TILETUNE: AN EFFICIENT END-TO-END AUTOTUNING SYSTEM FOR TILE-BASED GPU KERNELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Tile-based domain-specific languages simplify GPU kernel development, but achieving high performance still requires tuning parameters such as tile sizes and pipeline depths. Existing tuning approaches face a tradeoff between efficiency and generality: search-based methods are broadly applicable but incur compilation and benchmarking costs for each evaluated configuration, while heuristic-based methods reduce evaluation work but often require kernel-specific customization. Moreover, the explicit tile structure that this new generation of languages exposes remains largely unexploited, as existing tuners were designed for loop-level schedules or fixed kernel templates.

We present TILETUNE, an end-to-end autotuning system that provides efficient, out-of-the-box tuning for tile-based GPU kernels, which balances generality, efficiency, and tile-awareness. At the algorithmic level, we propose a lightweight cost model using static program information rather than user-written, kernel-specific selection rules. It uses a shared analysis of explicit tile operations and resource requirements to rank and prune candidates while retaining empirically good selections. At the system level, we build a parallel task-processing runtime that reduces the cost of evaluating survivors through efficient dispatching, grouped compilation, and overlapping compilation and benchmarking. Across a range of kernels and hardware platforms, TILETUNE achieves 4.02–$5.35\times$ end-to-end tuning speedup over the TileLang search-based baseline. Its shortlist retains the full-search optimum in all evaluated A100, H200 and B300 workloads.

1 INTRODUCTION

Tile-based domain-specific languages (DSLs) such as Triton, TileLang, and ThunderKittens simplify GPU kernel development by exposing computation and data movement at the tile level (Tillet et al., 2019; Wang et al., 2026; Spector et al., 2024). Yet tile programs still expose performance-critical choices, including tile sizes, thread assignments, and pipeline depths. These choices, which together define a kernel configuration, must be made carefully for each given workload and target GPU to ensure high performance. Attention implementations such as FlashAttention-2 and FlashInfer also depend on work partitioning and tile choices (Dao, 2023; Ye et al., 2025).

Autotuning automates the selection of a high-performance configuration (Ansel et al., 2014). In the setting considered here, the developer provides a parameterized tile program and a set of candidate configurations for a given workload and target device. A straightforward autotuner compiles each candidate into a GPU kernel, checks its correctness, and measures its execution latency (Nugteren & Codreanu, 2015). However, this tuning process can be time-consuming (Ringlein et al., 2025a; Schoonhoven et al., 2023). Each uncached candidate requires program specialization, lowering, and device compilation before it can be tested on the GPU. Correctness checks and repeated executions for latency measurement add further burdens. This tuning process would be quite expensive even for a few hundreds of configurations, with recurring costs when developers change workload shapes, target devices, or kernel implementations, substantially slowing down kernel development (Ringlein et al., 2025b).

Existing autotuning approaches expose a tradeoff between *generality* and *efficiency*, and lack of *tile-awareness*. First, for generality, *search-based* tuning directly compiles and benchmarks candidate configurations, using measured latency for final selection. The same procedure applies to different

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

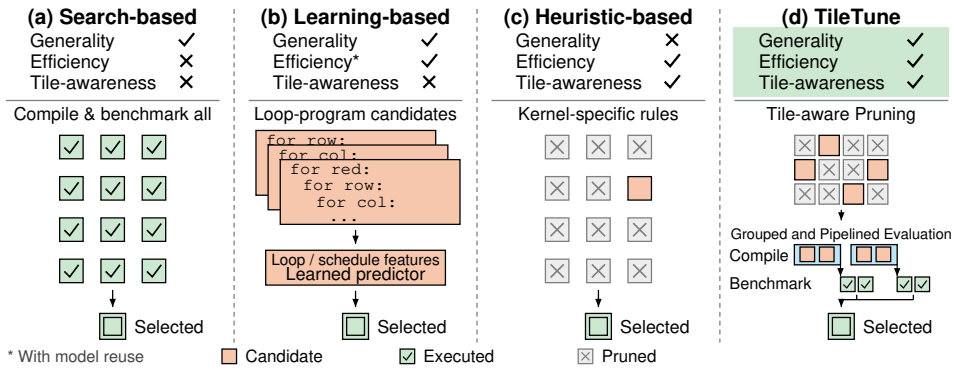

Figure 1: Four representative tuning workflows. (a) Search-based autotuning compiles and benchmarks every candidate before selecting the fastest measured kernel. (b) Learning-based autotuning uses measured configurations to train a predictor that prioritizes further candidates. (c) Heuristic-based autotuning uses kernel-specific rules to select a promising candidate. (d) TILETUNE prunes candidates using tile-program analysis, then groups compilation and overlaps it with benchmarking.

kernels without a hand-written performance ranking, but its evaluation cost grows with the number of configurations tested (Ringlein et al., 2025a). Second, for efficiency, *heuristic-based* tuning uses domain-specific performance knowledge to select promising configurations, as in DeepGEMM, Roller, tritonBLAS, and WaveTune (DeepSeek-AI, 2026a; Zhu et al., 2022; Swann et al., 2025; Zhang et al., 2026). Such methods reduce empirical evaluation, but extending their policies to new kernel families or hardware types would require specialized knowledge and adaptation (Ringlein et al., 2025b). Third, even learning-based tuners such as AutoTVM and Ansor reduce empirical evaluation through learned cost models (Chen et al., 2018b; Zheng et al., 2020), these systems are built around loop- and schedule-level representations, which result in excessive numbers of configurations and lengthy periods of model training/fitting, failing to exploit the explicit tile-program interface.

Taken together, existing methods offer general but costly empirical search, efficient but specialized heuristics, and learned guidance built around loop- and schedule-level representations (Figure 1). This leaves a clear gap: an efficient and general end-to-end autotuning system that works out of the box for modern tile-based languages. Such a system should support the full workflow, from a supplied tile program and candidate configurations to a validated, high-performance kernel, without requiring kernel-specific tuning policies. Our proposal, TILETUNE, addresses this gap through two stages: *tile-aware pruning* narrows the candidate set, while *grouped and pipelined evaluation* reduces the cost of evaluating the survivors.

To prune candidates with awareness of tile structure, TILETUNE derives a ranking score from the explicit operations and data movement in each tile program. Since a kernel's runtime is usually dominated by either compute or memory, a coarse estimate of each side suffices to filter out clearly slow candidates. Moreover, The static analysis and ranking policy are reused across supported kernels without kernel-specific rules, while the scores reflect each program and configuration. The model uses these scores to retain a promising subset, not to predict a single winner.

Grouped and pipelined evaluation reduces end-to-end cost by sharing device compilation across compatible candidates and allowing benchmark workers to measure ready kernels while other groups compile. Measured latency still determines the final choice. Together, the two stages aim to retain the generality of search-based tuning at a lower end-to-end cost.

We implement TILETUNE and integrate it into the autotuning framework of TileLang, a popular tile-based kernel DSL. Across a range of kernels and hardware platforms, TILETUNE achieves 4.02–$5.35\times$ end-to-end tuning speedup over the one-GPU TileLang autotuning baseline. We further compare pruning quality with representative learning-based and heuristic-based baselines. TILETUNE retains the full-search optimum in every evaluated A100, H200 and B300 workload, across all five kernel families (four on A100, which lacks native FP8 support).

Our contributions are summarized as follows:

- We develop a lightweight cost-model approach to candidate pruning based on static analysis of tile programs, targeting a reusable alternative to user-written kernel-specific rules while retaining empirical selection.
- We build an evaluation runtime that shares device compilation across compatible candidates and overlaps CPU compilation with GPU benchmarking through task dispatching to make the retained search more efficient.
- We integrate tile-aware pruning with grouped and pipelined evaluation in an end-to-end autotuning system, and evaluate its balance of generality, efficiency, and tile-awareness through tuning time, selected-kernel performance, and component costs. The end-to-end system achieves up to $5.35\times$ speedup over the baseline.

2 Autotuning for Tile Programs: Problem Setup

Tile-centric configuration family. Let P denote a specified tile program, h the target device, and $\mathcal{C}$ the supplied configuration list. The candidate family $\{P(c) : c \in \mathcal{C}\}$ specializes a common computation through tile and schedule parameters. A GEMM configuration, for example, can specify (B_M, B_N, B_K, S, T, R): three tile dimensions, pipeline depth, thread count, and rasterization choice.

Unlike loop-/schedule-level tuning, which explores transformations such as loop splitting and thread binding (Ragan-Kelley et al., 2013; Chen et al., 2018b; Shao et al., 2022), tile-program tuning starts from the tile operations and data movement explicit in P and varies their parameters through $\mathcal{C}$. Figure 2 illustrates these two tuning interfaces for GEMM. We focus on autotuning over tile-program configuration spaces.

Loop program

```
for ib in range(0, M, BM):
 for jb in range(0, N, BN):
  for iw in range(ib, ib + BM, WM):
   for jw in range(jb, jb + BN, WN):
    for kb in range(0, K, BK):
     for i in range(iw, iw + WM):
      for j in range(jw, jw + WN):
       for k in range(kb, kb + BK):
        C[i, j] += A[i, k] * B[k, j]
```

$|\mathcal{C}_{\text{loop}}| = 3^4 \times 4 \times 2^2 \times 4 \times 3 = 15{,}552$

Tile program

```
with T.Kernel(..., threads=T):
 T.use_swizzle(R)
 for ko in T.Pipelined(K/BK,S):
  T.copy(A[BM,BK], A_s)
  T.copy(B[BN,BK], B_s)
  T.gemm(A_s,B_s,C_f)
```

$|\mathcal{C}_{\text{tile}}| = 3 \times 6 \times 2 \times 4 \times 2^2 = 576$

Figure 2: FP16 GEMM at $(M, N, K) = (32768, 4096, 4096)$. The loop program sketches AutoTVM's Tensor Core dense schedule in TVM v0.9.0 (Apache TVM Developers, 2022b), while the tile program sketches TileLang's GEMM configuration space (Table 2). Tunable parameters are marked in red.

Measured latency and the reference optimum. A candidate produces an executable kernel only after successful compilation; correctness checks establish validity, and benchmarking yields its latency $L(c)$. Let $\mathcal{V} \subseteq \mathcal{C}$ contain the valid candidates in the full search. Assuming $\mathcal{V}$ is nonempty, its best configuration is

$$c^\star = \arg\min_{c \in \mathcal{V}} L(c). \tag{1}$$

Ranking scores for candidate selection. To reduce empirical evaluation, a class of autotuning methods uses an approximate performance model to assign each candidate a scalar score $\widehat{L}(c)$ (Chen et al., 2018b; Zhai et al., 2023; Zhu et al., 2022; Zhang et al., 2026). The goal is to preserve the latency ordering without requiring strict numerical agreement with $L(c)$. Ideally, for any $c_i, c_j \in \mathcal{V}$, we want $\widehat{L}(c_i) < \widehat{L}(c_j) \iff L(c_i) < L(c_j)$. But approximate scores may misorder candidate pairs. A direct score-based choice is

$$\tilde{c} = \arg\min_{c \in \mathcal{C}} \widehat{L}(c). \tag{2}$$

But $\tilde{c}$ may not match $c^\star$. The selected configuration still requires compilation and correctness checks.

3 Bridging Generality and Efficiency with Tile-Awareness

3.1 Search-based autotuning

Under the formulation in Section 2, a search-based autotuner exhaustively compiles $P(c)$ for every $c \in \mathcal{C}$. It checks each compiled kernel for correctness and measures $L(c)$ for valid configurations, and eventually returns the lowest-latency configuration $c^\star$ from Equation 1.

With fixed CPU and GPU resources, the tuning time of exhaustive search grows roughly in proportion to $|\mathcal{C}|$. Compilation takes CPU time to specialize the program and generate an executable for each configuration. Benchmarking takes GPU time to run valid kernels repeatedly and obtain reliable latency estimates. Tuning efficiency is therefore the main limitation of this approach. In preliminary runs on the original GEMM example grid, TileLang Autotune took about 230 s to tune $(M, N, K) = (32768, 4096, 4096)$ across 288 configurations. Nearly four minutes is already a long wait for a single kernel and shape, and developers pay it again whenever they change the workload shape, target device, or kernel implementation, slowing down kernel development.

3.2 Learning-based autotuning

To reduce the cost of exhaustive search, learning-based methods use measured latencies to prioritize further candidates. AutoTVM's XGBoost tuner is one example (Chen et al., 2018a;b; Apache TVM Developers, 2022c). Let $\mathcal{D} \subseteq \mathcal{V}$ contain the measured configurations and let $\phi(c)$ be the features of the schedule generated for c. The tuner fits f_θ using the measured throughputs, proportional to $1/L(c)$ for a fixed workload, and scores unmeasured configurations by

$$\widehat{L}(c) = -f_\theta(\phi(c)), \qquad c \in \mathcal{C} \setminus \mathcal{D}. \tag{3}$$

Here the negative sign matches the lower-is-better score in Section 2. The tuner evaluates promising configurations and refits the model as measurements continue accumulating (Apache TVM Developers, 2022c).

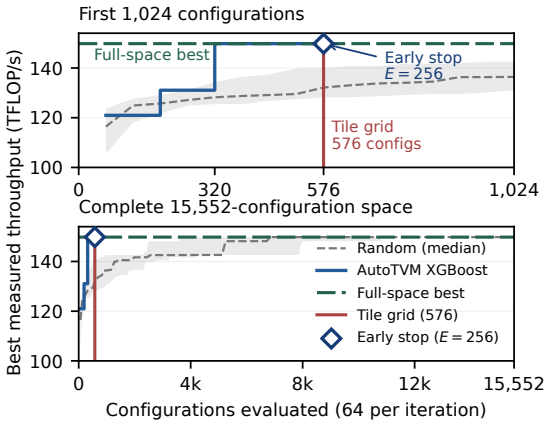

Figure 3: H100 AutoTVM GEMM: best measured throughput over 15,552 configurations. The diamond marks stopping with $E = 256$.

However, while the GEMM tile-program grid contains just 576 configurations (Table 2), AutoTVM's loop-schedule spaces could contain billions of programs (Chen et al., 2018b). Figure 3 reports our H100 replay over all 15,552 configurations of the AutoTVM GEMM schedule in Figure 2. Across three runs, XGBoost reached the measured full-space best after 341 evaluations on average. But in fact, this optimum is unknown during tuning. AutoTVM can only stop at a trial budget, e.g., E trials without improvement. The $E = 256$ stopping point in the figure occurs at 576 evaluations, matching the size of our tile-program GEMM grid. A CUDA tutorial sets $n_{\text{trial}} = 2000$ and $E = 600$ (Apache TVM Developers, 2022a), which would exhaust this grid. Stopping earlier cannot certify the fastest kernel. Thus, AutoTVM-style learning is a poor fit for compact tile-program spaces: without using explicit tiling structure to prune candidates before measurement (Zhu et al., 2022), any offline data-collection cost is hard to amortize over just 576 configurations.

3.3 Heuristic-based autotuning

Rather than learning from measurements, heuristic-based methods narrow the candidate set using expert knowledge of the operator and device. DeepGEMM, cuBLASLt, and CUTLASS's GEMM heuristics select or rank kernels using specialized rules (DeepSeek-AI, 2026a; NVIDIA, 2026a;b). Roller and tritonBLAS use analytical models to guide configuration selection, while WaveTune

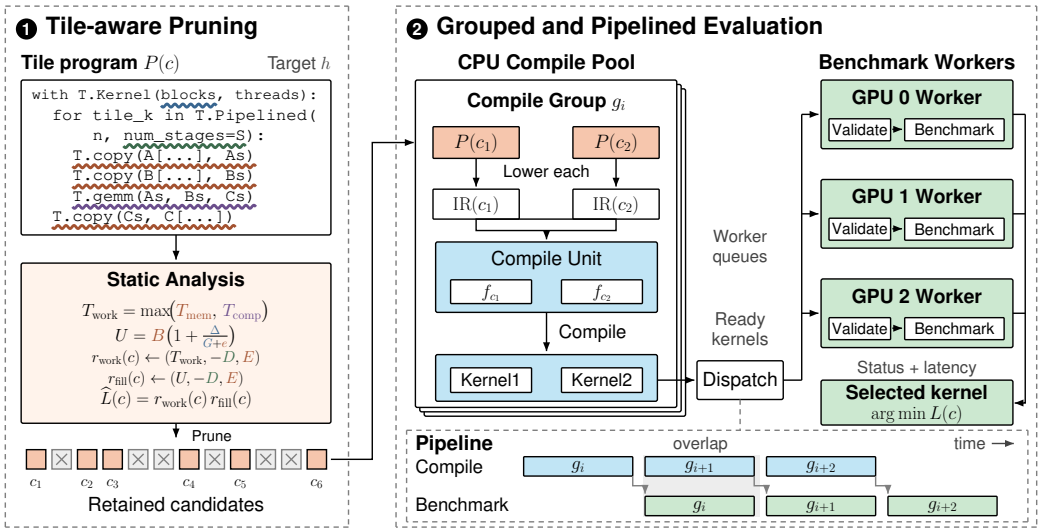

Figure 4: Two-stage workflow of TILETUNE. (1) Tile-aware pruning combines work and launch-underfill rankings to retain a subset before compilation. (2) The CPU compile pool lowers retained candidates separately and compiles compatible functions together. Ready kernels are dispatched to per-GPU benchmarking workers for validation and timing. The bottom timeline shows GPU benchmarking of one group overlapping CPU compilation of later groups. The valid kernel with the lowest measured latency is selected.

combines a structured model with offline profiling (Zhu et al., 2022; Swann et al., 2025; Zhang et al., 2026). These methods can reduce the compilation and benchmarking required by direct search. Their rules, templates, or calibration are often tailored to particular kernel families and hardware, so extending them to new kernels or devices may require additional engineering or profiling. This limits out-of-the-box generality. By pruning candidates without measuring them, these methods do not guarantee retaining the fastest configuration in $\mathcal{C}$.

3.4 BALANCING THE TRADE-OFFS

Figure 1 contrasts these tuning workflows. *Generality* calls for a policy that works across diverse kernels without kernel-specific rules. *Efficiency* calls for low end-to-end tuning time at comparable selected-kernel quality and resource budgets. *Tile-awareness* means using a program's explicit tiling structure to guide selection and designing for the compact tile configuration space, where training costs in learning-based approaches are hard to amortize. These tradeoffs leave popular tile DSLs without an out-of-the-box, end-to-end autotuner that can achieve all three. TILETUNE strikes this balance in two stages, first pruning with program-derived tile information, then efficiently measuring the survivors. Section 4 details its pruning analysis and evaluation runtime.

4 METHOD: TILETUNE

4.1 OVERVIEW

TILETUNE is designed to reduce the cost of empirical autotuning end to end, rather than only identify promising configurations. The tile program provides a common basis for ranking candidates across kernels, but the retained candidates still require compilation and measurement. Our two-stage system (Figure 4) addresses both costs: *tile-aware pruning* (Section 4.2) narrows the candidate set, using static analysis on the tile operations and data movement derived from the program; *grouped and pipelined evaluation* (Section 4.3) shares device compilation across compatible candidates and overlaps compilation with benchmarking. Among the retained candidates that pass validation, TILETUNE selects the kernel with the lowest measured latency.

Table 1: Notation for tile-aware pruning. Program facts are read from the elaborated tile program $P(c)$, device rates are measured once per GPU.

Symbol	Meaning	Source / definition
$G(c)$	blocks in the launch grid	program
$b(c)$	global-memory bytes per block	program (`T.copy` regions)
$e(c)$	global-memory accesses per block	program
$w_k(c)$	work of primitive k per block (e.g., matrix FLOPs)	program
$D(c)$	pipeline depth (buffering stages)	program
n_{SM}	number of SMs	device
$\beta,\ \rho_k$	per-SM rates of memory bytes and primitive k	device microbenchmarks
$W(c)$	SM waves, one block per SM	$\lceil G(c)/n_{\mathrm{SM}} \rceil$
$B(c),\ E(c)$	bytes and accesses across all waves	$W(c)\,b(c),\ W(c)\,e(c)$
$\Delta(c)$	blocks missing from three full waves	$\max(0,\ 3n_{\mathrm{SM}} - G(c))$
$r_{\mathrm{work}},\ r_{\mathrm{fill}}$	rank of c under each view (lower is better)	ties share the worst rank

4.2 TILE-AWARE PRUNING

For each $c \in \mathcal{C}$, we inspect the tile operations in $P(c)$ as defined in Section 2. Our key insight for tile-aware pruning is that *tile-based languages allow us to derive performance-critical factors using program analysis already supported by the compiler*. Specifically, operations such as `T.copy` expose requested global-memory regions, whose extents and enclosing loops give logical bytes accessed per block. Operations such as `T.gemm`, elementwise updates, and reductions expose the arithmetic performed per block. The launch grid and pipeline annotations provide the block count and buffering depth.

Work View. Following the Roofline model (Williams et al., 2009), one resource typically limits throughput: large GEMMs and prefill attention are compute-bound, while decoding GEMMs and attention are memory-bound. Since the pruner only needs to discard clearly slow candidates, we skip detailed models of pipelines, registers, or warp scheduling and coarsely estimate both sides (Table 1):

$$T_{\mathrm{mem}}(c) = \frac{B(c)}{\beta}, \qquad T_{\mathrm{comp}}(c) = W(c) \sum_k \frac{w_k(c)}{\rho_k}, \\ T_{\mathrm{work}}(c) = \max\big(T_{\mathrm{mem}}(c),\, T_{\mathrm{comp}}(c)\big). \tag{4}$$

The rates β and ρ_k are measured once per device. Taking the maximum lets the dominant side decide, so TILETUNE needs neither a bound classification nor kernel-specific rules.

Underfill View. Equation 4 is blind to wave effects on small grids (NVIDIA Corporation, 2023): it charges only the busiest SM at full rate, even when that SM holds a single block. In practice, several resident blocks per SM overlap their memory requests, while a lone block cannot hide memory latency. This matters most for decoding shapes, whose few output tiles leave one or two waves. We therefore add a memory-only view that inflates the bytes of grids below three waves:

$$U(c) = B(c) \left(1 + \frac{\Delta(c)}{G(c) + e(c)}\right). \tag{5}$$

The penalty vanishes at three waves and is smaller for kernels with many accesses per block, whose longer blocks already overlap more requests.

Dual-Rank Fusion. The two views order candidates lexicographically by $(T_{\mathrm{work}}(c), -D(c), E(c))$ and $(U(c), -D(c), E(c))$, preferring deeper buffering and fewer accesses at equal estimates. Since the views use different units and are both coarse, we multiply their ranks instead of weighting them:

$$\widehat{L}(c) = r_{\mathrm{work}}(c) \cdot r_{\mathrm{fill}}(c). \tag{6}$$

A candidate scores well only if neither view ranks it poorly, with no fitted weights or measurements. Following Section 2, we compile a budgeted shortlist rather than trusting the top candidate, since cache, synchronization, or register effects can still mislead the ranking. Appendix B gives the exact estimates, tie handling, and selection rule.

4.3 GROUPED AND PIPELINED EVALUATION

Pruning shrinks the candidate set, but survivors must still be compiled on the CPU and validated and timed on the GPU (Section 3.1). The CPU compile pool processes them in compile *groups*, and a dispatcher sends ready kernels to one worker queue per benchmarking GPU.

Grouped Compilation. Compiling each configuration in isolation repeats backend work. Each candidate is still specialized and lowered separately, but a group's lowered candidates are merged into one module, so code generation, CUDA compilation, and linking run once per group while still yielding distinct callable kernels (Figure 4). Grouping also cuts benchmarking overhead by loading one host module, and initializing the TVM-FFI backend once, per group. Larger groups share more work but delay when kernels become ready; Appendix A evaluates group size.

Post-Compile Filter. Device compilation exposes resource usage the ranking model cannot see. We drop kernels whose registers, shared memory, spills, or local memory exceed device limits or configured bounds, saving their validation and benchmarking but not their compilation. Other members of the group remain eligible.

Compilation–Benchmarking Overlap. The dispatcher enqueues each group's surviving kernels as soon as the group finishes, so GPUs benchmark them while later groups compile. Queues buffer ready kernels, so a busy worker never blocks compilation.

Benchmarking Worker. Each GPU's worker processes its queue sequentially: it checks correctness against a reference, runs warm-up iterations, and averages CUDA-event timings over repeated launches. A 256-MiB write flushes L2 before each start event, outside the timed interval.

5 EVALUATION

We implement TILETUNE and integrate it with TileLang, a popular tile-based GPU DSL (Wang et al., 2026) with about 7.5K GitHub stars[1]. TileLang has been used to develop kernels for DeepSeek-V4 training (DeepSeek-AI, 2026b). We evaluate TILETUNE on NVIDIA A100, H200, and B300 GPUs to answer the following research questions:

- *End-to-end tuning.* Compared with the out-of-the-box autotuner of a widely used tile-based DSL, how much tuning time does TILETUNE save, and does it still return a kernel as fast as full search (Section 5.1)?
- *Pruning in isolation.* Against learned and heuristic pruners under the same shortlist budget, how often does tile-aware pruning keep the full-search optimum, and what does its selection cost (Section 5.2)?

System ablations are reported in Appendix A.

Kernels. We evaluate five kernel families, with their configuration pools listed in Table 2. *GEMM* performs dense matrix multiplication for decoding, prefill, and feed-forward-network (FFN) shapes. *FP8 GEMM* performs low-precision matrix multiplication with the device-specific output and scaling formats described below. *Grouped GEMM* combines independent matrix multiplications with shared N, K and balanced or ragged row counts $\{M_i\}$. *FlashAttention* computes tiled attention without materializing the full attention matrix (Dao et al., 2022), covering both causal and noncausal cases. *Kimi Delta Attention (KDA)* uses channel-wise-gated linear attention in Kimi Linear (Zhang et al., 2025). We tune its intra-chunk coefficient computation rather than the full attention operator. GEMM and Grouped GEMM use BF16 inputs and outputs with FP32 accumulation. H200 FP8 GEMM uses E4M3 inputs and outputs without scale tensors. On B300, FP8 GEMM uses E4M3 inputs, packed UE8M0 scales at 128-element granularity along K, FP32 accumulation, and BF16 outputs. Native FP8 GEMM is unsupported on A100, indicated by – in the table. FlashAttention uses BF16. KDA uses BF16 inputs and outputs with FP32 gates.

Experimental Setup. We use five input shapes per kernel family, giving 25 workloads on H200 and B300 and 20 on A100, where FP8 GEMM is unsupported. Each workload uses the corresponding device-specific configuration pool in Table 2. Within each workload, all methods use the same candidate programs, inputs, correctness checks, and CUDA-event timing protocol. We use a 64-core

[1] `https://github.com/tile-ai/tilelang`, accessed September 2026.

Table 2: Kernel workloads, tuning knobs, and configuration counts on A100, H200, and B300.

Kernel	Shape	Cases	Knobs	A100	H200	B300
GEMM	(M, N, K)	Decoding/prefill/FFN	B_M, B_N, B_K, S, T, R	576	576	609
FP8 GEMM	(M, N, K)	Decoding/prefill/FFN	B_M, B_N, B_K, S, T, R	–	576	533
			S, T, V, G, O, U, W (Blackwell)			
Grouped GEMM	$(\{M_i\}, N, K)$	Balanced/ragged	B_M, B_N, B_K, S, T	576	576	576
FlashAttention	$(B, H, N_{\text{seq}}, D)$	Causal/noncausal	B_M, B_N, S, T	480	512	520
KDA	(B, H, N_{seq})	Intra-chunk	B_H, B_{DK}, S, T	234	645	513

CPU budget for compilation, with one pool of 64 compiler workers shared across benchmarking GPUs. Every candidate is checked for correctness before its latency enters the comparison.

Tuning Space. Table 2 lists the configuration pool for each kernel and device before correctness checks. B_M, B_N, B_K, S, T, R are as explained in Section 2. For FP8 GEMM, Blackwell fixes $(B_M, B_N, B_K) = (128, 256, 128)$, but also uses additional knobs V, G, O, U, W for the kernel variant, traversal group size, traversal order, TMA-store enablement, and output-store tile width, respectively. Only A100 KDA varies the head-dimension tile size B_{DK}.

5.1 End-to-end tuning

We evaluate TileTune as an end-to-end autotuning system: how long it takes from invocation until it returns a valid kernel, and how fast that kernel is compared with the full-search optimum.

Baselines. We compare against TileLang Autotune, the DSL's out-of-the-box end-to-end tuner. It exhaustively evaluates the supplied configurations and returns the fastest valid kernel it measures, so its result is the full-search optimum $c^\star$. Its standard setup uses one benchmarking GPU. For a comparison at the same GPU budget, we also run it with our four-GPU adaptation, which distributes benchmarking across workers but otherwise retains the baseline's compilation and search procedure. TileTune uses four GPUs with tile-aware pruning, grouped compilation, and compilation–benchmarking overlap.

Tuning time. Figure 5 shows end-to-end speedups on all three GPUs; we discuss H200 in detail. Across its 25 workloads, TileTune takes 3,384 s, versus 13,609 s and 7,308 s for the one- and four-GPU baselines, a $4.02\times$ and $2.16\times$ speedup. FP8 GEMM gains the most ($7.77\times$ over the one-GPU baseline) and FlashAttention the least ($2.70\times$), because the post-compile filter rejects many of its shortlisted candidates only after their compilation has been paid. A100 and B300 show similar gains, $5.07\times$ and $5.35\times$ over the one-GPU baseline.

Kernel quality. Speed matters only if the returned kernel is as good as the one full search finds. On every H200 workload, the full-search optimum of both baselines is in TileTune's shortlist, passes the post-compile filter, and is benchmarked, so TileTune can return it. Section 5.2 analyzes pruning quality in detail.

5.2 Pruning in isolation

Baselines. We compare TileTune's pruner with Roller (Zhu et al., 2022) and an XGBoost ranker (Chen & Guestrin, 2016), in both kernel quality and selection cost. For Roller, we use TileLang's Python implementation of its analytical tiling policy, with kernel-specific adapters for all five families. XGBoost trains on two shapes and validates on a third per family, disjoint from the five held-out workloads. All methods rank the same candidate pools with a strict 50% shortlist budget. We compare the retained configurations against saved full-search measurements, without applying the post-compile filter. Because every method keeps the same number of candidates, the downstream compilation and benchmarking cost is roughly the same, and the selection cost is what differs in efficiency.

Kernel quality. Table 3 reports how often each shortlist retains both full-search optima. TileTune retains both on every workload of all three GPUs, while XGBoost and Roller each miss optima on every GPU. On H200, XGBoost retains both on 20 of 25 workloads and Roller on 10. Retention alone hides how much a miss costs, so we also compute the geometric mean of $L(c^\star)/L(\hat{c})$, where

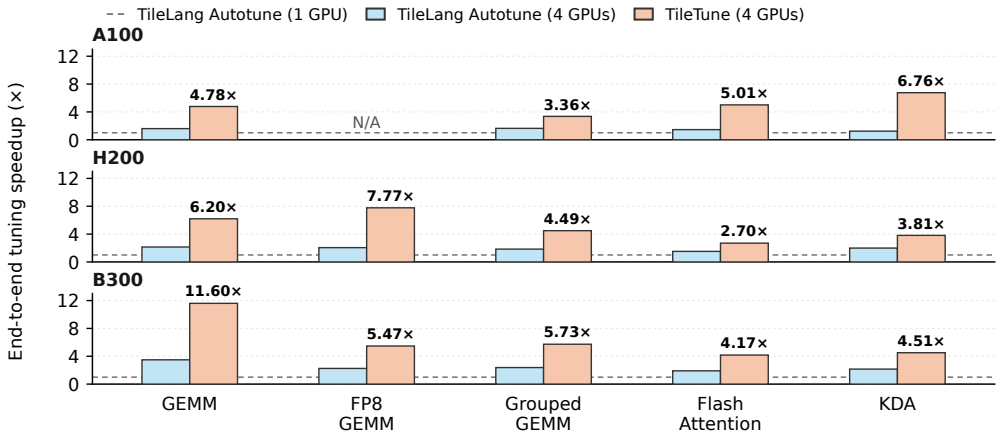

Figure 5: End-to-end tuning speedup over one-GPU TileLang Autotune (dashed line at $1\times$). Each bar sums the tuning time of a family's five workloads, labels give TILETUNE's speedup. FP8 GEMM is unsupported on A100.

Table 3: Exact-oracle retention under a strict 50% shortlist budget. Each cell counts workloads retaining both full-search optima. Cost is the total selection time over all workloads: static analysis for TILETUNE, label collection and fitting for XGBoost, and ranking for Roller.

	A100			H200			B300		
Kernel	TILETUNE	XGBoost	Roller	TILETUNE	XGBoost	Roller	TILETUNE	XGBoost	Roller
GEMM	5/5	5/5	3/5	5/5	5/5	1/5	5/5	5/5	3/5
FP8 GEMM	–	–	–	5/5	5/5	1/5	5/5	3/5	1/5
Grouped GEMM	5/5	4/5	5/5	5/5	5/5	3/5	5/5	5/5	4/5
FlashAttention	5/5	4/5	4/5	5/5	1/5	0/5	5/5	1/5	2/5
KDA	5/5	5/5	5/5	5/5	4/5	5/5	5/5	5/5	5/5
Overall	20/20	18/20	17/20	25/25	20/25	10/25	25/25	19/25	15/25
Cost (s)	195	659	26	246	855	31	173	796	18

$\hat{c}$ is the best configuration inside the shortlist. On H200 it is 100.00% for TILETUNE, 99.86% for XGBoost (worst 97.03%), and 91.13% for Roller (worst 72.43%).

Selection cost. The last row of Table 3 gives each method's selection cost. On H200, TILETUNE analyzes all 25 pools in 246.3 s and Roller ranks them in 30.8 s, both on the CPU. XGBoost costs 854.7 s, of which 850.1 s is spent collecting training and validation labels on the GPU. Roller is thus the cheapest but misses on 15 of 25 workloads; XGBoost is the most expensive yet still misses on 5 because it cannot capture enough labels. TILETUNE is the only method that keeps every optimum, and it needs neither training labels nor kernel-specific adapters.

6 CONCLUSION

TILETUNE balances generality, efficiency, and tile-awareness in tile-centric GPU tuning, bridging empirical search and specialized heuristics. Shared static tile-program analysis prunes unpromising configurations without requiring user-written selection rules for each kernel, while measurements still determine the final choice. A parallel task-processing runtime reduces the cost of that remaining search through efficient dispatch, grouped compilation, and overlapped compilation and benchmarking. Across kernels and hardware platforms, TILETUNE achieves 4.02–$5.35\times$ end-to-end tuning speedup over the one-GPU TileLang search-based baseline. The intended balance rests on explicit tile-program reasoning and is assessed through tuning time, selected-kernel quality, and coverage under shared defaults, rather than through speedup alone.

AI USE STATEMENT

In this work, we did not use generative AI tools for any task that requires disclosure: they were not used to develop models or conceptual frameworks, formulate or prove claims, propose hypotheses, design experiments, implement methods, generate or process data, or interpret results. Additionally, we used generative AI tools as a writing assistant to edit the paper for readability, including rephrasing and polishing author-written text. We have reviewed all AI-assisted text, and all authors checked the edited passages against our implementation and experimental results for technical accuracy. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

All authors have read and adhere to the ICLR Code of Ethics. This work studies autotuning for GPU kernels; it does not involve human subjects, personal or sensitive data, or the release of new datasets, and all workloads use synthetic kernel inputs. The main broader-impact consideration is the compute and energy cost of autotuning. By reducing end-to-end tuning time, TILETUNE lowers the GPU and CPU resources spent on kernel search, and we report tuning cost alongside kernel quality so that these savings can be assessed. We are not aware of harmful applications specific to this work beyond those of faster GPU software in general.

REFERENCES

Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pp. 303–316, 2014. doi: 10.1145/2628071.2628092.

Apache TVM Developers. Apache TVM AutoTVM: Tuning convolutional neural networks for CUDA. Apache TVM v0.9.0 tutorial, 2022a. URL https://github.com/apache/tvm/blob/v0.9.0/gallery/how_to/tune_with_autotvm/tune_relay_cuda.py.

Apache TVM Developers. Apache TVM: CUDA tensor core dense schedule. Apache TVM v0.9.0 source code, 2022b. URL https://github.com/apache/tvm/blob/v0.9.0/python/tvm/topi/cuda/dense_tensorcore.py.

Apache TVM Developers. Apache TVM AutoTVM: XGBTuner. Apache TVM v0.9.0 source code, 2022c. URL https://github.com/apache/tvm/blob/v0.9.0/python/tvm/autotvm/tuner/xgboost_tuner.py.

Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 785–794, 2016. doi: 10.1145/2939672.2939785.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, 2018a. URL https://www.usenix.org/conference/osdi18/presentation/chen.

Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, volume 31, 2018b. URL https://arxiv.org/abs/1805.08166.

Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning, 2023. URL https://arxiv.org/abs/2307.08691.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness, 2022. URL https://arxiv.org/abs/2205.14135v2.

DeepSeek-AI. DeepGEMM. GitHub repository, 2026a. URL https://github.com/deepseek-ai/DeepGEMM/tree/78b69000794d0937b47ae3387eff7663410264d1. Inspected revision 78b6900.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026b. URL https://arxiv.org/abs/2606.19348.

Cedric Nugteren and Valeriu Codreanu. Cltune: A generic auto-tuner for opencl kernels. In *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, pp. 195–202, 2015. doi: 10.1109/MCSOC.2015.10.

NVIDIA. cuBLASLt: cublasLtMatmulAlgoGetHeuristic. CUDA Toolkit Documentation, 2026a. URL https://docs.nvidia.com/cuda/cublas/index.html#cublasltmatmulalgogetheuristic.

NVIDIA. CUTLASS GEMM heuristics. NVIDIA CUTLASS Documentation, 2026b. URL https://docs.nvidia.com/cutlass/latest/media/docs/cpp/heuristics.html.

NVIDIA Corporation. Matrix multiplication background user's guide. NVIDIA Documentation, 2023. URL https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 519–530, 2013. doi: 10.1145/2491956.2462176.

Burkhard Ringlein, Thomas Parnell, and Radu Stoica. GPU performance portability needs autotuning, 2025a. URL https://arxiv.org/abs/2505.03780v3.

Burkhard Ringlein, Jan van Lunteren, Radu Stoica, and Thomas Parnell. The anatomy of a Triton attention kernel, 2025b. URL https://arxiv.org/abs/2511.11581v1.

Richard Schoonhoven, Ben van Werkhoven, and Kees Joost Batenburg. Benchmarking optimization algorithms for auto-tuning GPU kernels. *IEEE Transactions on Evolutionary Computation*, 27(3): 550–564, 2023. doi: 10.1109/TEVC.2022.3210654.

Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. Tensor program optimization with probabilistic programs. In *Advances in Neural Information Processing Systems*, volume 35, pp. 35515–35528, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/e894eafae43e68b4c8dfdacf742bcbf3-Abstract-Conference.html.

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

Ryan Swann, Muhammad Osama, Xiaohu Guo, Bryant Nelson, Lixun Zhang, Alex Brown, Yen Ong, Ali Yazdani, Sean Siddens, Ganesh Dasika, and Alex Underwood. tritonblas: Triton-based analytical approach for gemm kernel parameter selection, 2025. URL https://arxiv.org/abs/2512.04226.

Philippe Tillet, H. T. Kung, and David Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 10–19, 2019. doi: 10.1145/3315508.3329973.

Lei Wang, Yu Cheng, Yining Shi, Zhiwen Mo, Zhengju Tang, Wenhao Xie, Tong Wu, Lingxiao Ma, Yuqing Xia, Jilong Xue, Fan Yang, and Zhi Yang. TileLang: Bridge programmability and performance in modern neural kernels. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/76fb92288bf90360c527efb0d1c2aba6-Abstract-Conference.html.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009. doi: 10.1145/1498765.1498785.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. FlashInfer: Efficient and customizable attention engine for LLM inference serving. In *Proceedings of the Eighth Conference on Machine Learning and Systems*, 2025. URL https://proceedings.mlsys.org/paper_files/paper/2025/hash/dbf02b21d77409a2db30e56866a8ab3a-Abstract-Conference.html.

Yi Zhai, Yu Zhang, Shuo Liu, Xiaomeng Chu, Jie Peng, Jianmin Ji, and Yanyong Zhang. Tlp: A deep learning-based cost model for tensor program tuning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 833–845, 2023. doi: 10.1145/3575693.3575737.

Kaixuan Zhang, Chutong Ding, Shiyou Qian, Luping Wang, Jian Cao, Guangtao Xue, Cheng Huang, Guodong Yang, and Liping Zhang. WaveTune: Wave-aware bilinear modeling for efficient gpu kernel auto-tuning, 2026. URL https://arxiv.org/abs/2604.10187v1.

Yu Zhang, Zongyu Lin, Xingcheng Yao, Jiaxi Hu, Fanqing Meng, Chengyin Liu, et al. Kimi Linear: An expressive, efficient attention architecture, 2025. URL https://arxiv.org/abs/2510.26692.

Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 863–879, 2020. URL https://www.usenix.org/conference/osdi20/presentation/zheng.

Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. ROLLER: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 233–248. USENIX Association, 2022. URL https://www.usenix.org/conference/osdi22/presentation/zhu.

A Ablation Study

Benefit Breakdown. We use the GEMM kernel with $(M, N, K) = (4096, 4096, 4096)$ on H200 as the example for this part. Figure 6(a) adds one technique of TileTune at a time. Grouped compilation of eight candidates reduces end-to-end time from 394.6 to 236.2 s. Overlapping compilation and benchmarking lowers it to 211.9 s, four benchmarking GPUs lower it further to 126.8 s. The post-compile filter rejects 80 candidates, reducing time to 115.9 s. Tile-aware pruning instead selects 288 candidates before compilation, of which 208 survive the post-compile filter, reducing time to 66.83 s. It halves the number of compile groups from 72 to 36 and cuts the compile time from 100.04 to 45.50 s. The complete gain is $5.90\times$ faster.

Grouped compilation size. We sweep group sizes from 1 to 32 on the same 576-configuration H200 GEMM, using eight compiler workers and one benchmarking GPU. Each size is measured once. This isolated sweep disables pruning, the post-compile filter, and compilation–benchmarking overlap, unlike the system breakdown in Figure 6(a). Figure 6(b) shows that grouping eight candidates cuts compilation from 341.0 to 154.2 s and benchmark-stage wall time from 289.2 to 131.9 s, reducing total evaluation time from 630.2 to 286.2 s. Compilation falls because, although each candidate is lowered separately, candidates in a group share code generation, CUDA compilation, and linking. Benchmarking falls because each group produces one host module instead of eight, so the worker loads fewer modules and triggers the lazy initialization of the TVM-FFI backend once per group rather than once per candidate. Kernel execution itself is unaffected: aggregate CUDA-event time stays near 31 s at every group size, so the savings come from host-side module loading, lookup, and

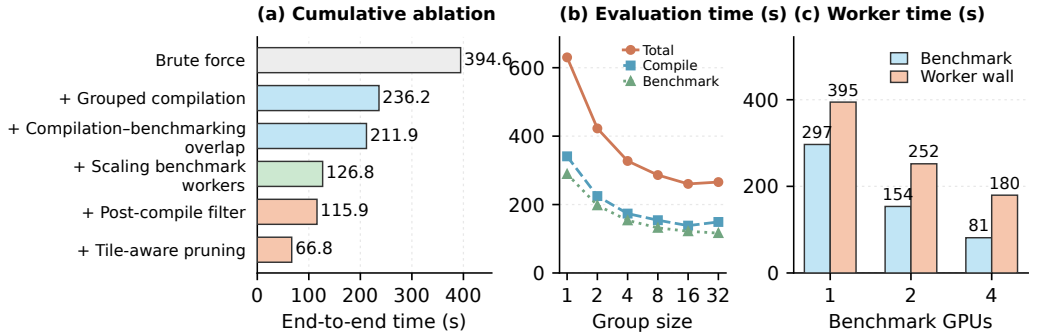

Figure 6: Ablation studies of GEMM on H200. (a) End-to-end time as each technique is added. (b) Total evaluation, compilation, and benchmarking wall time across group sizes. (c) Benchmark and worker-wall time across numbers of benchmarking GPUs.

initialization overhead in the benchmark span. Group size 16 minimizes total time at 260.4 s. At size 32, benchmarking falls further, but compilation rises and total time increases to 265.7 s.

Benchmarking worker scaling. To isolate GPU parallelism, we separately benchmark all 576 candidates with one, two, or four GPUs, keeping 64 compiler workers but disabling grouping, overlap, and post-compile filter. Figure 6(c) shows that the benchmark time falls from 297 to 154 and 81 s, while compilation stays near 91 s. Four GPUs give a $3.65\times$ benchmark-phase speedup but only a $2.20\times$ worker-wall speedup (395 s to 180 s). This not only shows the effectiveness of parallel benchmarking, but also highlights the importance of compilation-benchmarking overlap.

B Tile-Aware Pruning Details

Program-derived access facts. The pruner elaborates each $P(c)$, $c \in \mathcal{C}$, into a `PrimFunc` and reads logical global-memory access regions before layout inference and lowering. Let $\mathcal{A}(c)$ be its reads and writes. An access a contributes the byte extent $b_a(c)$ of its requested region and a visit count $v_a(c)$ from its enclosing loops. The analysis uses resolved upper bounds for variable loop counts. Masked accesses and partial tiles retain their requested extents; these are logical quantities rather than measured DRAM traffic. Read-only integer input metadata can resolve symbolic bounds and extents without introducing an operator-specific selection rule.

Program-derived compute facts. The same elaborated `PrimFunc` gives a logical compute ledger. Each tile operation contributes work of one primitive kind k: matrix FLOPs for `T.gemm`, scalar operations for elementwise updates, copies, and fills, exponentials and reciprocal square roots, and sum or max reductions. Scalar work counts the nodes of the value expression tree, excluding buffer-index arithmetic. A reduction counts source minus destination elements. Parallel extents and serial loop visits are multiplied once; predicated work keeps its full requested extent. Let $w_k(c)$ be the resulting work per block. An unsupported operation or unresolved count leaves $w_k(c)$ unknown rather than zero.

Memory work and launch underfill. Let $G(c)$ be the block count in the launch grid, $n_{\mathrm{SM}}(h)$ the target's SM count, and $e(c)$ the number of logical nonempty accesses per block, counting loop visits. Define the wave factor, logical byte-work, and access count as

$$
\begin{aligned}
W(c) &= \left\lceil \frac{G(c)}{n_{\mathrm{SM}}(h)} \right\rceil, \\
e(c) &= \sum_{\substack{a \in \mathcal{A}(c) \\ b_a(c) > 0}} v_a(c), \\
B(c) &= W(c) \sum_{a \in \mathcal{A}(c)} b_a(c) v_a(c), \qquad E(c) = W(c) e(c).
\end{aligned}
\tag{7}
$$

The one-CTA-per-SM wave factor accounts for launch size but does not estimate physical occupancy; $E(c)$ counts logical requests rather than coalesced memory transactions. We adjust byte-work for underfilled launches:

$$\Delta(c) = \max\bigl(0, 3n_{\mathrm{SM}}(h) - G(c)\bigr),$$
$$U(c) = B(c) + \left\lceil \frac{B(c)\Delta(c)}{G(c) + e(c)} \right\rceil. \tag{8}$$

The penalty vanishes at three SM waves; more per-block accesses temper its size. This adjustment orders candidates rather than predicting occupancy or latency.

Let $D(c)$ be the maximum positive pipeline depth exposed by loop annotations or recognized shared-memory ring buffers, with a minimum of one. Partially resolved depth is reported as a lower bound.

Primitive profile. For each device we measure, once, the per-SM per-cycle rates β of global-memory bytes and ρ_k of each primitive kind: tensor-core matrix FLOPs for the instruction and data types used by the kernel, scalar, exponential and reciprocal-square-root operations, and FP32 sum and max reductions. These are logical rates from small microbenchmarks, not chip-wide peaks, and the profile is reused unchanged for every kernel and workload on that device. A matrix rate applies only when its instruction and input/accumulator types match the kernel; a mismatch or a missing rate for nonzero work leaves the score unknown instead of falling back to memory.

Primary terms of the two views. Each view $v \in \{\mathrm{work}, \mathrm{fill}\}$ orders candidates first by a primary term $X_v(c)$, expressed in memory-equivalent bytes. For the work view, the compute estimate sums the service time of every primitive kind and scales it by the wave factor, $T_{\mathrm{comp}}(c) = W(c) \sum_k w_k(c)/\rho_k$. Rather than compare cycles, we convert it to bytes at the fixed bandwidth and take the larger side:

$$X_{\mathrm{work}}(c) = \max\Bigl(B(c),\ \lceil \beta\, T_{\mathrm{comp}}(c) \rceil\Bigr). \tag{9}$$

Up to the ceiling, this is $\beta \max(T_{\mathrm{mem}}, T_{\mathrm{comp}})$ from Equation 4; exact rational arithmetic rounds upward by less than one byte. Compute and memory are assumed to overlap fully, and primitive kinds are assumed to serialize. The view uses no pool normalization, occupancy estimate, pipeline timing, or bound classification. The underfill view keeps the memory side only, with $X_{\mathrm{fill}}(c) = U(c)$ from Equation 8; it does not adjust the compute estimate of the work view.

Integer encoding. Each view orders candidates lexicographically by $(X_v(c), -D(c), E(c))$: smaller primary term first, then deeper buffering, then fewer logical accesses. We encode this order and its ties as an integer:

$$\sigma_v(c) = Q\frac{X_v(c)(X_v(c)+1)}{2} + (Q - D(c))(X_v(c)+1) + E(c), \qquad Q = 2^{16} - 1. \tag{10}$$

Each nonempty access contributes at least one byte, so $0 \le E(c) \le B(c) \le X_v(c)$ for both views. Requiring $1 \le D(c) \le Q$ keeps lower-priority terms from reversing a higher-priority comparison. The triangular term assigns nonoverlapping integer bands to successive values of $X_v(c)$; integer arithmetic also preserves ties without floating-point rounding.

Rank fusion. Let $\mathcal{E}$ contain candidates whose work and underfill scores are both resolved and that are not flagged by pre-lowering resource checks. For each view $v \in \{\mathrm{work}, \mathrm{fill}\}$ with score σ_v, we assign every tie group its last rank,

$$r_v(c) = \left|\{u \in \mathcal{E} : \sigma_v(u) \le \sigma_v(c)\}\right|, \tag{11}$$

and define $\widehat{L}(c) = r_{\mathrm{work}}(c)\, r_{\mathrm{fill}}(c)$ for $c \in \mathcal{E}$. Because the ranks depend on $\mathcal{E}$, $\widehat{L}$ is recomputed for each candidate pool and never cached per kernel. Tail ranks are conservative: tied candidates share the worse rank, and equal products stay tied.

Budgeted selection. Let $N = |\mathcal{C}|$ and choose $k = \lfloor \alpha N \rfloor \ge 1$, with $0 < \alpha \le 1$. We rank $\mathcal{E}$ by $\widehat{L}$, again assign each tie group its last rank, and retain only groups within the budget:

$$\hat{r}(c) = \left|\left\{u \in \mathcal{E} : \widehat{L}(u) \le \widehat{L}(c)\right\}\right|, \qquad \mathcal{R} = \{c \in \mathcal{E} : \hat{r}(c) \le k\}. \tag{12}$$

A group crossing k is excluded entirely, so the shortlist may contain fewer than k candidates or be empty. The original N includes unscored candidates and analysis failures. All evaluations use $\alpha = 0.5$. An alternative explicit `top_k` mode includes the boundary group by default unless strict selection is enabled.

The shortlist is frozen before lowering. Later compilation, resource-check, or measurement failures do not trigger replacement candidates. If no valid measured candidate remains, tuning reports failure; otherwise it returns

$$\hat{c} = \arg\min_{c \in \mathcal{R} \cap \mathcal{V}} L(c), \qquad \mathcal{R} \cap \mathcal{V} \neq \emptyset. \tag{13}$$

Unresolved memory effects, byte extents, visit counts, compute work, grid size, SM count, or profile rates give an unknown score and exclude a candidate from pure ranked selection. Uncertainty in detailed scheduling or storage estimates alone does not invalidate resolved work inputs. Compiler-dependent resource checks remain separate from the ranking and run when evidence becomes available.

All candidates incur elaboration and analysis, including those rejected by pruning; only the shortlist proceeds to lowering and compilation. Beyond the one-time primitive profile, the pruner needs no per-kernel calibration, training data, cache simulation, pipeline timing, or occupancy prediction. Detailed dependency, tile-liveness, and shared-storage reports are disabled by default, although backward tile propagation runs when required for a strict accumulator-resource check. Omitted cache, synchronization, register, and scheduling effects can misorder candidates, so retaining the full-search winner $c^\star$ is not guaranteed.

C LIMITATIONS

Pruning accuracy. Tile-aware pruning relies on coarse static estimates of memory traffic, compute work, and launch underfill. It does not model cache behavior, synchronization, register allocation, or warp scheduling, and compiler decisions or data-dependent execution can change a candidate's performance. Pruning therefore offers no general guarantee of retaining the fastest configuration. The 50% shortlist budget retained every full-search optimum in our evaluation, but a smaller budget or a kernel dominated by unmodeled effects may lose it.

Evaluation scope. We evaluate five kernel families on three NVIDIA GPUs. Three of the families are GEMM variants, so this suite does not establish generality over arbitrary tile kernels. TILETUNE is implemented for TileLang with a CUDA backend; other tile DSLs and GPU vendors are not evaluated. Our comparison with learning-based tuning targets models built on loop- and schedule-level representations; learned models that use tile-level features may narrow this gap.

System assumptions. Grouped compilation requires candidates with compatible compiler contexts, and each candidate is still lowered separately. Multi-GPU benchmarking assigns kernels to workers statically and assumes comparable measurement devices. Group size and worker count are user-set parameters rather than automatically tuned. When pruning leaves few candidates or compilation caches are warm, grouping and overlap provide smaller gains. Finally, the reported speedups are aggregated per kernel family and GPU; individual workloads may benefit less.