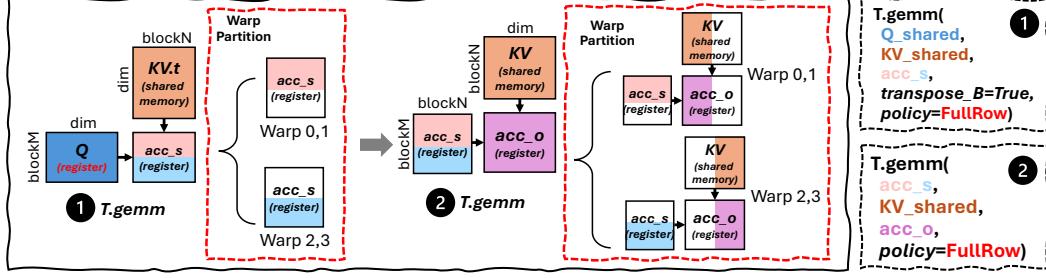
890
891
892
893 TILELANG also provides platform-specific recommendations and inference. Taking MLA as an
894 example, we illustrate how TILELANG performs tile recommendation and inference based on the
895 code shown in Figure 2.896 On the H100, each SM features 228 KiB of shared memory and a 256 KiB register file, whereas the
897 MI300X provides 64 KiB of Local Data Share (LDS) and a total of 512 KiB in registers. Given these
898 architectural differences, TILELANG first recommends different tile configurations.899 Table 2: Comparison of specifications between NVIDIA H100 SXM and AMD MI300X.
900

901 Specification	902 NVIDIA H100 SXM	903 AMD MI300X
904 Clock Frequency	905 1.83 GHz	906 2.10 GHz
907 DDR Memory Bandwidth	908 3.35 TB/s	909 5.30 TB/s
910 L2 Bandwidth	911 9.45 TB/s	912 16.63 TB/s
913 L1/Shared Memory BW	914 30.92 TB/s	915 81.72 TB/s
916 Compute Units (SMs/CUs)	917 132 SMs	918 304 CUs
919 Shared Memory per SM/CU	920 228 KiB	921 64 KiB
922 Register File per SM/CU	923 256 KiB	924 512 KiB
925 Peak FP16 Performance	926 989 TFLOPs	927 1307 TFLOPs

928
929 As shown in Figure 11, for memory placement, users may initially allocate the Q tile to shared
930 memory on the MI300X. However, this approach fails due to the limited capacity of shared memory.
931 TILELANG detects this constraint and instead recommends placing both Q and acc_s in registers. In
932 contrast, on the H100, both tiles fit comfortably in shared memory and are placed there accordingly.
933 For Software Pipelining, TILELANG disables pipelining on the MI300X to support larger tile sizes
934 and reduce register pressure, whereas on the H100, pipelining is enabled to maximize pipeline
935 overlap. Tile sizes are also adjusted accordingly to fit each platform’s resource constraints. For Warp
936 Partitioning, users may initially adopt a default policy for the two **gemm** operators, which often leads
937 to sub-optimal performance. TILELANG addresses this by analyzing the underlying hardware and
938 recommending platform-specific partitioning strategies, as illustrated in Figure 3. On the H100, both
939 **gemm** operators use the **FullCol** scheme, partitioning acc_s and acc_o vertically to match the
940 Tensor Core shape. In contrast, TILELANG applies a **FullRow** policy on the MI300X, partitioning
941 tiles horizontally.



(a) The Warp Partition schedule recommended by TILELANG for MLA on H100.



(b) The Warp Partition schedule recommended by TILELANG for MLA on MI300X.

Figure 11: Cooperative workflow between tile-recommendation and inference stages on NVIDIA H100 and AMD MI300X GPUs.

E LAYOUT INFERENCE ALGORITHM

Goal and Scope. Algorithms 1 and 2 compute a hardware-aware, globally consistent layout mapping over a fused tile graph (FTG). Starting from partially annotated buffers and operator semantics, the pass infers concrete memory layouts (global/shared/register fragments), resolves aliasing, and returns both the final layout map \mathcal{L} and the loop-level binding/predication maps (`ForMap`, `PredMap`) used by the loop-lowering routine in Algorithm 3. The inference explicitly respects coalescing, bank-conflict avoidance, and Tensor-Core-friendly register tiling, while minimizing register pressure for the remaining degrees of freedom.

Three-Phase Inference. The core procedure (`LayoutInference`) executes in three stages:

- **Phase I: Strict constraints (“STRICT”).** Each operator is visited once with `RunInferStep` at level “STRICT”. This enforces hard constraints dictated by hardware-sensitive primitives (e.g., swizzled shared-memory layouts and MMA-aligned register tiles for GEMM). All layouts fixed here are recorded into L_{strict} and treated as immutable thereafter.
- **Phase II: Common propagation (“COMMON”).** A worklist over all operators drives fixed-point propagation. When an update materializes for a buffer, its users are enqueued. This phase spreads compatible, non-rigid constraints across the FTG until convergence, ensuring consistent thread bindings and compatible address formulas across producers and consumers.
- **Phase III: Free choices with register-cost minimization (“FREE”).** Remaining unconstrained buffers are partitioned by connected components (w.r.t. uses and aliasing). For each component, the algorithm enumerates candidate “roots”: it snapshots state, seeds inference from a root with level “FREE”, and greedily extends to other members. Candidates that trigger a conflict (layout mismatch or iterator normalization errors) are discarded. Among feasible candidates, it selects the one minimizing total fragment registers via `SumFragmentRegisters`, then restores and commits the best snapshot. This realizes a lightweight, local backtracking that controls search while favoring low register pressure.

Key Subroutines. `RunInferStep` constructs the per-operator context (target, thread bounds, current L , analyzer, and any out-of-bound info) and calls `op.InferLayout(args, level)`

972
973
974
975
976
977
978
979

Algorithm 1 Hardware-Aware Layout Inference over Fused Tile Graph

```

980
981 1: procedure LayoutInference(FTG, AnnotatedLayouts, Target)
982 2: (Ops, UseList, AliasGroups, ThreadVar, ThreadBounds, BufferOOB) = CollectFromIR(FTG, Target)
983 3: L = copy(AnnotatedLayouts)
984 4: L_strict = empty_map()
985 5: for i = 0 to |Ops| - 1 do ▷ Phase I: Strict constraints
986 6:   RunInferStep(i, "STRICT", false, L, L_strict)
987 7: for all (buf, lay) in L do
988 8:   L_strict[buf] = lay
989 10: for all comp in ConnectedComponents(Ops, UseList, AliasGroups) do ▷ Phase II: Common propagation
990 11:   q = queue_of_all_op_indices()
991 12:   FinishInferQueue("COMMON", L, L_strict, q)
992 13:   for all comp in ConnectedComponents(Ops, UseList, AliasGroups) do ▷ Phase III: Free choices, minimize register cost
993 14:     best = (infinity, null) ▷ (reg_cost, payload)
994 15:     for all root in comp.members do
995 16:       ops_bak = Snapshot(Ops)
996 17:       L_tmp = copy(L)
997 18:       ok = TryInferFromRoot(root, "FREE", L_tmp, L_strict)
998 19:       if ok then
999 20:         for all other in comp.members, other != root do
1000 21:           ok = ok and TryInferFromRoot(other, "FREE", L_tmp, L_strict)
1001 22:         if ok then
1002 23:           cost = SumFragmentRegisters(L_tmp)
1003 24:           candidate = (cost, (Snapshot(Ops), L_tmp))
1004 25:           best = MinByRegister(best, candidate)
1005 26:           Restore(Ops, ops_bak)
1006 27:           assert(best != null)
1007 28:           (ops_snap, L_best) = best.payload
1008 29:           ApplySnapshot(Ops, ops_snap)
1009 30:           L = L_best ▷ Alias completion
1010 31:
1011 32: for all (var, buffers) in AliasGroups do
1012 33:   if exists b in buffers such that L[b] is defined then
1013 34:     ref = L[b]
1014 35:     for all buf in buffers do
1015 36:       if L[buf] is undefined then
1016 37:         L[buf] = ReshapeIfNeeded(ref, shape(buf))
1017 38:   for all (buffer, dummy) in UseList do
1018 39:   if scope(buffer) == "local.fragment" then
1019 40:     assert(L[buffer] is defined)
1020 41:
1021 42: (ForMap, PredMap) = CollectLoopLayoutsAndPredicates(Ops)
1022 43: return (L, ForMap, PredMap)
1023
1024
1025

```

1026
1027
1028
1029

Algorithm 2 Core Subroutines for Layout Inference

```

1030
1031 1: procedure RunInferStep(op_id, level, update_queue, L, L_strict)
1032 2: op = Ops[op_id]
1033 3: args = {target, thread_bounds = ThreadBounds[op_id], layout_map = L, analyzer, buffer_oob = Buffer-
1034 OOB[op_id]}
1035 4: updates = op.InferLayout(args, level)                                ▷ list of (buffer, layout)
1036 5: for all (buf, lay_new) in updates do
1037 6:   if buf in L then
1038 7:     if scope(buf) == “local.fragment” and level != “STRICT” and buf not in L_strict then
1039 8:       if FragmentContains(L[buf], lay_new) then
1040 9:         L[buf] = lay_new
1041 10:        PropagateAlias(buf, lay_new, L, update_queue)
1042 11:        continue
1043 12:        assert(IsEqualLayout(L[buf], lay_new))
1044 13:        PropagateAlias(buf, lay_new, L, update_queue)
1045 14:      else
1046 15:        L[buf] = lay_new
1047 16:        PropagateAlias(buf, lay_new, L, update_queue)
1048 17:        if update_queue then
1049 18:          enqueue_all_users(buf)
1050
1051 19: procedure FinishInferQueue(level, L, L_strict, q)
1052 20: while q is not empty do
1053 21:   id = q.pop()
1054 22:   RunInferStep(id, level, true, L, L_strict)
1055
1056 23: procedure PropagateAlias(src_buf, src_layout, L, update_queue)
1057 24: for all sib in AliasGroups[src_buf.storage_var] where sib != src_buf do
1058 25:   if shape(src_layout) == shape(sib) then
1059 26:     tgt = src_layout
1060 27:   else
1061 28:     tgt = Reshape(src_layout, shape(sib))
1062 29:   if sib in L then
1063 30:     assert(IsEqualLayout(L[sib], tgt))
1064 31:   else
1065 32:     L[sib] = tgt
1066 33:     if update_queue then
1067 34:       enqueue_all_users(sib)
1068
1069 35: procedure TryInferFromRoot(root, level, L_tmp, L_strict)
1070 36: success = false
1071 37: try
1072 38: RunInferStep(root, level, true, L_tmp, L_strict)
1073 39: FinishInferQueue(level, L_tmp, L_strict, q)
1074 40: success = true
1075 41: catch LayoutConflict or NormalizeIterError
1076 42: success = false
1077 43: return success
1078
1079 44: function SumFragmentRegisters(L)
1080 45: total = 0
1081 46: for all (buf, layout) in L do
1082 47:   if layout.kind == “Fragment” then
1083 48:     total = total + product(layout.output_shape)
1084
1085 49: return total
  
```

1080 **Algorithm 3** Loop Lowering: Binding, Vectorization, Predication

1081 1: **procedure** ApplyLoopLayoutTransformations(ForLoop, ForMap, PredMap, thread_var)

1082 2: loop_layout = ForMap[ForLoop]

1083 3: parallel_loop =

1084 (not skip_thread_partition) and

1085 (not local_register_only(ForLoop)) and

1086 (not store_into_local(ForLoop))

1087 4: **if** parallel_loop **then**

1088 5: ForLoop = PartitionLoop(ForLoop, thread_var, analyzer, loop_layout)

1089 6: has_non_local = touches_non_local(ForLoop.body)

1090 7: has_reducer = contains_reducer(ForLoop.body)

1091 8: has_cast_ops = contains_nonreduction_cast_store(ForLoop.body)

1092 9: **if** (has_non_local or has_cast_ops) and (not has_reducer) **then**

1093 10: ForLoop = VectorizeLoop(ForLoop)

1094 11: **if** ForLoop in PredMap and parallel_loop **then**

1095 12: **return** IfThenElse(PredMap[ForLoop], ForLoop)

1096 13: **return** ForLoop

1096
1097
1098 to obtain layout updates. Updates are merged into L with two safeguards: (i) for pre-existing
1099 buffers, strict equality is required unless the buffer is a register Fragment and the new layout
1100 contains the old one (per FragmentContains), in which case a safe refinement is allowed
1101 when not in “STRICT” and not locked by L_{strict} ; (ii) every update triggers PropagateAlias,
1102 which reshapes to sibling shapes as needed and enforces alias-wise equality, enqueueing users
1103 when in worklist mode. TryInferFromRoot runs a guarded, queue-based inference seeded
1104 at a chosen root and catches LayoutConflict/NormalizeIterError to mark a candidate
1105 infeasible. SumFragmentRegisters accumulates the product of each fragment layout’s output
1106 shape, serving as a proxy for total register footprint.

1107
1108 **Alias Completion and Validity Checks.** After the three phases, each alias group is revis-
1109 ited: if any sibling has a layout, the rest are filled by reshaping that layout to their shapes.
1110 The pass asserts that all local.fragment buffers used in the IR are defined. Finally,
1111 CollectLoopLayoutsAndPredicates summarizes loop-level binding decisions and out-
1112 of-bounds predicates into (ForMap, PredMap).

1113 **Loop Lowering: Binding, Vectorization, Predication (Alg. 3).** Given a loop and
1114 (ForMap, PredMap), the lowering proceeds as follows:

1115

- 1116 • **Thread binding.** If the loop is parallelizable (not skipped, not purely local-register, and it
1117 touches non-local memory), PartitionLoop binds iterations to the hardware thread variable
1118 using the loop layout from ForMap, aligning with the previously inferred thread/block
1119 organization.
- 1120 • **Vectorization.** If the loop body either touches non-local memory or performs non-reduction
1121 cast stores, and there is no reducer present, VectorizeLoop is applied. This realizes the
1122 vector length implied by the chosen layout, improving coalescing and matching hardware
1123 vector widths while mitigating bank conflicts.
- 1124 • **Predication.** If the loop may encounter boundary conditions and is parallel, the loop body
1125 is guarded with IfThenElse using the predicate from PredMap, ensuring safe accesses
1126 without sacrificing parallel throughput.

1127
1128 **Discussion.** The division into STRICT/COMMON/FREE keeps the search tractable: rigid,
1129 hardware-mandated forms are locked first; compatible information is then propagated to convergence;
1130 and only the remaining degrees of freedom are explored via component-local, snapshot-and-choose
1131 search guided by a register-cost objective. Alias propagation guarantees storage-consistent ad-
1132 dress mappings, while fragment-aware refinement enables safe specialization of register tiling. The
1133 produced (\mathcal{L} , ForMap, PredMap) bridge high-level tile indices and low-level memory/thread orga-
1134 nization, enabling performant, portable lowering across backends.

1134 **Algorithm 4** Pipeline Inference

1135 1: **procedure** PipelineInference(f , NumStages, InitOrderMap, InitStageMap)

1136 2: OrderMap \leftarrow copy(InitOrderMap)

1137 3: StageMap \leftarrow copy(InitStageMap)

1138 4: **for all** serial loop L in f .body **do**

1139 5: **if** $L \in$ OrderMap **and** $L \in$ StageMap **then**

1140 6: **continue**

1141 7: **if** $L \notin$ NumStages **then**

1142 8: **continue**

1143 9: $n = \text{NumStages}[L]$

1144 10: root = L .body

1145 11: seq = FlattenToSeq(root)

1146 12: Infos \leftarrow []

1147 13: **for** $i = 0$ to $|\text{seq}| - 1$ **do**

1148 14: $(R, W, C) = \text{RWCollect}(\text{seq}[i])$

1149 15: Infos.push_back(StageInfo(R, W, i , C))

1150 16: $S = \text{CollectCopyReads}(\text{Infos})$

1151 17: PropagateProducers(Infos, S)

1152 18: ComputeLastUse(Infos)

1153 19: order_idx = 0

1154 20: **for all** p in Infos **do**

1155 21: **if** FirstStage(p) **and** p .last_use $\neq -1$ **then**

1156 22: **continue**

1157 23: p .order = order_idx; order_idx \leftarrow order_idx + 1; p .stage = n

1158 24: **for all** q in Infos **do**

1159 25: **if** FirstStage(q) **and** q .last_use = p .original_idx **then**

1160 26: q .order = order_idx; order_idx \leftarrow order_idx + 1; q .stage = 0

1161 27: **assert**(order_idx = $|\text{Infos}|$)

1162 28: $k = \text{TailCopyCount}(\text{Infos})$

1163 29: **if** $k > 0$ **and** $n \geq 2$ **then**

1164 30: **for all** p in Infos **do**

1165 31: p .order = $(p$.order + k) \ mod \ $|\text{Infos}|$

1166 32: **if not** p .copy **and** **not** p .producer **then**

1167 33: p .stage = p .stage - 1

1168 34: orders = [p .order for $p \in$ Infos]

1169 35: stages = [p .stage for $p \in$ Infos]

1170 36: ApplySoftwarePipeline(L , orders, stages, OrderMap, StageMap)

1171 37: **return** (OrderMap, StageMap)

F PIPELINE INFERENCE ALGORITHM

1171 **Goal and Scope.** Algorithm 4 computes a software-pipelined schedule for serial loops that
 1172 expose staged data movement and computation. Given an input function f , an a priori upper
 1173 bound on the number of pipeline stages NumStages, and optional initial order/stage annotations
 1174 (InitOrderMap, InitStageMap), the pass produces a pair of maps (OrderMap, StageMap).
 1175 For each eligible serial loop L in f .body, the algorithm assigns (i) a total order index to every
 1176 statement in the flattened loop body and (ii) a stage id in $\{0, \dots, n - 1\}$, where $n = \text{NumStages}[L]$,
 1177 thereby enabling backend-specific software pipelining and overlapped execution of copies and
 1178 compute.

1179 **Loop Selection and Linearization.** The outer procedure PipelineInference first filters
 1180 serial loops: if a loop L already has entries in both OrderMap and StageMap, or if it lacks a stage
 1181 budget in NumStages, it is skipped. For each remaining loop, the body is linearized into a sequence
 1182 seq via FlattenToSeq, which yields a stable, single-pass order of statements. Each sequence
 1183 element is then summarized into a StageInfo record containing its read set, write set, original
 1184 index, and a Boolean flag indicating whether the statement performs a global-to-shared copy.

1185 **Read/Write Classification and Copy Detection.** The helper RWCollect (Algorithm 5) traverses
 1186 a statement and classifies its memory behavior into three components: a set of read regions R , a

1188
1189 **Algorithm 5** Core Subroutines for Pipeline Inference
1190
1191 1: **function** RWCollect(stmt)
1192 2: $R = []$; $W = []$; $C = \text{false}$; $\text{within} = \text{false}$; $\text{isg} = \text{false}$
1193 3: **Visit**(stmt):
1194 4: **on** BufferStore(b , idxs , v):
1195 5: $W += \text{Region}(b, \text{idxs})$; $\text{isg} = \text{false}$; **Visit**(v);
1196 6: **if** isg **and** $\text{scope}(b) \in \{\text{"shared"}, \text{"shared.dyn"}\}$ **then** $C = \text{true}$;
1197 7: **on** BufferLoad(b , idxs):
1198 8: $R += \text{Region}(b, \text{idxs})$;
1199 9: **if** $\text{scope}(b) == \text{"global"}$ **and** not within **then** $\text{isg} = \text{true}$;
1200 10: **on** IfThenElse(c , a , b):
1201 11: $\text{within} = \text{true}$; **Visit**(c); $\text{within} = \text{false}$; **Visit**(a);
1202 12: **if** $b.\text{defined}()$ **then** **Visit**(b);
1203 13: **return** (R, W, C)
1204 14:
1205 15: **function** CollectCopyReads(Infos)
1206 16: $S = \text{set}()$
1207 17: **for all** p in Infos **where** $p.\text{copy}$ **do**
1208 18: **for all** r in $p.\text{reads}$ **do**
1209 19: $S.\text{add}(r.\text{buffer})$
1210 20: **return** S
1211 21:
1212 22: **procedure** PropagateProducers(Infos, S)
1213 23: **for all** p in Infos **where** $p.\text{copy}$ **do**
1214 24: $\text{upd} = \text{true}$
1215 25: **while** upd **do**
1216 26: $\text{upd} = \text{false}$
1217 27: **for all** q in Infos **where** $\text{not } q.\text{copy}$ **and** $q.\text{original_idx} < p.\text{original_idx}$ **do**
1218 28: **if exists** w in $q.\text{writes}$ **with** $w.\text{buffer} \in S$ **then**
1219 29: $q.\text{producer} = \text{true}$; $\text{upd} = \text{true}$
1220 30: **for all** r in $q.\text{reads}$ **do**
1221 31: $S.\text{add}(r.\text{buffer})$
1222 32:
1223 33: **procedure** ComputeLastUse(Infos)
1224 34: **for all** p in Infos **where** FirstStage(p) **do**
1225 35: **for** $i = p.\text{original_idx} + 1$ to $|\text{Infos}| - 1$ **do**
1226 36: **if exists** r in Infos[i]. reads , w in $p.\text{writes}$ **with** $r.\text{buffer} = w.\text{buffer}$ **and** MayConflict($r.\text{region}$,
1227 37: $w.\text{region}$) **then**
1228 38: $p.\text{last_use} = \max(p.\text{last_use}, i)$
1229 39: **function** TailCopyCount(Infos)
1230 40: $c = 0$; $mn = |\text{Infos}|$; $mx = 0$
1231 41: **for all** p in Infos **do**
1232 42: **if** FirstStage(p) **then**
1233 43: $c \leftarrow c + 1$; $mn = \min(mn, p.\text{order})$
1234 44: **else**
1235 45: $mx = \max(mx, p.\text{order})$
1236 46: **if** $mn > mx$ **then**
1237 47: **return** c
1238 48: **else**
1239 49: **return** -1
1240 50:
1241 51: **procedure** ApplySoftwarePipeline(L , orders, stages, OrderMap, StageMap)
1242 52: $\text{OrderMap}[L] \leftarrow \text{orders}$
1243 53: $\text{StageMap}[L] \leftarrow \text{stages}$
1244 54:
1245 55: **function** FirstStage(p)
1246 56: **return** $p.\text{copy}$ **or** $p.\text{producer}$
1247 57:
1248 58: **function** MayConflict(a , b)
1249 59: **return** Intersect(IntSet(a), IntSet(b)) $\neq \text{Nothing}$

1242 set of write regions W , and a Boolean flag C for copy-like behavior. The visitor marks loads and
 1243 stores according to buffer scope (e.g., global, shared, shared.dyn), and uses a simple state
 1244 machine over the control-flow context (within, isg) to detect global loads that feed subsequent
 1245 shared-memory stores. Whenever a BufferStore into a shared buffer is preceded by such a global
 1246 read, the statement is classified as a *copy* ($C = \text{true}$), allowing the later stages to identify candidate
 1247 prologue/epilogue moves for software pipelining.

1248
 1249 **Producer Propagation and Lifetime Analysis.** Given the per-statement summaries,
 1250 CollectCopyReads aggregates the set S of buffers read by copy statements, which serves
 1251 as the seed for producer discovery. PropagateProducers then iteratively walks backwards over
 1252 the sequence to mark statements that *produce* any of the buffers in S before a copy: whenever a
 1253 non-copy statement writes to a buffer in S , it is labeled as a producer, and its own read buffers are
 1254 added to S . This fixed-point propagation captures multi-hop producer chains that eventually feed
 1255 global-to-shared copies.

1256 Next, ComputeLastUse computes a conservative *last-use index* for each first-stage statement
 1257 (copy or producer). For a given p , the pass scans later infos and checks whether any read region of a
 1258 later statement may conflict with any write region of p , using MayConflict and an interval-set
 1259 intersection test. The largest index where such a conflict occurs is recorded as $p.\text{last_use}$, providing a
 1260 lifetime window that guides stage assignment and the positioning of prefetch-like operations.

1261
 1262 **Stage Assignment and Tail Rotation.** With producer/copy labels and last-use information in place,
 1263 PipelineInference assigns an initial order and stage for each info in a single forward pass. The
 1264 core policy is that a first-stage statement p that is *dead* within the steady-state (i.e., $\text{FirstStage}(p)$
 1265 and $p.\text{last_use} = -1$) participates directly in the final pipelined schedule; otherwise, such statements
 1266 are skipped in this phase until their consumers are placed. For each selected p , the algorithm emits p
 1267 at the current order index with stage n (typically the last stage), then searches for matching first-stage
 1268 statements q whose last use equals $p.\text{original_idx}$ and places those q immediately after p at stage 0.
 1269 This yields an interleaving of first-stage and steady-state work that respects data dependencies and
 1270 the inferred lifetimes. The invariant $\text{order_idx} = |\text{Infos}|$ is asserted at the end to guarantee that
 1271 all statements receive a unique order.

1272 To further improve the pipeline structure, TailCopyCount detects whether first-stage statements
 1273 form a contiguous tail segment in the assigned order. It counts the number of first-stage infos c , and
 1274 tracks the minimum order index among them (mn) and the maximum order index of non-first-stage
 1275 statements (mx). If $mn > mx$, first-stage statements appear strictly after all other work, and the
 1276 function returns c ; otherwise, it returns -1 . When a positive tail count k is found and at least two
 1277 stages are available ($n \geq 2$), the algorithm rotates the schedule by k positions in a modular fashion
 1278 and decrements the stage of non-copy, non-producer statements. Intuitively, this rotation shifts tail
 1279 copies into the prologue while pulling steady-state computation earlier, yielding a more balanced
 1280 pipeline across the n stages.

1281
 1282 **Map Materialization and Backend Interface.** After rotation, the final per-statement orders and
 1283 stages are collected into arrays `orders` and `stages`, which are committed to the global maps
 1284 via ApplySoftwarePipeline. For each loop L , `OrderMap` [L] records a permutation of the
 1285 flattened body, and `StageMap` [L] records a stage id for each element in that permutation. These
 1286 maps serve as the contract between the high-level pipeline inference and downstream backends: code
 1287 generators can exploit the stage structure to schedule prefetches, overlaps of global-to-shared copies
 1288 with compute, and explicit prologue/epilogue code, without re-running dependence analysis on the
 1289 original IR.

1290
 1291 **Discussion.** The pipeline inference algorithm deliberately decouples (i) classification of copy
 1292 and producer statements, (ii) lifetime and conflict analysis, and (iii) stage-aware ordering and
 1293 optional schedule rotation. The use of region-based read/write summaries and conservative
 1294 MayConflict checks ensures correctness under aliasing and partially overlapping accesses. At
 1295 the same time, the simple rotation heuristic (TailCopyCount) captures a common pattern in
 1296 GPU kernels where global-to-shared transfers form a logical prologue or epilogue. By emitting
 1297 (`OrderMap`, `StageMap`) instead of directly rewriting the IR, the pass remains backend-agnostic

1296 while still exposing enough structure for aggressive software pipelining and latency hiding across
 1297 diverse hardware targets.
 1298

1300 G EFFECTIVENESS OF THE COST MODEL

1302 TILELANG employs an analytical cost model to prune suboptimal candidates and prioritize high-
 1303 potential ones. This approach yields schedules that match or closely approach the performance of
 1304 the best results of brute-force search or exhaustive autotuning, while requiring orders of magnitude
 1305 less tuning effort. For example, on GEMM-FP16 \times FP16 workloads derived from models such as
 1306 LLaMA-70B, TILELANG prunes 95% of candidate schedules, retaining only the top 5%. Despite
 1307 this aggressive pruning, it achieves on average 98.47% of the performance (in TFLOPS) of the
 1308 best configurations found by exhaustive search, substantially reducing compilation time with only
 1309 negligible performance loss. This also serves as an evaluation of the effectiveness of our roofline-
 1310 based analytical cost model.

	<i>M</i>	<i>N</i>	<i>K</i>	Predicted-TopX / Best (TFLOPS)
1314	512	1024	8192	100.0%
1315	512	12288	12288	99.9%
1316	512	28672	8192	98.7%
1317	2048	12288	49152	100.0%
1318	4096	1024	7168	100.0%
1319	4096	14336	14336	100.0%
1320	4096	28672	8192	99.6%
1321	8192	8192	28672	100.0%
1322	8192	28672	8192	100.0%
1323	16384	1024	7168	98.4%

1323 Table 3: Accuracy of our analytical cost model: predicted top-5% schedules retain over 98% of the
 1324 best performance while pruning 95% of candidate schedules.

1328 H TUNING TIME

1330 As demonstrated in Table 4, TILELANG leverages a hardware-aware recommendation mechanism
 1331 to efficiently automate the design of high-performance computational kernels. The system achieves
 1332 average tuning durations of approximately 10 seconds across both NVIDIA H100 and AMD MI300X
 1333 accelerators. For the most complex operations, extended tuning times average 13.69 seconds on
 1334 H100 and 15.08 seconds on MI300X, reflecting the scalability of our approach under computationally
 1335 intensive workloads.

1336 We also conducted a direct tuning-time comparison with Ansor/AutoTVM and Triton for the GEMM
 1337 and 2D convolution kernel on H100. TileLang and Triton are tuned with 20 configs, and the number
 1338 of trials is set to 100 in Ansor. The results in Table 56 show that TileLang tunes markedly faster
 1339 than both frameworks—especially vs. TVM Ansor, which requires much longer empirical search.
 1340 This improvement comes from TileLang’s first-class tile IR, which defines a far more structured
 1341 optimization space, and from our cost-model-guided inference, which avoids large brute-force
 1342 searches.

1343 Table 4: Average Tuning Times for Different Operators

Operation	GEMM	DequantGEMM	FlashMHA	FlashMLA	FlashBSA
H100 Time (s)	9.05	9.15	13.48	13.69	13.46
MI300 Time (s)	10.99	11.10	14.67	15.03	15.08

1350 Table 5: Comparison of Average Tuning Times for GEMM
1351

1352 Operation	1353 GEMM1	1354 GEMM2	1355 GEMM3	1356 GEMM4
1354 TileLang Time (s)	1355 11.81	1356 11.78	1357 14.59	1358 14.31
1355 Triton Time (s)	1356 18.43	1357 18.24	1358 20.15	1359 20.07
1356 Ansor Time (s)	1357 518.52	1358 455.51	1359 3007.00	1360 4142.05

1359 Table 6: Comparison of Average Tuning Times for Conv2D
1360

1361 Operation	1362 Conv2D1	1363 Conv2D2	1364 Conv2D3	1365 Conv2D4	1366 Conv2D5	1367 Conv2D6	1368 Conv2D7	1369 Conv2D8
1362 TileLang Time (s)	1363 11.56	1364 17.49	1365 17.65	1366 17.41	1367 19.18	1368 18.87	1369 17.50	1370 17.21
1363 Triton Time (s)	1364 17.56	1365 18.76	1366 37.59	1367 38.57	1368 37.61	1369 18.82	1370 19.13	1371 18.97

1366

I MATMUL IMPLEMENTATION DIFFS: TVM VS. TILELANG VS. TRITON

1368 TILELANG achieves significant code-size reduction through its fundamentally different tile abstraction.
1369 Instead of manipulating raw pointers, TILELANG represents tiles as first-class IR constructs,
1370 endowed with explicit semantics for indexing, data movement, and pipelining. High-level primitives
1371 such as `copy`, `gemm`, and `pipelined` enable the compiler to automatically perform address com-
1372 putation and pipeline orchestration. In contrast, Triton requires programmers to manually manage
1373 memory access via pointer and offset arithmetic, resulting in a lower level of abstraction.

1374 Moreover, TILELANG supports maintaining execution context via `T.Kernel`, which obviates the
1375 need for developers to explicitly compute grid dimensions and launch kernels. This design choice
1376 further reduces code size by eliminating boilerplate associated with kernel invocation.

1378 In TVM, matrix multiplication kernels are typically expressed through simple tensor-compute defi-
1379 nitions. Performance optimization is achieved by applying hand-written schedules, which can transform
1380 the computation into a high-performance form. While this manual scheduling process may require
1381 tens to hundreds of lines of Python code, TVM also provides automatic scheduling mechanisms such
1382 as Ansor to explore schedule configurations. After scheduling, the tensor expressions are compiled
1383 and built into executable code.

1384 However, TVM’s scheduling-independent compute-expression abstraction has limited expressiveness
1385 for certain operators, such as FlashAttention and irregular sparse kernels. Automatic scheduling in
1386 TVM also faces challenges when dealing with a very large search space and when targeting new
1387 hardware backends with insufficient operator expressiveness. TILELANG addresses these limitations
1388 by employing a human-in-the-loop methodology to enhance expressiveness for complex and irregular
1389 workloads, and by leveraging a cost-model-driven scheduling approach to mitigate the search-space
1390 explosion issue.

1391

J COMPARISON OF FTG IR AND TVM IR

1393 Figure 13 (a) shows an FTG IR produced after the Pipeline Inference Pass, which explicitly ma-
1394 terializes a software pipeline with a prologue (prefetch the first A/B tiles and clear the accumula-
1395 tor), a steady-state loop that overlaps compute and prefetch, and an epilogue (final compute and
1396 write-back). This IR emphasizes schedule semantics and readability: block tiling, shared-memory
1397 staging, and double buffering are expressed with concise, high-level primitives such as `T.Kernel`,
1398 `alloc_shared/alloc_fragment`, `T.clear`, `T.copy`, and `T.gemm`. The same semantics
1399 are then lowered to TVM IR in Figure 13 (b), which preserves the pipeline structure while making
1400 GPU execution details explicit: `blockIdx/threadIdx` bindings, scoped buffers in shared/local
1401 memory, unrolled initialization, asynchronous copies (e.g., `cp.async`) with commit/wait groups,
1402 and a target-specific GEMM call. In short, (a) captures a tile-wise schedule for readability and port-
1403 ability, whereas (b) exposes fine-grained GPU mechanisms for maximum control and performance,
1404 without changing the prologue–steady-state–epilogue structure.

```
1404
1405
1406
1407
1408 ① TileLang
1409
1410
1411 1 @T.prim_func
1412 2 def matmul_kernel(
1413 3     A: T.Tensor((M, K), dtype),
1414 4     B: T.Tensor((K, N), dtype),
1415 5     C: T.Tensor((M, N), dtype),
1416 6     ):
1417     with T.Kernel(T.ceildiv(N, block_N), T.ceildiv(M, block_M)) as (bx, by):
1418         A.shared = T.alloc_shared(block_M, block_K, dtype)
1419         B.shared = T.alloc_shared((block_K, block_N), dtype)
1420         C.local = T.alloca_fragment((block_M, block_N), accum_dtype)
1421         T.clear(C.local)
1422         for ko in T.Pipelined(T.ceildiv(K, block_K)):
1423             T.copy(A[bx * block_M, ko * block_K], A.shared)
1424             T.copy(B[ko * block_K, bx * block_N], B.shared)
1425             T.gemm(A.shared, B.shared, C.local)
1426             T.copy(C.local, C[bx * block_M, bx * block_N])
1427
1428     A, B, C: torch.Tensor
1429     matmul_kernel(A, B, C)
```

2 TVM

```

1420 1 A = te.placeholder((M, K), name="A", dtype=in_dype)
1421 2 B = te.placeholder((K, N), name="B", dtype=in_dype)
1422 3 k = te.reduce_axis((0, K), name="k")
1423 4 C = te.compute(
1424 5     (M, N),
1425 6     lambda i, j: te.sum(A[i, k].astype(accum_dype) *
1426 7         B[k, j].astype(accum_dype), axis=k),
1427 8     name="C",
1428 9 )
1429 10 args = [A, B, C]
1430 11 func = te.create_prim_func(args)
1431 12 module = tvm.IRModule(("main", func))
1432 13 module = ansor.autotune(module)
1433 14 executable = tvm.compile(module)
1434 15 A, B, C = torch.Tensz
1435 16 executable(A, B, C)

```

3 Triton

```

attrition.jit
1 def matmul_kernel(
2     a_ptr, b_ptr, c_ptr,
3     M, N, K,
4     stride_am, stride_ak,
5     stride_bk, stride_bn,
6     stride_cm, stride_cn,
7     BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_N: tl.constexpr, BLOCK_SIZE_K: tl.constexpr,
8     GROUP_SIZE_M: tl.constexpr,
9     ACTIVATION: tl.constexpr
10):
11    pid = tl.program_id(axis=0)
12    num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
13    num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)
14    num_pid_in_group = GROUP_SIZE_M * num_pid_n
15    group_id = pid // num_pid_in_group
16    first_pid_m = group_id * GROUP_SIZE_M
17    group_size_m = min(num_pid_m - first_pid_m, GROUP_SIZE_M)
18    pid_m = first_pid_m + ((pid % num_pid_in_group) % group_size_m)
19    pid_n = (pid % num_pid_in_group) // group_size_m
20    offs_am = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)) % M
21    offs_bn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)) % N
22    offs_ck = tl.arange(0, BLOCK_SIZE_K)
23    a_ptrs = a_ptr + (offs_am[:], None) * stride_am + offs_k[None, :] * stride_ak
24    b_ptrs = b_ptr + (offs_k[None, :] * stride_bk + offs_bn[None, :] * stride_bn)
25    accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
26    for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
27        a = tl.load(a_ptrs, mask=offs_k[None, :] < K - k * BLOCK_SIZE_K, other=0.0)
28        b = tl.load(b_ptrs, mask=offs_k[None, :] < K - k * BLOCK_SIZE_K, other=0.0)
29        accumulator = tl.dot(a, b, accumulator)
30        a_ptrs += BLOCK_SIZE_K * stride_ak
31        b_ptrs += BLOCK_SIZE_K * stride_bk
32        c = accumulator.to(tl.float16)
33        offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
34        offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
35        c_ptrs = c_ptr + stride_cm * offs_cm[:, None] + stride_cn * offs_cn[None, :]
36        c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)
37        tl.store(c_ptrs, c, mask=c_mask)
38
39
40 def matmul(a, b):
41     ...
42     grid = lambda META: (cdiv(M, META['BLOCK_SIZE_M']) * cdiv(N, META['BLOCK_SIZE_N']))
43     matmul_kernel[grid](
44         a, b, c,
45         M, N, K,
46         a.stride(0), a.stride(1),
47         b.stride(0), b.stride(1),
48         c.stride(0), c.stride(1),
49         ACTIVATION=activation
50     )
51
52 return c

```

Figure 12: Side-by-side diff of minimal GEMM implementations in TVM, TILELANG, and Triton (first page).

```

1436     def matmul(
1437         M: T.Tensor((M, N), "float16"),
1438         N: T.Tensor((N, K), "float16"),
1439         C: T.Tensor((M, N), "float16"),
1440         threads=16):
1441             with T.Kernel(
1442                 "ceildiv(M, block_N),",
1443                 "ceildiv(M, block_M), threads=threads) as (bx, by):
1444                 # Buffer Allocation
1445                 A_shared = T.alloc_shared((block_M, block_N), "float16")
1446                 B_shared = T.alloc_shared((block_K, block_N), "float16")
1447                 C_local = T.alloc_fragment((block_M, block_N), "float16")
1448
1449                 # Initialize C_local
1450                 T.clear(C_local)
1451
1452                 # Main Loop with Expanded Pipeline
1453                 T.copy(A[bx * block_M, 0 : block_K], A_shared)
1454
1455                 T.copy(B[0 : block_K, bx * block_N], B_shared)
1456
1457                 # Main Loop with Pipeline Annotation
1458                 for k in T.serial(ceildiv(K, block_K)):
1459                     T.gemm(A_shared, B_shared, C_local)
1460
1461                     T.copy(A[bx * block_M, k * block_K : block_K], A_shared)
1462
1463                     T.copy(B[k * block_K, bx * block_N], B_shared)
1464
1465                 # Compute the last stage
1466                 T.gemm(A_shared, B_shared, C_local)
1467
1468                 # Copy the result to the output buffer
1469                 T.copy(C_local, C[bx * block_M, bx * block_N])

```

```

1 @T.prim_func
2   def Matmul(
3     A: T.Tensor((M, K), "float16"), B: T.Tensor((K, N), "float16"),
4     C: T.Tensor((M, N), "float16"),
5   ):
6     A_shared = T.decl_buffer((block_M * block_K,), dtype, scope="shared")
7     B_shared = T.decl_buffer((block_K * block_N,), dtype, scope="shared")
8     C_local = T.decl_buffer((128,), accm_dtype, scope="local")
9     bx = T.thread_binding(T.ceildiv(N, block_N), "blockIdx.x")
10    by = T.thread_binding(T.ceildiv(M, block_M), "blockIdx.y")
11    tid = T.thread_binding(threads, "threadIdx.x")
12    for i in T.unroll(128):
13      C_local[i] = T.float32(0)
14    for i in T.unroll(4):
15      T.ptx_cp.async("uint8", A_shared.data, 0, A.data, 0, 16)
16      T.ptx_cp.async("uint8", B_shared.data, 0, B.data, 0, 16)
17    T.ptx_commit_group()
18    for ko in T.thread.ceildiv(K, block_K) - 1:
19      T.ptx_wait_group(0)
20      T.call_extern(
21        "tl::gemm_ss<128, 128, 32, 2, 2, 0>",
22        T.tvm_access_ptr(A_shared.data, 0, block_M * block_K, 1),
23        T.tvm_access_ptr(B_shared.data, 0, block_K * block_N, 1),
24        T.tvm_access_ptr(C_local.data, 0, 128, 3),
25      )
26      for i in T.unroll(4):
27        T.ptx_cp.async("uint8", A_shared.data, 0, A.data, 0, 16)
28        T.ptx_cp.async("uint8", B_shared.data, 0, B.data, 0, 16)
29        T.ptx_commit_group()
30    T.ptx_wait_group(0)
31    T.call_extern(
32      "tl::gemm_ss<128, 128, 32, 2, 2, 0>",
33      T.tvm_access_ptr(A_shared.data, 0, block_M * block_K, 1),
34      T.tvm_access_ptr(B_shared.data, 0, block_K * block_N, 1),
35      T.tvm_access_ptr(C_local.data, 0, 128, 3),
36    )
37    ...

```

Figure 13: A side-by-side comparison showing how tile-level FTG-IR is lowered into Tensor IR.

K COMPARISON OF FLASHMLA IMPLEMENTATIONS ON DIFFERENT ARCHITECTURES: NVIDIA vs. AMD

```

1  def flash_att():
2      O: T.Tensor, Q:Pe: T.Tensor, KV: T.Tensor,
3      K_pe: T.Tensor, Output: T.Tensor,
4  :)
5  with T.Kernelheadedness // (Block_H, Kv_group_num, batch, threads=threads) as (hid, bid):
6  :)
7  O_shared = T.alloccshared(block_H, Kv_group_num, dype)
8  O_pe_shared = T.alloccshared(block_H, pe_dim, dype)
9  KV_shared = T.alloccshared(block_H, Kv_group_num, dype)
10 K_pe_shared = T.alloccshared(block_H, pe_dim, dype)
11 accs = T.alloccfragment(block_H, block_N, accum_dype)
12
13 S_shared = T.alloccshared(block_H, block_N, dype)
14
15 accs.cast = T.alloccfragment(block_H, block_N, accum_dype)
16 O_pe.cast = T.alloccshared(block_H, block_N, dype)
17 T.annote(O_shared, 1)
18 T.annote(KV_shared, 2)
19 T.annote(K_pe_shared, 3)
20 scores, max_prev = T.alloccfragment(block_H, accum_dype)
21 scores_scale = T.alloccfragment(block_H, accum_dype)
22 scores.sum = T.alloccfragment(block_H, accum_dype)
23 logsum = T.alloccfragment(block_H, accum_dype)
24 cur_kv_head = hid // (Kv_group_num // block_H)
25 T.copy(BLOCK_H, hid // VALID_BLOCK_H, hid // VALID_BLOCK_H, 1, O_shared)
26 T.copy(O_pe, hid // VALID_BLOCK_H, hid // VALID_BLOCK_H, 1, O_pe_shared)
27 T.fillacc(0.0)
28 T.fillacc(-max, -T.infinity(accum_dype))
29 loop_range = T.coalidiv(seqlen_kv, block_N)
30 for k in T.PipelineDil(lolo_range, num_stages=num_stages):
31     T.copy((hid + K * block_N, cur_kv_head, 1), KV_shared)
32     T.copy(K_pe, hid + K * block_N, cur_kv_head, 1, K_pe_shared)
33
34     T.gemm(O_shared, KV_shared, accs, transpose_B=True, policy=FullCol, clear_accum=True)
35     T.gemm(O_pe_shared, K_pe_shared, accs, transpose_B=True, policy=FullCol)
36
37     T.copy(scores_max, scores_max_prev)
38     T.fillacc(scores_max, -T.infinity(accum_dype))
39     T.reduce(max(accs, scores_max, dim=1, clear=False)
40     for i in T.Parallel(block_H):
41         scores_max[i] = T.max(scores_max[i], scores_max_prev[i])
42     for i in T.Parallel(block_H):
43         scale = T.exp2(scores_max_prev[i] * scale - scores_max[i] * scale)
44     for i in T.Parallel(block_H, block_N):
45         accs[i, jj] = T.exp2(accs[i, jj] * scale - scores_max[i] * scale)
46     T.reduce(sum(accs, scores.sum, dim=1)
47     T.copy(accs, accs, shared)
48
49     logsum[1] = logsum[1] * scores.scale[1] + scores.sum[1]
50     for i, j in T.Parallel(block_H, dim):
51         accs[i, jj] *= scores.scale[i]
52     T.gemm(O_shared, KV_shared, accs, policy=FullCol)
53
54     for i, j in T.Parallel(block_H, dim):
55         accs[i, jj] /= logsum[1]
56     T.copy(accs, accs, shared)
57
58     T.copy(O_shared, Output[bid, hid // VALID_BLOCK_H:(hid + 1) * VALID_BLOCK_H, 1])
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
587
588
589
589
590
591
592
593
594
595
596
597
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
627
628
629
629
630
631
632
633
634
635
635
636
637
637
638
638
639
639
640
640
641
641
642
642
643
643
644
644
645
645
646
646
647
647
648
648
649
649
650
650
651
651
652
652
653
653
654
654
655
655
656
656
657
657
658
658
659
659
660
660
661
661
662
662
663
663
664
664
665
665
666
666
667
667
668
668
669
669
670
670
671
671
672
672
673
673
674
674
675
675
676
676
677
677
678
678
679
679
680
680
681
681
682
682
683
683
684
684
685
685
686
686
687
687
688
688
689
689
690
690
691
691
692
692
693
693
694
694
695
695
696
696
697
697
698
698
699
699
700
700
701
701
702
702
703
703
704
704
705
705
706
706
707
707
708
708
709
709
710
710
711
711
712
712
713
713
714
714
715
715
716
716
717
717
718
718
719
719
720
720
721
721
722
722
723
723
724
724
725
725
726
726
727
727
728
728
729
729
730
730
731
731
732
732
733
733
734
734
735
735
736
736
737
737
738
738
739
739
740
740
741
741
742
742
743
743
744
744
745
745
746
746
747
747
748
748
749
749
750
750
751
751
752
752
753
753
754
754
755
755
756
756
757
757
758
758
759
759
760
760
761
761
762
762
763
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1500
1501
1501
1502
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1510
1511
1511
1512
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1520
1521
1521
1522
1522
1523
1523
1524
1524
1525
1525
1526
1526
1527
1527
1528
1528
1529
1529
1530
1530
1531
1531
1532
1532
1533
1533
1534
1534
1535
1535
1536
1536
1537
1537
1538
1538
1539
1539
1540
1540
1541
1541
1542
1542
1543
1543
1544
1544
1545
1545
1546
1546
1547
1547
1548
1548
1549
1549
1550
1550
1551
1551
1552
1552
1553
1553
1554
1554
1555
1555
1556
1556
1557
1557
1558
1558
1559
1559
1560
1560
1561
1561
1562
1562
1563
1563
1564
1564
1565
1565
1566
1566
1567
1567
1568
1568
1569
1569
1570
1570
1571
1571
1572
1572
1573
1573
1574
1574
1575
1575
1576
1576
1577
1577
1578
1578
1579
1579
1580
1580
1581
1581
1582
1582
1583
1583
1584
1584
1585
1585
1586
1586
1587
1587
1588
1588
1589
1589
1590
1590
1591
1591
1592
1592
1593
1593
1594
1594
1595
1595
1596
1596
1597
1597
1598
1598
1599
1599
1600
1600
1601
1601
1602
1602
1603
1603
1604
1604
1605
1605
1606
1606
1607
1607
1608
1608
1609
1609
1610
1610
1611
1611
1612
1612
1613
1613
1614
1614
1615
1615
1616
1616
1617
1617
161
```

Figure 14: Comparison of FlashMLA implementations targeting NVIDIA (left) and AMD (right) architectures.

Figure 14 illustrates the code-level divergences between the FlashMLA implementations for NVIDIA and AMD architectures. While the high-level algorithmic structure remains unified, the implementation diverges to exploit distinct architectural strengths. First, Region 1 highlights the memory scope allocation for Query (Q) tiles, where the NVIDIA backend utilizes shared memory while the AMD backend prioritizes register memory. Similarly, Regions 2 and 5 collectively illustrate the divergence in intermediate accumulator storage: NVIDIA buffers these results in shared memory, whereas AMD maintains them directly in registers. Regarding output strategy, Regions 3 and 7 show that NVIDIA stages the output tile in shared memory to ensure coalesced transactions, in contrast to AMD, which performs a direct copy from registers to global memory. Finally, Regions 4 and 6 depict the adaptation of GEMM policies, employing a FullCol strategy for NVIDIA and FullRow for AMD to ensure optimal instruction performance.

L COMPARISON WITH RECENT SYSTEMS

Table 7 evaluates Causal MHA on an H100 ($B = 64, H = 64, D = 128$). TileLang is more concise than Tilus (Ding et al., 2025) (66 vs. 83 LOC) while achieving a 59%–75% performance gain. Table 8 evaluates the same MHA on an RTX 4090 ($B = 16, H = 32, D = 128$). TileLang achieves the best performance among baselines and outperforms the latest Tilus by 3%–4% with significantly fewer lines of code.

seq_len	Tilus (ms)	TileLang (ms)
1024	6.85	4.29
2048	25.65	15.69
4096	97.14	55.44

Table 7: Performance of Causal MHA on H100 ($B = 64$, $H = 64$, $D = 128$).

1512

1513

1514

1515

1516

1517

	seq_len	FA2 (tflops)	Triton (tflops)	Tilus (tflops)	TileLang (tflops)
MHA	1024	137.58	106.61	133.82	143.40
	2048	151.46	121.44	154.36	159.03
	4096	162.70	129.75	159.26	162.79
	8192	164.10	134.40	161.60	165.56
LoC		389	197	393	138

1523

Table 8: Performance and Lines of Code(LoC) of Causal MHA on RTX 4090 ($B = 16, H = 32, D = 128$).

1526

1527

1528

1529

1530

1531

1532

1533

1534

1535

(M, N, K)	TileLang (ms)	Gluon (ms)	Helion (ms)	Tilus (ms)
8192, 1024, 8192	0.16	0.30	0.26	0.31
8192, 8192, 8192	1.40	2.22	1.64	2.97
8192, 28672, 8192	5.09	7.82	5.87	10.19
8192, 8192, 28672	5.09	7.77	5.89	10.02

1540

Table 9: Comparison of TileLang, Gluon (LoC = 68), Helion (LoC = 24), and Tilus (LoC = 110) on GEMM workloads.

1541

1542

1543

1544

1545

1546

1547

1548

1549

1550

1551

1552

seq_len	TileLang (ms)	Helion (ms)
1024	0.17	0.19
2048	0.33	0.36
4096	0.65	0.71
8192	1.28	1.41
16384	2.53	3.01
32768	5.08	5.56

1559

Table 10: Performance of Mamba-chunk-scan on H100, with batch size 8, 80 attention heads, model dimension 64, dstate 128, and sequence lengths ranging from 1024 to 32768. Helion LOC=116, TileLang LOC=114.

1562

1563

1564

1565

1566 We evaluate the latest official Helion (PyTorch, 2025) and Gluon (OpenAI, 2025) examples that
 1567 support execution on Hopper, covering both GEMM and Mamba-chunk-scan workloads. As shown in
 1568 Table 9, TileLang achieves 1.15–1.62×, 1.52–1.83×, and 1.87–2.12× speedups over Helion, Gluon,
 1569 and Tilus, respectively.

1570 On Mamba-2-chunk-scan, TileLang further provides 1.10–1.19× speedups over Helion, as reported
 1571 in Table 10.

1572 The differences are attributable to design limitations in existing DSLs. Helion lacks an effective
 1573 tile-recommendation system, making optimization difficult and causing long tuning times (over 20
 1574 minutes). Gluon lacks appropriate abstractions and interfaces for pipeline scheduling, which makes it
 1575 difficult to achieve effective overlap of computation and memory operations. Second, Gluon operates
 1576 at a lower programming abstraction level, requiring users to manually make a larger number of design
 1577 and optimization decisions. As a result, writing high-performance kernels in Gluon is considerably
 1578 more challenging. This is also reflected by the fact that the Gluon implementation of GEMM requires
 1579 substantially more lines of code than the corresponding TileLang implementation. Tilus exposes
 1580 thread-block-level control over shared memory and registers, but without tile abstraction and critical
 1581 tile-related optimization. In contrast, TileLang’s tile-recommendation mechanism efficiently identifies
 1582 good tile configurations, and its pipeline-inference strategy generates an effective schedule that
 1583 overlaps computation and memory I/O. These additions provide a clearer qualitative and quantitative
 1584 comparison among recent DSLs and further highlight TILELANG’s performance and usability
 1585 advantages.

1586

1587 M BASELINE COMMIT HASHES

1588

1589 To ensure reproducibility, we provide the specific commit hashes for the baselines used in our
 1590 experiments that were not specified in the main text (see Table 11).

1591

1592 Table 11: Commit hashes for baseline frameworks.

1593

1594 System	Commit Hash
1595 Gluon	61cef5bdbfe2f179208f057d72c0b43b4885e5d2
1596 Helion	9a30bd18dcd87e08784691d5799e1af71fe0502f
1597 Tilus	505f566210319e2f55eeeddc393f01a203950510
1598 Anstor	64969035fd4f3c1ddcc23caa84567bf90e33889c
1599 ThunderKittens	572073a3935f91a268d37d5262cee0d950c2e9b2
1600 Marlin	1f25790bdd49fba53106164a24666dade68d7c90
1601 Block-Sparse-Attention	6ec5a27a0cd6bd92ea6296698d64e460c73da27e
1602 ComposableKernel	b8893b933963e86b76fa3fa088edede4504119f9
1603 Tilus (Variant?)	b2085f5ea08c504efeb2cab7cfaae3cd99701634
1604 FlashAttention-2 (FA2)	5d2cd3bcbaeff6fe1bfc5d0ff489451b0d4827a6
1605 TritonBench	a490be73ba84ab977de5cf78055a1dcb2e314f40

1606

1607

1608 N OPERATOR SHAPES IN OUR BENCHMARK

1609

1610

1611

1612

1613

1614

1615

1616

1617

1618

1619

1620

1621

1622

Table 12: Matrix shapes in our FP16 Matmul evaluation

1623

1624

1625

1626

1627

1628

	D0	D1	D2	D3
M	8192	8192	8192	8192
N	1024	8192	28672	8192
K	8192	8192	8192	28672

1629

1630

1631

1632

1633

1634

1635

1636

1637

1638

Table 13: Matrix shapes in our Fused Dequantize-Matmul evaluation

	M0	M1	M2	M3
M	1	1	1	1
N	1024	8192	28672	8192
K	8192	8192	8192	28672

Table 14: FlashAttention and Block Sparse Attention(with 50%, 90% sparsity) shapes in our evaluation

	FA0	FA1	FA2	FA3	FA4	FA5	FA6	FA7
batch	64	64	64	64	64	64	64	64
nheads	64	64	64	64	64	64	64	64
seq_len	1024	2048	4096	8192	1024	2048	4096	8192
head_dim	128	128	128	128	128	128	128	128
causal	false	false	false	false	true	true	true	true

Table 15: FlashMLA shapes in our evaluation

	FMLA0	FMLA1	FMLA2	FMLA3
batch	64	64	64	64
nheads	128	128	128	128
seq_len	1024	2048	4096	8192
head_dim	512	512	512	512
pe_dim	64	64	64	64
causal	false	false	false	false

Table 16: Convolution-2D shapes in our evaluation

	Conv0	Conv1	Conv2	Conv3	Conv4	Conv5	Conv6	Conv7
N	128	128	128	128	128	128	128	128
C	2048	512	512	512	256	1024	512	64
H	7	7	14	7	14	14	28	56
W	7	7	14	7	14	14	28	56
F	512	2048	512	512	256	256	128	64
K	1	1	3	3	3	1	1	1
S	1	1	2	1	1	1	1	1
D	1	1	1	1	1	1	1	1
P	0	0	1	1	1	0	0	0
G	1	1	1	1	1	1	1	1
HO	7	7	7	7	14	14	28	56
WO	7	7	7	7	14	14	28	56
Count	2	3	1	2	5	5	3	1

1672

1673

1674

1675

1676

1677

1678

1679

1680

1681

1682

1683

1684

1685

1686

1687

Table 17: Chunk-Gated-Delta-Net kernel shapes in our evaluation

1688

	CGDN0	CGDN1	CGDN2	CGDN3	CGDN4	CGDN5
batch	1	1	1	64	64	64
nheads	32	32	32	32	32	32
seq_len	16384	32768	65536	1024	2048	4096
head_dim	128	128	128	128	128	128

1695

1696

Table 18: Vertical Slash Sparse Attention shapes in our evaluation

1697

	VSSA0	VSSA1	VSSA2	VSSA3
batch	1	1	1	1
nheads	1	1	1	1
seq_len	8192	16384	32768	65536
head_dim	64	64	64	64
vertical size	1000	1000	800	1000
slash size	600	200	600	600

1705

1706

Table 19: Attention Sink shapes in our evaluation

1707

1708

1709

	Sink0	Sink1	Sink2	Sink3
batch	1	1	1	1
nheads	64	64	64	64
kv_heads	8	8	8	8
seq_len	1024	2048	4096	8192
kv_seq_len	1024	2048	4096	8192
head_dim	64	64	64	64
casual	true	true	true	true

1710

1711

1712

1713

1714

1715

1716

1717

1718

1719

1720

1721

1722

1723

1724

1725

1726

1727

1728 O KERNEL IMPLEMENTATIONS
17291730 O.1 MATRIX MULTIPLICATION (MATMUL)
1731

```

1732 1 @tilelang.jit
1733 2 def Matmul(A: T.Tensor, B: T.Tensor, C: T.Tensor):
1734 3     with T.Kernel(N // block_N, M // block_M,
1735 4         threads=threads) as (bx, by):
1736 5         A_shared = T.alloc_shared(block_M, block_K)
1737 6         B_shared = T.alloc_shared(block_K, block_N)
1738 7         C_local = T.alloc_fragment(block_M, block_N)
1739 8
1740 9         T.clear(C_local)
174110     for k in T.Pipelined(K // block_K, num_stages=2):
174211         T.copy(A[by * block_M, k * block_K], A_shared)
174312         T.copy(B[k * block_K, bx * block_N], B_shared)
174413         T.gemm(A_shared, B_shared, C_local)
174514
174615     T.copy(C_local, C[by * block_M, bx * block_N])

```

1742 Figure 15: Kernel Implementation of Matrix Multiplication.
17431744
1745 O.2 DEQUANTIZED MATRIX MULTIPLICATION
1746

```

1747 1 @tilelang.jit
1748 2 def dequantize_gemm(A: T.Tensor, B: T.Tensor, C: T.Tensor):
1749 3     with T.Kernel(T.ceildiv(N, n_partition), M, threads=(reduce_thread, n_partition)) as (bx, by):
1750 4         A_local = T.alloc_local([micro_size_k], in_dtype)
1751 5         B_quant_local = T.alloc_local([micro_size_k_compressed], storage_dtype)
1752 6         B_dequantize_local = T.alloc_local([micro_size_k], in_dtype)
1753 7         accum_res = T.alloc_local((1,), accum_dtype)
1754 8         reduced_accum_res = T.alloc_local((1,), accum_dtype)
1755 9
175610     T.clear(accum_res)
175711     for ko in T.serial(T.ceildiv(K, block_K)):
175812         for v in T.vectorized(micro_size_k):
175913             A_local[v] = A[by, ko * block_K + kr * micro_size_k + v]
176014
176115         for v in T.vectorized(micro_size_k_compressed):
176216             B_quant_local[v] = B[
176317                 bx * n_partition + ni,
176418                 ko * (reduce_thread * micro_size_k_compressed) +
176519                 kr * micro_size_k_compressed + v,
176620             ]
176721
176822         T.call_extern(
176923             "fast_decode_int4",
177024             T.address_of(B_quant_local[0]),
177125             T.address_of(B_dequantize_local[0]),
177226             dtype=in_dtype,
177327         )
177428
177529         for ki in T.serial(micro_size_k):
177630             accum_res[0] += A_local[ki] * B_dequantize_local[ki]
177731
177832         with T.attr(
177933             T.comm_reducer(lambda x, y: x + y, [T.Cast(accum_dtype, 0)]),
178034             "reduce_scope",
178135             T.reinterpret(T.uint64(0), dtype="handle"),
178236         ):
178337             T.evaluate(
178438                 T.tvm_thread_allreduce(
178539                     T.uint32(1),
178640                     accum_res[0],
178741                     True,
178842                     reduced_accum_res[0],
178943                     kr,
179044                     dtype="handle",
179145                 ))
179246         if kr == 0:
179347             C[by, bx * n_partition + ni] = reduced_accum_res[0]

```

1778 Figure 16: Implementation of Weight-Only Quantization ($W_{\text{FP4_E2M1}} A_{\text{FP16}}$) Matmul using TILE-
1779 LANG, showcasing support for mixed-precision computations via a simple form.
1780

1781

1782 O.3 FLASH ATTENTION IMPLEMENTATION
 1783

```

1784 1  @tilelang.jit
1785 2  def flash_attention(Q: T.Tensor, K: T.Tensor, V: T.Tensor, Output: T.Tensor):
1786 3      with T.Kernel(
1787 4          T.ceildiv(seq_len, block_M), heads, batch, threads=threads) as (bx, by, bz):
1788 5          Q_shared = T.alloc_shared([block_M, dim], dtype)
1789 6          K_shared = T.alloc_shared([block_N, dim], dtype)
1790 7          V_shared = T.alloc_shared([block_N, dim], dtype)
1791 8          O_shared = T.alloc_shared([block_M, dim], dtype)
1792 9          acc_s = T.alloc_fragment([block_M, block_N], accum_dtype)
179310  acc_s_cast = T.alloc_fragment([block_M, block_N], dtype)
179411  acc_o = T.alloc_fragment([block_M, dim], accum_dtype)
179512  scores_max = T.alloc_fragment([block_M], accum_dtype)
179613  scores_max_prev = T.alloc_fragment([block_M], accum_dtype)
179714  scores_scale = T.alloc_fragment([block_M], accum_dtype)
179815  scores_sum = T.alloc_fragment([block_M], accum_dtype)
179916  logsum = T.alloc_fragment([block_M], accum_dtype)
180017
180118  T.copy(Q[bz, bx * block_M:(bx + 1) * block_M, by, :], Q_shared)
180219  T.fill(acc_o, 0)
180320  T.fill(logsum, 0)
180421  T.fill(scores_max, -T.infinity(accum_dtype))
180522
180623  loop_range = (
180724      T.min(T.ceildiv(seq_len, block_N), T.ceildiv(
180825          (bx + 1) * block_M, block_N)) if is_causal else T.ceildiv(seq_len, block_N))
180926
181027  for k in T.Pipelined(loop_range, num_stages=num_stages):
181128      T.copy(K[bz, k * block_N:(k + 1) * block_N, by, :], K_shared)
181229      if is_causal:
181330          for i, j in T.Parallel(block_M, block_N):
181431              acc_s[i, j] = T.if_then_else(bx * block_M + i >= k * block_N + j, 0,
181532                  -T.infinity(acc_s.dtype))
181633
181734      else:
181835          T.clear(acc_s)
181936          T.gemm(Q_shared, K_shared, acc_s, transpose_B=True, policy=T.GemmWarpPolicy.FullRow)
182037          T.copy(scores_max, scores_max_prev)
182138          T.fill(scores_max, -T.infinity(accum_dtype))
182239          T.reduce_max(acc_s, scores_max, dim=1, clear=False)
182340          for i in T.Parallel(block_M):
182441              scores_scale[i] = T.exp2(scores_max_prev[i] * scale - scores_max[i] * scale)
182542          for i, j in T.Parallel(block_M, block_N):
182643              acc_s[i, j] = T.exp2(acc_s[i, j] * scale - scores_max[i] * scale)
182744          T.reduce_sum(acc_s, scores_sum, dim=1)
182845          for i in T.Parallel(block_M):
182946              logsum[i] = logsum[i] * scores_scale[i] + scores_sum[i]
183047          T.copy(acc_s, acc_s_cast)
183148          for i, j in T.Parallel(block_M, dim):
183249              acc_o[i, j] *= scores_scale[i]
183350          T.copy(V[bz, k * block_N:(k + 1) * block_N, by, :], V_shared)
183451          T.gemm(acc_s_cast, V_shared, acc_o, policy=T.GemmWarpPolicy.FullRow)
183552  for i, j in T.Parallel(block_M, dim):
183653      acc_o[i, j] /= logsum[i]
183754  T.copy(acc_o, O_shared)
183855  T.copy(O_shared, Output[bz, bx * block_M:(bx + 1) * block_M, by, :])
  
```

Figure 17: Implementation of Flash Attention with TILELANG.

1836 O.4 FLASHMLA IMPLEMENTATION
 1837

```

1838 1  @tilelang.jit
1839 2  def flash_mla(
1840 3      Q: T.Tensor([batch, heads, dim], dtype),
1841 4      Q_pe: T.Tensor([batch, heads, pe_dim], dtype),
1842 5      KV: T.Tensor([batch, seqlen_kv, kv_head_num, dim], dtype),
1843 6      K_pe: T.Tensor([batch, seqlen_kv, kv_head_num, pe_dim], dtype),
1844 7      Output: T.Tensor([batch, heads, dim], dtype),
1845 8  ):
1846 9      with T.Kernel(batch, heads // min(block_H, kv_group_num), threads=256) as (bx, by):
184710         Q_shared = T.alloc_shared([block_H, dim], dtype)
184811         S_shared = T.alloc_shared([block_H, block_N], dtype)
184912         Q_pe_shared = T.alloc_shared([block_H, pe_dim], dtype)
185013         KV_shared = T.alloc_shared([block_N, dim], dtype)
185114         K_pe_shared = T.alloc_shared([block_N, pe_dim], dtype)
185215         O_shared = T.alloc_shared([block_H, dim], dtype)
185316         acc_s = T.alloc_fragment([block_H, block_N], accum_dtype)
185417         acc_o = T.alloc_fragment([block_H, dim], accum_dtype)
185518         scores_max = T.alloc_fragment([block_H], accum_dtype)
185619         scores_max_prev = T.alloc_fragment([block_H], accum_dtype)
185720         scores_scale = T.alloc_fragment([block_H], accum_dtype)
185821         scores_sum = T.alloc_fragment([block_H], accum_dtype)
185922         logsum = T.alloc_fragment([block_H], accum_dtype)
186023
186124         cur_kv_head = by // (kv_group_num // block_H)
186225         T.use_swizzle(10)
186326
186427         T.copy(Q[bx, by * VALID_BLOCK_H:(by + 1) * VALID_BLOCK_H, :], Q_shared)
186528         T.copy(Q_pe[bx, by * VALID_BLOCK_H:(by + 1) * VALID_BLOCK_H, :], Q_pe_shared)
186629         T.fill(acc_o, 0)
186730         T.fill(logsum, 0)
186831         T.fill(scores_max, -T.infinity(accum_dtype))
186932
187033         loop_range = T.ceildiv(seqlen_kv, block_N)
187134         for k in T.Pipelined(loop_range, num_stages=2):
187235             T.copy(KV[bx, k * block_N:(k + 1) * block_N, cur_kv_head, :], KV_shared)
187336             T.copy(K_pe[bx, k * block_N:(k + 1) * block_N, cur_kv_head, :], K_pe_shared)
187437             T.clear(acc_s)
187538             T.gemm(
187639                 Q_shared, KV_shared, acc_s, transpose_B=True, policy=T.GemmWarpPolicy.FullCol)
187740             T.gemm(
187841                 Q_pe_shared,
187942                 K_pe_shared,
188043                 acc_s,
188144                 transpose_B=True,
188245                 policy=T.GemmWarpPolicy.FullCol)
188346             T.copy(scores_max, scores_max_prev)
188447             T.fill(scores_max, -T.infinity(accum_dtype))
188548             T.reduce_max(acc_s, scores_max, dim=1, clear=False)
188649             for i in T.Parallel(block_H):
188750                 scores_scale[i] = T.exp2(scores_max_prev[i] * scale - scores_max[i] * scale)
188851             for i, j in T.Parallel(block_H, block_N):
188952                 acc_s[i, j] = T.exp2(acc_s[i, j] * scale - scores_max[i] * scale)
189053                 T.reduce_sum(acc_s, scores_sum, dim=1)
189154                 T.copy(acc_s, S_shared)
189255             for i in T.Parallel(block_H):
189356                 logsum[i] = logsum[i] * scores_scale[i] + scores_sum[i]
189457             for i, j in T.Parallel(block_H, dim):
189558                 acc_o[i, j] *= scores_scale[i]
189659             T.gemm(S_shared, KV_shared, acc_o, policy=T.GemmWarpPolicy.FullCol)
189760             for i, j in T.Parallel(block_H, dim):
189861                 acc_o[i, j] /= logsum[i]
189962             T.copy(acc_o, O_shared)
190063             T.copy(O_shared, Output[bx, by * VALID_BLOCK_H:(by + 1) * VALID_BLOCK_H, :])
  
```

1880 Figure 18: Implementation of FlashMLA with TILELANG.
 1881
 1882
 1883
 1884
 1885
 1886
 1887
 1888
 1889

1890 O.5 BLOCK SPARSE ATTENTION IMPLEMENTATION
 1891

```

1892 1  @tilelang.jit
1893 2  def blocksparse_attn(Q: T.Tensor, K: T.Tensor, V: T.Tensor, BlockMask: T.Tensor, Output: T.Tensor):
1894 3  with T.Kernel(
1895 4      T.ceildiv(seq_len, block_M), heads, batch, threads=threads) as (bx, by, bz):
1896 5      Q_shared = T.alloc_shared([block_M, dim], dtype)
1897 6      K_shared = T.alloc_shared([block_N, dim], dtype)
1898 7      V_shared = T.alloc_shared([block_N, dim], dtype)
1899 8      O_shared = T.alloc_shared([block_M, dim], dtype)
1900 9      acc_s = T.alloc_fragment([block_M, block_N], accum_dtype)
190110     acc_s_cast = T.alloc_fragment([block_M, block_N], dtype)
190111     acc_o = T.alloc_fragment([block_M, dim], accum_dtype)
190112     scores_max = T.alloc_fragment([block_M], accum_dtype)
190113     scores_max_prev = T.alloc_fragment([block_M], accum_dtype)
190114     scores_scale = T.alloc_fragment([block_M], accum_dtype)
190115     scores_sum = T.alloc_fragment([block_M], accum_dtype)
190116     logsum = T.alloc_fragment([block_M], accum_dtype)
190117
190118     T.copy(Q[bz, bx * block_M:(bx + 1) * block_M, by, :], Q_shared)
190119     T.fill(acc_o, 0)
190120     T.fill(logsum, 0)
190121     T.fill(scores_max, -T.infinity(accum_dtype))
190122
190123     loop_range = (
190124         T.min(T.ceildiv(seq_len, block_N), T.ceildiv(
190125             (bx + 1) * block_M, block_N)) if is_causal else T.ceildiv(seq_len, block_N))
190126
190127     for k in T.Pipelined(loop_range, num_stages=num_stages):
190128         if BlockMask[bz, bx, by, k]:
190129             T.copy(K[bz, k * block_N:(k + 1) * block_N, by, :], K_shared)
190130             if is_causal:
190131                 for i, j in T.Parallel(block_M, block_N):
190132                     acc_s[i, j] = T.if_then_else(bx * block_M + i >= k * block_N + j, 0,
190133                                         -T.infinity(acc_s.dtype))
190134             else:
190135                 T.clear(acc_s)
190136                 T.gemm(Q_shared, K_shared, acc_s, transpose_B=True, policy=T.GemmWarpPolicy.FullRow)
190137                 T.copy(scores_max, scores_max_prev)
190138                 T.fill(scores_max, -T.infinity(accum_dtype))
190139                 T.reduce_max(acc_s, scores_max, dim=1, clear=False)
190140                 for i in T.Parallel(block_M):
190141                     scores_scale[i] = T.exp2(scores_max_prev[i] * scale - scores_max[i] * scale)
190142                 for i, j in T.Parallel(block_M, block_N):
190143                     acc_s[i, j] = T.exp2(acc_s[i, j] * scale - scores_max[i] * scale)
190144                 T.reduce_sum(acc_s, scores_sum, dim=1)
190145                 for i in T.Parallel(block_M):
190146                     logsum[i] = logsum[i] * scores_scale[i] + scores_sum[i]
190147                 T.copy(acc_s, acc_s_cast)
190148                 for i, j in T.Parallel(block_M, dim):
190149                     acc_o[i, j] *= scores_scale[i]
190150                 T.copy(V[bz, k * block_N:(k + 1) * block_N, by, :], V_shared)
190151                 T.gemm(acc_s_cast, V_shared, acc_o, policy=T.GemmWarpPolicy.FullRow)
190152
190153     for i, j in T.Parallel(block_M, dim):
190154         acc_o[i, j] /= logsum[i]
190155     T.copy(acc_o, O_shared)
190156     T.copy(O_shared, Output[bz, bx * block_M:(bx + 1) * block_M, by, :])
  
```

1928 Figure 19: Implementation of Block Sparse Flash Attention with TILELANG.
 1929
 1930
 1931
 1932
 1933
 1934
 1935
 1936
 1937
 1938
 1939
 1940
 1941
 1942
 1943