Evaluating CUDA Tile for AI Workloads on Hopper and Blackwell GPUs

Divakar Kumar Yadav
University of Wisconsin–Milwaukee
Milwaukee, WI, USA
dkyadav@uwm.edu

Tian Zhao
University of Wisconsin–Milwaukee
Milwaukee, WI, USA
tzhao@uwm.edu

Deepak Kumar
Illinois Institute of Technology
Chicago, IL, USA
dkumar15@hawk.illinoistech.edu

***Abstract*—NVIDIA's CUDA Tile (CuTile) introduces a Python-based, tile-centric abstraction for GPU kernel development that aims to simplify programming while retaining Tensor Core and Tensor Memory Accelerator (TMA) efficiency on modern GPUs. We present the first independent, cross-architecture evaluation of CuTile against established approaches – cuBLAS, Triton, WMMA, and raw SIMT – on three NVIDIA GPUs spanning Hopper and Blackwell: H100 NVL, B200, and RTX PRO 6000 Blackwell Server Edition. We benchmark representative AI workloads, including GEMM, fused multi-head attention, and end-to-end LLM inference in BF16/FP16 precision, to assess both performance and portability.**

Our results show that CuTile's effectiveness is strongly workload- and architecture-dependent. On datacenter-class Blackwell (B200), CuTile achieves up to 1,007 TFLOP/s for fused attention, outperforming FlashAttention-2 by 2.5× while requiring only 60 lines of Python kernel code. For GEMM, CuTile reaches 52–79% of cuBLAS performance in 22 lines of code (versus 123 for WMMA), making it a practical replacement for hand-written CUDA kernels but not yet for vendor-optimized libraries. However, the same CuTile attention kernel achieves only 53% of FlashAttention-2 throughput on RTX PRO 6000 (`sm_120`), exposing significant cross-architecture optimization gaps. In contrast, Triton sustains 62–101% of cuBLAS performance across all tested platforms without architecture-specific tuning, demonstrating substantially stronger portability.

***Index Terms*—CUDA Tile, GPU Kernel Abstractions, Triton, cuBLAS, Tensor Cores, GEMM, FlashAttention, LLM Inference, Hopper, Blackwell, Code Productivity, Adoption Guide**

I. Introduction

Writing high-performance GPU kernels remains challenging. The transformer architecture [20] has driven language models to hundreds of billions of parameters [17], [21], [22], and modern serving systems—including vLLM [5], TensorRT-LLM [6], and Megatron-LM [23]—rely on hand-optimized kernels such as FlashAttention-2 [2] and CUTLASS [4] to approach peak hardware utilization. These kernels typically span hundreds to thousands of lines of architecture-specific CUDA code and often require extensive retuning or redesign with each new GPU generation.

In late 2025, NVIDIA introduced CUDA Tile (CuTile), a Python-based, tile-centric programming model [7] intended to reduce this engineering burden. CuTile abstracts warps, registers, and shared memory behind high-level primitives (`ct.load`, `ct.mma`, `ct.store`) while still leveraging Tensor Cores and the Tensor Memory Accelerator (TMA) on Blackwell GPUs. The promise is compelling: express a Tensor Core kernel in tens of lines of Python rather than hundreds of lines of CUDA C++. Whether this promise holds in practice, however, remains an open question.

This paper addresses the practical question faced by GPU kernel developers: *Should one switch to CuTile, and if so, for which workloads, on which GPUs, and with what trade-offs?*

We evaluate CuTile head-to-head against established alternatives spanning vendor libraries, high-level DSLs, and hand-written CUDA:

1) **cuBLAS** [10] — NVIDIA's closed-source, auto-tuned BLAS library (1 LOC; performance upper bound)
2) **Triton** [8] — OpenAI's Python-based GPU compiler and leading open-source alternative (∼53–62 LOC)
3) **CUDA Tile (CuTile)** [7] — NVIDIA's tile-centric Python DSL (∼22–60 LOC; *subject of this study*)
4) **WMMA** [9] — Hand-written CUDA using the Warp Matrix Multiply-Accumulate API (∼123 LOC)
5) **Raw SIMT** — Hand-written CUDA without Tensor Cores (∼32 LOC; baseline)

We conduct experiments on three GPUs spanning two architecture generations: NVIDIA H100 NVL (Hopper, `sm_90`), RTX PRO 6000 Blackwell Server Edition (`sm_120`), and NVIDIA B200 (Blackwell, `sm_100`). CuTile is supported only on Blackwell GPUs, whereas the other approaches run on all three platforms, providing a controlled setting that isolates CuTile's impact from underlying architectural improvements.

A. Summary of Findings

Our evaluation yields a clear but nuanced set of conclusions:

- *Switch for fused attention on datacenter Blackwell (B200).* CuTile achieves up to 1,007 TFLOP/s—2.5× higher than FlashAttention-2—using 60 lines of Python code, representing the strongest single result observed in this study.
- *Prefer CuTile over WMMA for GEMM on Blackwell GPUs.* CuTile delivers 1.5–5.0× higher throughput than WMMA while requiring 5.6× less code (22 vs. 123 lines), making it a compelling replacement for hand-written WMMA kernels.
- *Do not replace cuBLAS for standard GEMM.* CuTile achieves 52–79% of cuBLAS throughput; for workloads

TABLE I
HARDWARE PLATFORMS USED IN THIS STUDY.

	H100 NVL	RTX PRO 6000	B200
Architecture	Hopper	Blackwell	Blackwell
Compute Capability	sm_90	sm_120	sm_100
Streaming Multiprocessors	132	188	148
VRAM (GB)	94	96	179
TDP (W)	400	600	1,000
Max Shared Mem/Block (KB)	228	48	227
SM Clock (MHz)	–	2,430	1,965
CuTile Support	No	Yes	Yes

already served by `torch.matmul` or cuBLAS, there is little performance incentive to switch.

- *Prefer Triton when cross-architecture portability is required.* Triton sustains 62–101% of cuBLAS performance across all tested GPUs, including Hopper, without architecture-specific tuning.
- *Exercise caution on workstation-class Blackwell (sm_120).* On the RTX PRO 6000, CuTile fused attention reaches only 53% of FlashAttention-2 throughput, revealing substantial compiler immaturity relative to datacenter Blackwell.

II. Background: CuTile and Its Alternatives

A. GPU Architectures Under Evaluation

Table I summarizes the three GPU platforms evaluated in this study. The H100 NVL (Hopper, `sm_90`) [18] represents the current datacenter standard, featuring 132 streaming multiprocessors (SMs) and WGMMA-class Tensor Cores. The B200 (`sm_100`) [19] is NVIDIA's next-generation datacenter GPU with 148 SMs and the new Blackwell Tensor Core architecture. The RTX PRO 6000 Blackwell Server Edition (`sm_120`) is a professional workstation GPU with 188 SMs and a distinct Blackwell variant that differs from B200 in both memory hierarchy and shared-memory configuration.

B. CuTile's Capabilities

CUDA Tile (CuTile) [7] introduces a tile-centric abstraction for GPU kernel programming via the `cuda.tile` Python package. Rather than exposing warps, registers, and shared memory explicitly, CuTile allows developers to express computation in terms of logical tiles while still targeting Tensor Cores and the Tensor Memory Accelerator (TMA) on supported GPUs. Its core primitives include:

- `ct.load` / `ct.store` — TMA-backed memory operations that abstract global-to-shared memory transfers
- `ct.mma` — Tensor Core matrix multiply–accumulate on logical tiles
- `@ct.kernel` with `ByTarget` — a mechanism for architecture-specific tuning (e.g., CTA clustering)

CuTile's fused attention kernel implements the online softmax algorithm [32], which also underpins FlashAttention-2. A key constraint for adoption is that CuTile requires a Blackwell GPU (compute capability $\geq$ 10.0) and the `tileiras` compiler introduced in CUDA Toolkit 13.1. CuTile does not support Hopper (`sm_90`) or earlier architectures, in contrast to Triton, which targets a broader range of recent NVIDIA GPUs. As a result, CuTile cannot currently serve as a single, portable kernel development framework for heterogeneous GPU fleets.

C. Alternatives to CuTile

Figure 1 situates CuTile within the broader performance–productivity design space for both GEMM and fused attention. For developers considering CuTile, the central question is whether it occupies a region of this design space that is not already covered by existing approaches, including cuBLAS, Triton, WMMA, cuDNN, and FlashAttention-2.

III. Related Work: What CuTile Aims to Replace

A. Hand-Optimized GPU Kernels (Status Quo)

FlashAttention [1] and FlashAttention-2 [2] introduced tiled, IO-aware exact attention that avoids materializing the $O(N^2)$ attention matrix, achieving near-peak Tensor Core utilization through carefully hand-optimized CUDA kernels. FlashDecoding [3] extended these techniques to the autoregressive decode phase. While these kernels deliver state-of-the-art performance, they consist of thousands of lines of architecture-specific CUDA code and require substantial retuning across GPU generations—precisely the engineering burden CuTile seeks to reduce.

For GEMM, CUTLASS [4] provides highly optimized C++ templates that form the backbone of production inference systems such as vLLM [5] and TensorRT-LLM [6]. CUTLASS 3.x added support for Hopper features including TMA and WGMMA, but effective use still requires developers to manually specify warp-level tile shapes, shared-memory layouts, and epilogue fusions. CuTile's `ct.mma` and `ct.load`/`ct.store` primitives explicitly target this complexity by abstracting warp mapping and memory movement while retaining access to Tensor Cores and TMA.

B. Higher-Level Programming Models

Triton [8] is the most closely related alternative to CuTile. It provides a Python-based DSL whose kernels are JIT-compiled via an MLIR-based compiler, with the `tl.dot` primitive mapping automatically to Tensor Core instructions. Prior work shows that Triton kernels typically trail hand-optimized CUTLASS by 5–15% on memory-bound workloads [5], but Triton supports all recent NVIDIA architectures, making it significantly more portable than CuTile.

Schedule-based frameworks such as TVM and TensorIR [13], [14] generate optimized kernels from high-level tensor expressions, while systems like Roller [15] and AKG [16] further automate the search over implementation spaces. Halide [26] pioneered the separation of algorithm and schedule, influencing many of these designs. Despite strong research results, these systems have not displaced hand-optimized CUDA, CUTLASS, or FlashAttention in production inference engines.

NVIDIA's cuDNN [25] offers vendor-optimized deep learning primitives, including fused attention, but remains closed-source and does not permit kernel-level customization. At a

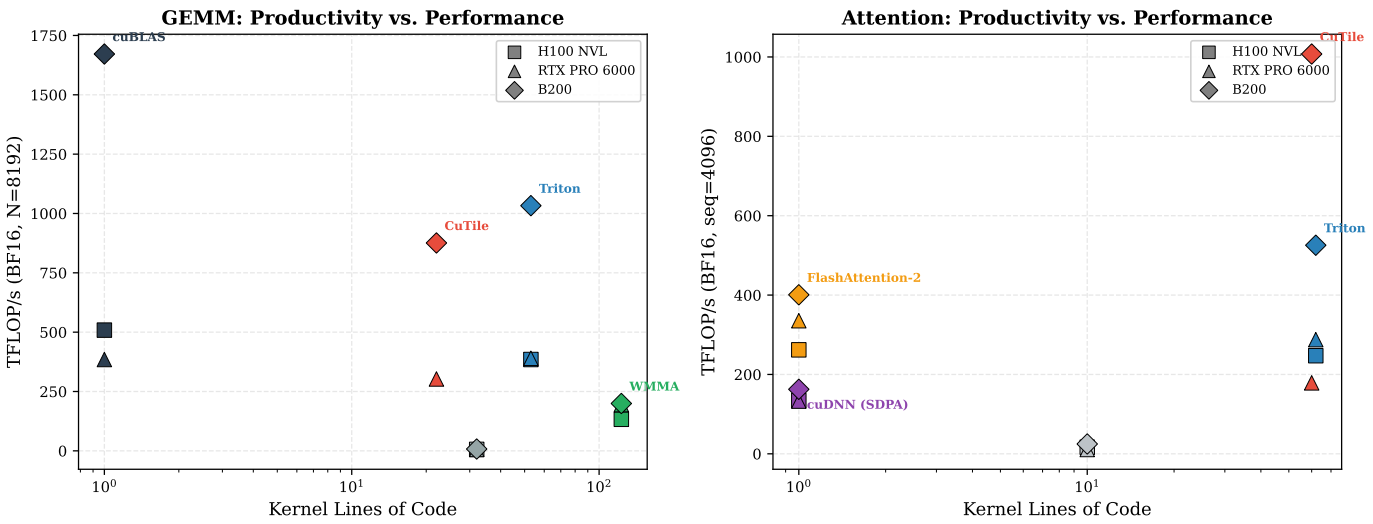

Fig. 1. Performance–productivity frontier for GEMM (left, N=8192) and attention (right, seq=4096). Each point represents one abstraction on one GPU. CuTile (red diamonds) offers high productivity but exhibits GPU-dependent performance, while Triton (blue triangles) provides the most consistent portability–performance balance.

lower abstraction level, the WMMA API [9] exposes Tensor Cores directly at the warp level, requiring explicit management of threads, registers, and shared memory. CuTile differs from both approaches by attempting to fully abstract warp-level programming and memory tiling while remaining an officially supported NVIDIA programming model—effectively positioning it as NVIDIA's response to Triton.

C. Cross-Architecture Performance Studies

GPU kernel benchmarking has a long history, from early dense linear algebra studies [27] to modern architecture-specific tuning guides [11], [12]. These studies consistently show that hand-tuned kernels suffer 20–50% performance degradation when moved across architectures without retuning.

To our knowledge, no prior work has evaluated CuTile's performance portability across GPU architectures. This paper provides the first empirical study of CuTile across two Blackwell variants (`sm_100` and `sm_120`) and benchmarks it against abstractions—such as cuBLAS and Triton—that span both Hopper and Blackwell generations.

IV. Methodology

A. Workloads and Microbenchmarks

We evaluate three kernel families that dominate transformer training and inference: GEMM, fused multi-head attention (FMHA), and end-to-end LLM inference. All experiments use BF16 or FP16 precision and identical mathematical formulations across implementations.

a) GEMM.: We benchmark dense matrix multiplication with shapes representative of both attention and feed-forward network (FFN) layers in transformer models:

- Square: $M = N = K \in \{4096, 8192, 12288, 16384\}$
- Rectangular (LLaMA-7B FFN): $(M, K, N) \in \{2048, 4096\} \times \{(4096, 11008), (11008, 4096)\}$

We compare 5 implementations spanning the performance productivity spectrum:

- **cuBLAS** `torch.matmul` (vendor-optimized baseline)
- **Triton** with autotuning over 18 configurations
- **CuTile** using `ct.mma`
- **WMMA** using `nvcuda::wmma`, double-buffered `cp.async`, `NUM_STAGES=2`
- **Raw SIMT** using scalar FMA instructions without Tensor Cores

b) Fused Multi-Head Attention (FMHA).: We evaluate scaled dot-product attention with causal masking under the standard multi-head attention configuration of LLaMA-7B (32 heads, d=128). Although many modern systems use grouped-query or multi-query attention for decode efficiency, we focus on standard MHA to isolate compute-bound behavior during prefill.

Experiments use batch size 8 and sequence lengths $\{512, 1024, 2048, 4096, 8192\}$. Implementations include:

- FlashAttention-2 (`flash_attn` 2.8.3)
- cuDNN SDPA (invoked via PyTorch)
- Triton (FlashAttention-v2-style kernel with autotuning)
- CuTile FMHA (online softmax using `ct.mma`)
- Naive unfused attention (three separate GEMMs with softmax)

c) End-to-End LLM Inference.: To assess system-level impact, we benchmark a LLaMA-7B-like model consisting of 4 transformer layers (d_{model}=4096, 32 heads, d_{head}=128, d_{FFN}=11008) in BF16. We measure both prefill (batch×sequence=2048) and single-token decode with context length 2048. Batch sizes are $\{1, 8, 32\}$.

TABLE II
SOFTWARE ENVIRONMENT ACROSS PLATFORMS.

	H100 NVL	**RTX PRO 6000**	**B200**
NVIDIA Driver	580.105	570.195	580.126
CUDA Toolkit	12.6	12.8 + 13.1	12.8 + 13.1
PyTorch	2.7.1	2.8.0	2.8.0
Triton	3.3.1	3.4.0	3.4.0
FlashAttention	2.8.3	2.8.3	2.8.3
CuTile	N/A	1.1.0	1.1.0
OS	Ubuntu 24.04	Ubuntu 24.04	Ubuntu 24.04

We evaluate four execution modes:

- Eager execution with naive attention
- Eager execution with cuDNN SDPA
- Eager execution with FlashAttention-2
- `torch.compile` with SDPA

B. Measurement Protocol

All timings use CUDA event-based measurement (`torch.cuda.Event(enable_timing=True)`), ensuring GPU-side accuracy and excluding host overhead. Each benchmark runs 10 warmup iterations followed by 50 timed iterations. Due to its extreme slowness, Raw SIMT uses 3 warmup and 3 timed iterations.

Reported runtimes are arithmetic means of the timed iterations. We do not report standard deviations, as observed variance was consistently below 1% across all implementations except Raw SIMT.

Throughput is reported in TFLOP/s, computed as:

$$\text{GEMM: } \frac{2MNK}{t \cdot 10^{-3} \cdot 10^{12}}, \quad \text{Attention: } \frac{4BHN^2d}{t \cdot 10^{-3} \cdot 10^{12}}$$

with the attention FLOP count halved for causal masking.

To ensure correctness, all implementations were validated against reference outputs (e.g., PyTorch and cuBLAS) for numerical equivalence. We additionally performed consistency checks across multiple runs and verified stability of results across different input sizes. Kernel implementations were cross-checked against official documentation and established baselines to minimize the likelihood of implementation errors.

C. Software Environment

Table II summarizes the software stack used on each platform. The H100 system runs PyTorch 2.7.1 with CUDA 12.6, while both Blackwell systems use PyTorch 2.8.0 with CUDA 12.8. CuTile 1.1.0 requires the `tileiras` compiler provided by CUDA Toolkit 13.1.

V. GEMM: CAN CUTILE REPLACE CUBLAS OR TRITON?

A. BF16 Square GEMM

Table III presents the primary GEMM results across all three GPUs. The central question for GEMM is whether CuTile's 22-line kernel can approach the performance of cuBLAS (1 line) or Triton (53 lines), which would justify the switch for developers writing custom GEMM variants.

Fig. 2 visualizes these results.

a) Should you switch to CuTile for GEMM?:

1) **Not if you're using cuBLAS.** cuBLAS dominates on all platforms: 1,672 TFLOP/s on B200 (at N=8192), 533 TFLOP/s on H100, and 385–386 TFLOP/s on RTX PRO 6000. CuTile achieves only 52–79% of cuBLAS on Blackwell. If your workload is a standard GEMM (`torch.matmul`), there is *no reason to switch*.
2) **Not if Triton works for your GPU fleet.** Triton achieves 98% of cuBLAS on H100 (N=4096), 62–70% on B200, and is *tied* with cuBLAS at larger sizes on RTX PRO 6000. Unlike CuTile, Triton runs on all three GPUs without code changes.
3) **Yes, if you are currently writing WMMA kernels.** CuTile delivers 1.5–5.0× higher throughput than WMMA on most sizes, in 5.6× less code (22 vs. 123 lines). On the RTX PRO 6000, CuTile reaches 303 TFLOP/s vs. WMMA's 195 TFLOP/s. On the B200, CuTile reaches 962 TFLOP/s vs. WMMA's 200 TFLOP/s. For any developer hand-writing CUDA Tensor Core code, CuTile is a strict upgrade.
4) **Yes, if you need custom GEMM fusions on Blackwell.** CuTile's tile-level API (`ct.mma` + custom epilogues) makes it practical to write fused GEMM variants that are impossible with cuBLAS and awkward in Triton. At 52–79% of cuBLAS, the performance cost of customizability may be acceptable.
5) **Raw SIMT is ∼64–217× slower than cuBLAS**, confirming that Tensor Core utilization is essential for competitive GEMM performance and reinforcing the need for abstractions like CuTile.

B. Rectangular GEMM (LLaMA-7B FFN Projections)

Table IV shows results for rectangular shapes typical of LLaMA-7B's feed-forward network.

The performance ordering is consistent with square GEMM: cuBLAS > Triton > CuTile > WMMA ≫ Raw SIMT. CuTile maintains its position as a strong WMMA replacement in real-world FFN shapes, achieving 261–905 TFLOP/s on Blackwell versus WMMA's 180–204 TFLOP/s. On the H100 (where CuTile is unavailable), cuBLAS reaches 561 TFLOP/s on the down-projection, which is *higher* than square GEMM due to better L2 cache utilization for the asymmetric shape.

C. FP16 vs. BF16 GEMM

A practical concern for developers considering CuTile is whether precision choice affects the performance gap. We benchmark FP16 GEMM to answer this. On the B200, FP16 cuBLAS is within 5% of BF16 (1,597 vs. 1,672 TFLOP/s at N=8192), and CuTile FP16 is generally close to BF16 (within 5% for most sizes, but up to 11% lower at N=8192). This size-dependent sensitivity in the `tileiras` compiler's FP16 code path is a minor concern: CuTile's relative standing versus cuBLAS and Triton does not change with precision. On the H100, FP16 cuBLAS reaches 498 TFLOP/s at N=4096 (vs. 533 TFLOP/s in BF16, 7% lower). On the RTX PRO 6000, an anomaly emerges: Triton FP16 (378 TFLOP/s) *exceeds*

TABLE III
GEMM PERFORMANCE (TFLOP/S) ON BF16 SQUARE MATRICES. CUTILE IS ONLY AVAILABLE ON BLACKWELL GPUS. BEST RESULT PER COLUMN IN **BOLD**.

GPU	Implementation	4096×4096	8192×8192	12288×12288	16384×16384
H100 NVL	cuBLAS	**533.4**	**508.6**	**489.5**	**450.1**
	Triton	525.0	384.8	422.4	392.9
	WMMA	113.7	132.9	121.5	118.6
	Raw SIMT	6.3	6.2	6.2	6.2
RTX PRO 6000	cuBLAS	**378.0**	384.8	386.3	386.4
	Triton	346.0	**389.9**	**386.6**	**386.5**
	CuTile	261.1	302.8	294.5	293.9
	WMMA	179.0	194.9	189.2	185.6
	Raw SIMT	5.9	5.7	5.5	5.6
B200	cuBLAS	**1,518.7**	**1,671.8**	**1,557.4**	**1,517.5**
	Triton	1,066.8	1,032.9	1,002.4	1,011.5
	CuTile	962.2	875.8	852.0	849.9
	WMMA	192.9	199.5	200.0	203.6
	Raw SIMT	8.0	7.7	7.7	7.7

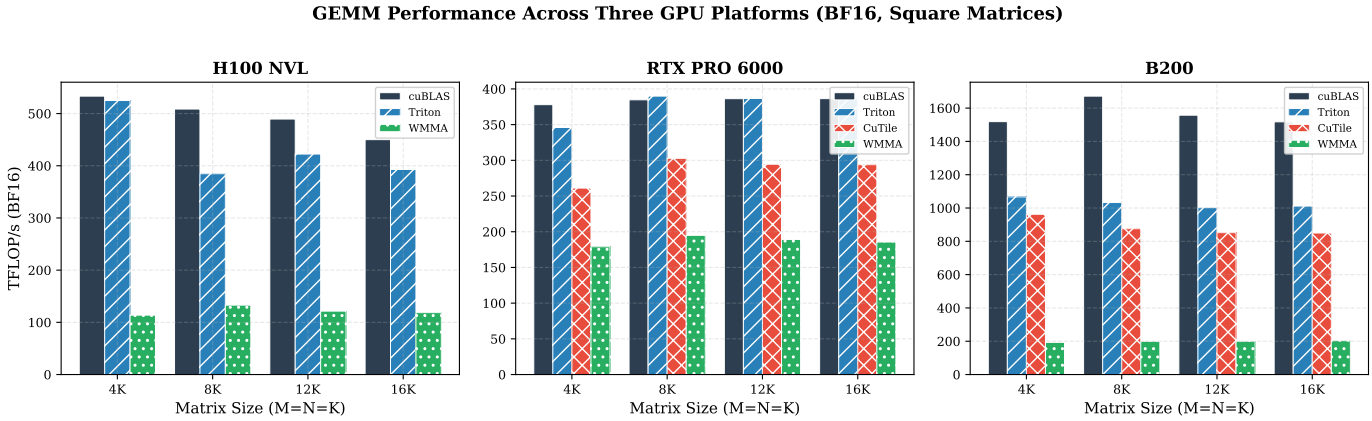

Fig. 2. GEMM performance (TFLOP/s, BF16) across four square matrix sizes on three GPUs. cuBLAS dominates on all platforms. CuTile (available only on Blackwell) achieves 52–79% of cuBLAS, significantly outperforming WMMA. Raw SIMT is omitted (6–8 TFLOP/s) for visual clarity.

TABLE IV
RECTANGULAR GEMM (BF16, TFLOP/S). LLAMA-7B FFN SHAPES: UP = $(M, 4096, 11008)$, DOWN = $(M, 11008, 4096)$.

		seq=2048		seq=4096	
GPU	**Impl.**	**Up**	**Down**	**Up**	**Down**
H100	cuBLAS	502.6	561.2	523.4	547.3
	Triton	455.2	549.1	478.3	496.9
RTX PRO	cuBLAS	377.4	371.1	388.8	377.4
	CuTile	260.9	276.1	296.4	281.8
	Triton	365.6	353.4	386.0	372.1
B200	cuBLAS	1,448	1,562	1,558	1,562
	CuTile	876.6	761.7	905.6	867.8
	Triton	951.4	928.5	974.7	953.3

cuBLAS FP16 (310 TFLOP/s) by 22%, suggesting a cuBLAS tuning gap on `sm_120` unrelated to CuTile.

VI. FUSED ATTENTION: CUTILE'S BEST CASE – AND ITS LIMITS

A. BF16 Causal Attention

Table V presents fused attention results. If the GEMM story was "CuTile is good but not the best," the attention story is far more dramatic – and far more GPU-dependent. This is where CuTile either *dominates* or *disappoints*, depending entirely on which Blackwell chip you have.

Fig. 3 shows how each implementation scales with sequence length across the three GPUs, and Fig. 4 isolates the dramatic CuTile divergence between B200 and RTX PRO 6000.

a) CuTile attention is exceptional on B200 but disappointing elsewhere: The most striking finding of this study – and the most important for adoption decisions – is that the *same* CuTile attention kernel produces dramatically different results on two Blackwell GPUs:

- **B200 (`sm_100`) – Switch to CuTile:** CuTile achieves 1,007 TFLOP/s at seq=4096, which is 2.51× faster than FlashAttention-2 (401 TFLOP/s) and 1.92× faster than Triton (526 TFLOP/s). This is the highest attention

TABLE V

FUSED ATTENTION PERFORMANCE (TFLOP/S) ON BF16 CAUSAL ATTENTION, BATCH=8, 32 HEADS, d=128. BEST PER COLUMN IN **BOLD**. CUTILE AVAILABLE ONLY ON BLACKWELL. – INDICATES OUT-OF-MEMORY.

GPU	Implementation	seq=512	seq=1024	seq=2048	seq=4096	seq=8192
H100 NVL	FlashAttention-2	165.3	**222.9**	**253.4**	**262.0**	**272.5**
	cuDNN (SDPA)	100.8	118.4	121.8	134.2	137.2
	Triton	**168.4**	194.1	222.0	247.2	258.8
	Naive (Unfused)	18.8	19.3	18.2	16.6	–
RTX PRO 6000	FlashAttention-2	**177.3**	**226.8**	**293.2**	**335.4**	**348.6**
	cuDNN (SDPA)	89.3	106.6	125.0	132.9	137.1
	Triton	161.3	202.2	252.5	287.7	303.3
	CuTile	87.5	127.4	157.5	178.8	191.2
	Naive (Unfused)	9.2	10.3	10.7	10.9	–
B200	FlashAttention-2	207.4	297.3	362.0	400.7	422.1
	cuDNN (SDPA)	112.1	137.9	153.8	162.6	167.2
	Triton	234.5	362.5	461.9	525.6	562.0
	CuTile	**405.9**	**669.9**	**887.4**	**1,007.4**	**1,001.3**
	Naive (Unfused)	25.1	27.9	25.8	24.8	–

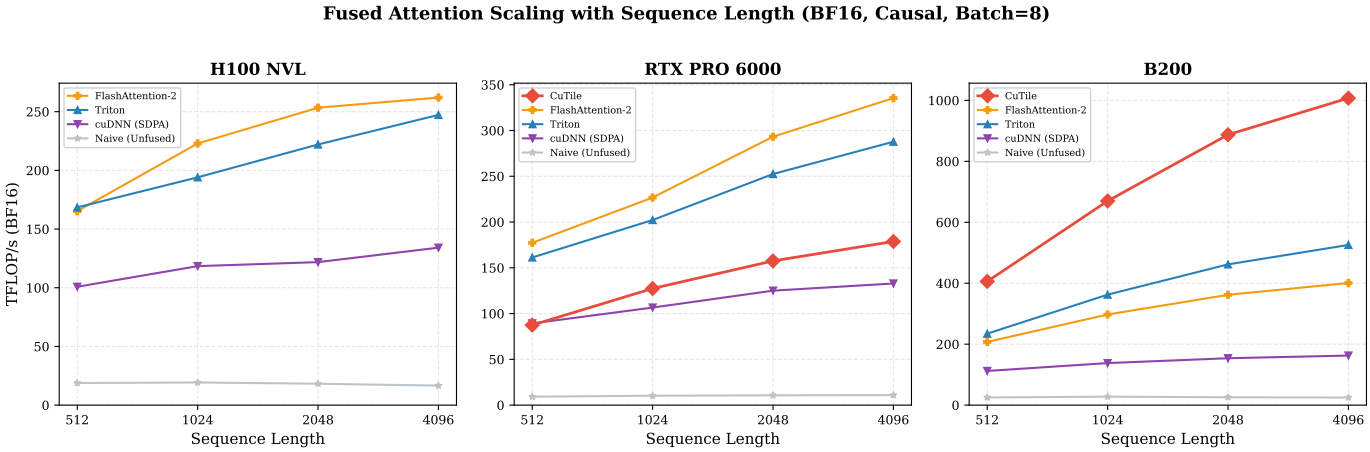

Fig. 3. Fused attention throughput (TFLOP/s) vs. sequence length (BF16, causal, batch=8). On the B200 (right), CuTile (red) dramatically outscales all other implementations, reaching 1,007 TFLOP/s at seq=4096. On the RTX PRO 6000 (center), CuTile underperforms FlashAttention-2 and Triton at all sequence lengths.

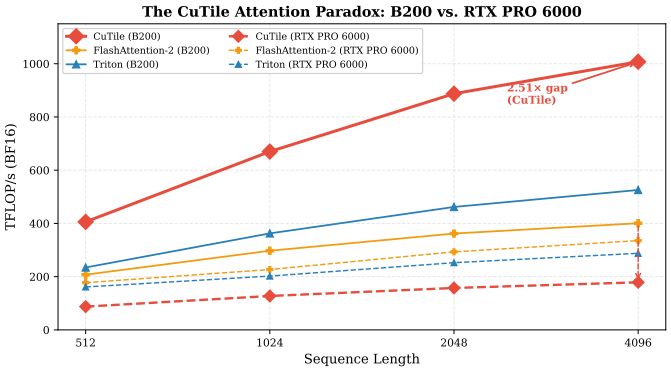

Fig. 4. The CuTile attention paradox: identical kernel code produces 2.51× *higher* throughput than FlashAttention-2 on B200 (solid lines) but 47% *lower* on RTX PRO 6000 (dashed lines). This 5.6× cross-architecture gap is the largest observed for any abstraction.

throughput observed in our entire study. For attention-heavy workloads on B200, *CuTile is the best available implementation.*

- **RTX PRO 6000 (`sm_120`) – Do not switch:** CuTile achieves only 179 TFLOP/s at seq=4096, which is 53% of FlashAttention-2 (335 TFLOP/s) and 62% of Triton (288 TFLOP/s). FlashAttention-2 and Triton remain the better choices.

We attribute this 5.6× performance gap between the two Blackwell variants to differences in the `tileiras` compiler's optimization for `sm_100` versus `sm_120`, and potentially to differences in the TMA and Tensor Core microarchitectures. The B200 (`sm_100`) appears to be the primary optimization target for the CuTile compiler, consistent with NVIDIA's datacenter-first development strategy. This has a critical implication for adoption: developers must benchmark CuTile on their specific GPU before committing to it.

b) Without Blackwell, FlashAttention-2 is the safe choice: FlashAttention-2 provides the most consistent attention performance across all three GPUs, scaling from 165 TFLOP/s (H100, seq=512) to 422 TFLOP/s (B200,

seq=8192). Its hand-optimized CUDA kernels deliver reliable performance regardless of architecture.

c) Triton performs well on B200: At seq=4096 on the B200, Triton reaches 526 TFLOP/s, exceeding FlashAttention-2 (401 TFLOP/s) by 1.31×. For developers who need both attention *and* GEMM performance from a single framework, Triton's consistently strong showing across both workloads makes it a compelling alternative to CuTile.

B. FP16 vs. BF16 Attention

CuTile's relative advantage on B200 persists in FP16: 926 TFLOP/s at seq=4096 (vs. 1,007 in BF16, 8% lower), still 2.31× faster than FlashAttention-2 (400 TFLOP/s in FP16, essentially unchanged from BF16). The CuTile attention recommendation is precision-invariant: switch on B200, don't switch on RTX PRO 6000.

VII. End-to-End LLM Inference

CuTile does not yet provide end-to-end model inference integration – it offers individual kernel primitives (GEMM, attention) but not a complete inference pipeline. We include these end-to-end results to show what performance levels established PyTorch backends achieve, providing context for when CuTile might become relevant for full-model serving. Table VI presents end-to-end inference throughput for a LLaMA-7B–like model [17] across all three GPUs using standard PyTorch backends. Efficient LLM inference requires careful optimization of both the compute-bound prefill and memory-bound decode phases [28]; we focus on the kernel-level performance while noting that advanced techniques such as speculative decoding [31] and continuous batching [5] further improve throughput in production.

Below are the key findings and CuTile implications.

a) Fused attention backends (SDPA, FA2) provide 1.7–2.4× prefill speedup over the naive unfused baseline across all GPUs: Given CuTile's 2.51× attention advantage on B200, a CuTile-integrated inference pipeline could potentially further improve prefill throughput beyond what SDPA/FA2 achieve.

b) torch.compile provides minimal additional benefit for this 4-layer model configuration: At batch=1 on the H100, torch.compile achieves 364.2 TFLOP/s versus 364.0 for Eager (FA2) – a negligible difference. This suggests that kernel-level optimization (where CuTile operates) matters more than graph-level compilation for well-optimized models.

c) Decode performance is memory-bandwidth bound and remarkably uniform across attention backends once fused attention is used: On the B200 at batch=32, all three fused backends achieve 4,594–4,647 tok/s. CuTile would provide no decode advantage here, as decode is bottlenecked by memory bandwidth, not compute.

d) B200 dominates in absolute throughput: 924 TFLOP/s prefill and 4,647 tok/s decode at batch=32, compared to 350 TFLOP/s and 2,887 tok/s on the H100. This 2.6× prefill advantage reflects the B200's higher SM count and memory bandwidth – the same platform where CuTile attention excels.

VIII. The Productivity Payoff: How Much Code Does CuTile Save?

Performance is only half the adoption decision. The other half is developer productivity – how much engineering effort does each abstraction demand? Table VII compares lines of code across all implementations.

a) CuTile's strongest argument: the GEMM kernel is 22 lines: At 22 lines for the GEMM kernel, CuTile requires 5.6× less code than WMMA (123 lines) and 2.4× less than Triton (53 lines). CuTile's `ct.load`, `ct.mma`, and `ct.store` primitives abstract the entire shared memory tiling, register allocation, and warp scheduling pipeline that must be manually managed in WMMA. For developers currently maintaining WMMA kernels, this productivity gain alone may justify the switch.

b) For attention, CuTile offers no productivity advantage over Triton: Both require comparable effort (60 vs. 62 kernel LOC), as both implement the same Flash Attention v2 algorithm with online softmax. The similar code sizes reflect that attention's algorithmic complexity (causal masking, running max/sum for online softmax) dominates over the tiling abstractions. The decision between CuTile and Triton for attention therefore comes down purely to performance (CuTile wins on B200) and portability (Triton wins everywhere else).

c) The adoption tradeoff: CuTile occupies a unique position: it offers near-library-call brevity while exposing enough control for kernel customization. For teams targeting *only Blackwell datacenter GPUs*, CuTile provides the best productivity-to-performance ratio. For teams with mixed GPU fleets (Hopper + Blackwell, or datacenter + workstation), Triton provides better portability despite requiring up to 2.4× more kernel code for GEMM (53 vs. 22 lines; attention code sizes are comparable at 62 vs. 60 lines).

IX. Decision Framework: When to Use CuTile

A. Understanding CuTile's GPU Sensitivity

The 5.6× performance gap between CuTile attention on B200 (1,007 TFLOP/s) and RTX PRO 6000 (179 TFLOP/s) is the most important finding for adoption decisions. Before switching to CuTile, developers must understand *why* this gap exists:

a) Compiler maturity: The `tileiras` compiler (CUDA Toolkit 13.1) is optimized primarily for `sm_100` (datacenter Blackwell). The `sm_120` (workstation Blackwell) appears to receive less compiler optimization.

b) Micro-architectural differences: Despite both being "Blackwell," `sm_100` and `sm_120` have significant differences. The RTX PRO 6000 has only 48 KB shared memory per block (vs. 227–228 KB opt-in on B200/H100), which constrains tile sizes and double-buffering strategies that CuTile relies on.

c) CTA clustering sensitivity: CuTile uses CTA (Cooperative Thread Array) clustering via the `ByTarget` decorator. Our experiments showed that the optimal CTA cluster size differs between `sm_100` and `sm_120`, and incorrect configuration can cause 4× slowdowns.

TABLE VI
END-TO-END LLM INFERENCE (LLAMA-7B, 4 LAYERS, BF16). PREFILL: TFLOP/S (HIGHER IS BETTER). DECODE: TOKENS/SEC (HIGHER IS BETTER).

Batch	Backend	H100 NVL		RTX PRO 6000		B200	
		Prefill (TF/s)	**Decode** (tok/s)	**Prefill** (TF/s)	**Decode** (tok/s)	**Prefill** (TF/s)	**Decode** (tok/s)
1	Eager (Naive)	210.8	589	140.2	555	374.7	767
	Eager (SDPA)	363.6	**667**	282.8	574	826.9	866
	Eager (FA2)	364.0	527	283.3	**575**	**837.3**	865
	torch.compile	**364.2**	630	**283.5**	574	827.5	**866**
8	Eager (Naive)	204.1	1,215	139.7	1,206	389.2	1,582
	Eager (SDPA)	361.1	**2,137**	303.6	2,123	910.0	3,113
	Eager (FA2)	**367.9**	2,133	**303.8**	**2,124**	**918.0**	**3,142**
	torch.compile	357.7	2,123	303.2	2,124	905.9	3,083
32	Eager (Naive)	197.4	1,433	141.5	1,482	381.6	1,924
	Eager (SDPA)	**350.7**	2,851	303.4	3,029	920.5	4,601
	Eager (FA2)	349.7	**2,887**	**303.4**	**3,030**	**924.0**	**4,647**
	torch.compile	346.0	2,840	302.4	3,029	920.4	4,594

TABLE VII
LINES OF CODE COMPARISON. KERNEL LOC = GPU KERNEL ONLY; TOTAL LOC INCLUDES LAUNCHER/WRAPPER CODE.

Operation	Implementation	Abstraction	Kernel	Total
GEMM	cuBLAS (`torch.matmul`)	Library API	1	1
	Triton	Python DSL	53	76
	CuTile	Python DSL	22	45
	WMMA (CUDA C++)	CUDA C++	123	183
	Raw SIMT (CUDA C++)	CUDA C++	32	58
Attention	FlashAttention-2	Library API	1	1
	PyTorch SDPA	Library API	1	4
	Triton	Python DSL	62	87
	CuTile FMHA	Python DSL	60	95
	Naive (Unfused)	PyTorch eager	10	10

TABLE VIII
WHEN TO CHOOSE CUTILE OVER EACH ALTERNATIVE.

Instead of ...	Switch to CuTile when ...
cuBLAS	You need custom fusions that cuBLAS doesn't provide (e.g. fused GEMM+activation)
Triton	You target only B200/B100 *and* need attention kernels (2.5× advantage)
WMMA	Always, on any Blackwell GPU (1.5–5× faster, 5.6× less code)
FA2	Only on B200/B100 for attention; FA2 is better on RTX PRO and all Hopper
Raw SIMT	Always (64–217× faster with Tensor Cores)

The practical implication is that CuTile should be considered mature for `sm_100` (B100, B200) but *experimental* for `sm_120` (RTX PRO series).

B. The Normalized Picture

Fig. 5 presents a normalized performance heatmap across all GPU–implementation combinations, making the adoption decision visual.

For GEMM, the hierarchy is consistent: cuBLAS > Triton > CuTile > WMMA ≫ Raw SIMT on every GPU. This means the GEMM adoption decision is stable regardless of target hardware. For attention, the hierarchy *changes between GPUs* – CuTile is best on B200 but worst-among-fused on RTX PRO 6000 – making the adoption decision hardware-dependent.

C. CuTile vs. Each Alternative: A Direct Comparison

Table VIII summarizes when CuTile is the better choice over each alternative. Fig. 6 visualizes the relative GEMM performance, showing CuTile's position in the ecosystem.

D. CuTile's Scaling Advantage for Long Sequences

One area where CuTile shows uniquely strong behavior is scaling with problem size. On the B200, CuTile attention throughput *increases* from 406 TFLOP/s (seq=512) to 1,007 TFLOP/s (seq=4096), a 2.48× improvement that suggests excellent pipeline utilization at larger tile counts. In contrast, FlashAttention-2 shows more modest scaling (207→401 TFLOP/s, 1.93×). For long-context LLM applications on B200, CuTile's advantage grows with sequence length, making it increasingly attractive as context windows expand.

E. Limitations and Caveats

Developers considering CuTile should be aware of these limitations in our evaluation:

a) Single-GPU evaluation: We benchmark 1 instance per GPU. Manufacturing variation could affect results by 1–3%.

b) No Nsight Compute profiling: We report application-level TFLOP/s but do not decompose into Tensor Core utilization or memory bandwidth. Future work should include roofline analysis [24].

c) CuTile compiler is evolving: The `tileiras` compiler from CUDA Toolkit 13.1 is still maturing. Future versions may significantly improve `sm_120` performance, potentially changing our RTX PRO 6000 recommendation.

d) No FP8/INT8: All experiments use BF16/FP16. CuTile's advantage for quantized inference (increasingly important for production) is unknown.

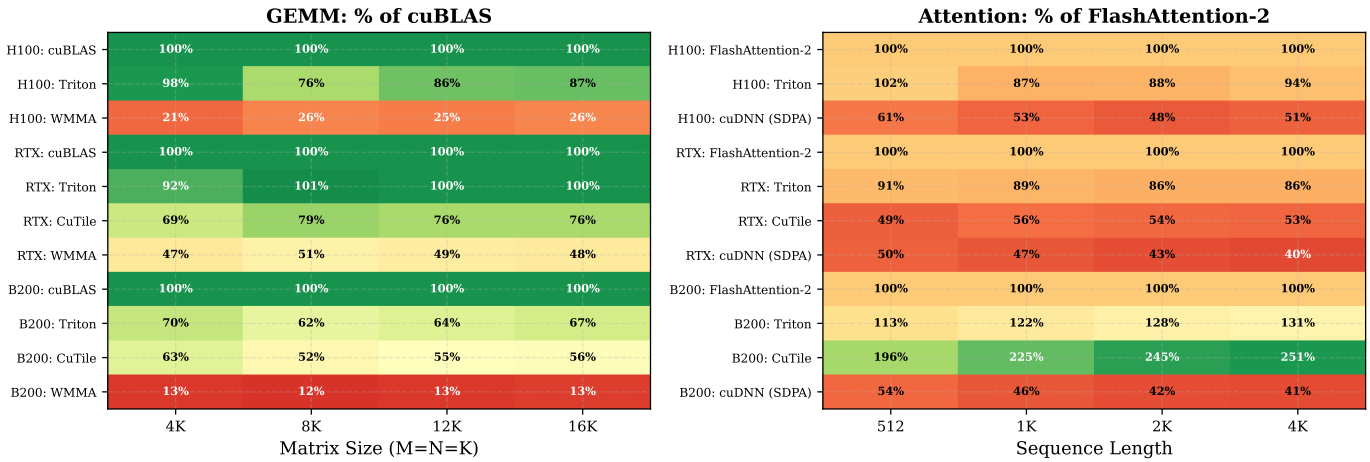

Fig. 5. Normalized performance heatmap. Left: GEMM throughput as percentage of cuBLAS. Right: attention throughput as percentage of FlashAttention-2. CuTile on B200 achieves up to 251% of FA2 (dark green), while CuTile on RTX PRO 6000 reaches only 53% (orange-red). This GPU-dependent behavior is unique to CuTile among all tested abstractions.

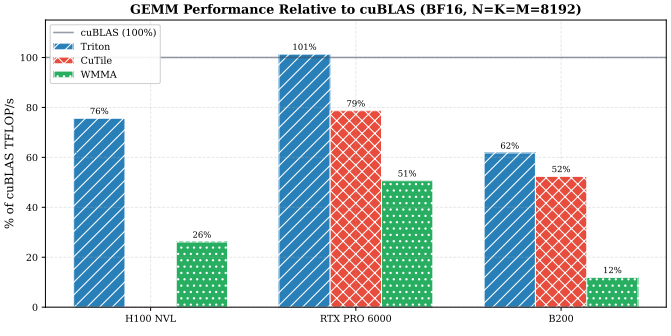

Fig. 6. GEMM performance as a percentage of cuBLAS at N=8192 across three GPUs. Triton (blue) achieves the most consistent cross-architecture performance (62–101%). CuTile (red) shows 52% on B200 and 79% on RTX PRO, but is unavailable on H100.

e) 4-layer proxy model: Our end-to-end inference uses a 4-layer proxy, not a full 32-layer LLaMA-7B. A CuTile-integrated full model may exhibit different bottlenecks.

f) Blackwell-only: CuTile cannot run on Hopper, Ampere, or earlier GPUs. Teams with mixed fleets need a dual-framework strategy.

g) Software version differences: The H100 runs PyTorch 2.7.1 with CUDA 12.6, while both Blackwell GPUs run PyTorch 2.8.0 with CUDA 12.8. These differences could introduce confounds in cross-GPU comparisons for Triton and FlashAttention-2, though we expect the effect to be small (<2%) as both versions use the same core algorithms.

Given the rapid evolution of GPU architectures and the CuTile compiler stack, the results presented in this study reflect the current state of the ecosystem and may change with future CUDA releases and hardware generations.

X. Conclusion: Should You Switch to CuTile?

We have evaluated CuTile head-to-head against cuBLAS, Triton, WMMA, and Raw SIMT across NVIDIA's Hopper (H100) and Blackwell (B200, RTX PRO 6000) architectures on three workloads: GEMM, fused attention, and end-to-end LLM inference. Below is our recommendations.

A. Switch to CuTile If

a) Writing fused attention kernels for B200/B100 datacenter GPUs: CuTile delivers 1,007 TFLOP/s – 2.5× faster than FlashAttention-2 – in 60 lines of Python. No other abstraction comes close.

b) Currently maintaining WMMA kernels and your deployment targets Blackwell: CuTile provides 1.5–5.0× better throughput in 5.6× less code. It is a strict upgrade.

c) Need custom GEMM fusions (e.g. fused GEMM + activation, custom epilogues) on Blackwell and cannot use cuBLAS's fixed API: CuTile's 22-line kernel is far more accessible than WMMA's 123 lines.

B. Not to Switch If

a) Need cross-architecture portability (Hopper + Blackwell, or mixed GPU fleets): CuTile does not run on Hopper or earlier architectures. Triton is a good option (62–101% of cuBLAS on all GPUs, no architecture-specific code changes).

b) Using standard GEMM (`torch.matmul` / cuBLAS) and do not need kernel customization: CuTile achieves only 52–79% of cuBLAS. There is no reason to switch.

c) Targeting RTX PRO 6000 or other workstation Blackwell GPUs (`sm_120`): CuTile attention achieves only 53% of FlashAttention-2 on this platform due to compiler immaturity.

d) Need FP8 or INT8 kernels: CuTile's quantized performance is untested.

C. Wait and Re-evaluate When

a) CUDA Toolkit 14.x or later is released: The `tileiras` compiler is still maturing. Our RTX PRO 6000 results may improve dramatically with future compiler versions, potentially resolving the 5.6× cross-GPU gap.

b) CuTile gains Hopper support: If NVIDIA backports CuTile to `sm_90`, the portability objection disappears and CuTile becomes directly competitive with Triton.

c) CuTile is integrated into PyTorch or vLLM: End-to-end inference frameworks that use CuTile attention on B200 could unlock the 2.5× attention speedup at the system level.

D. The broader lesson

CuTile is not yet a replacement for established abstractions, but it is the first credible vendor-backed attempt to make Tensor Core programming *accessible* to Python developers. The fact that a 60-line CuTile kernel outperforms FlashAttention-2's thousands of lines of hand-optimized CUDA on B200 is a paradigm shift. As the compiler matures and GPU support broadens, tile-centric programming models like CuTile may fundamentally change how GPU kernels are developed.

The artifacts are located at: https://github.com/uwm-se/CuTile.

References

[1] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[2] Tri Dao. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2307.08691.

[3] Tri Dao, Daniel Haziza, Francisco Massa, and Grigory Sizov. Flash-Decoding for Long-Context Inference. Blog post, 2023. https://crfm.stanford.edu/2023/10/12/flashdecoding.html.

[4] NVIDIA. CUTLASS: CUDA Templates for Linear Algebra Subroutines. https://github.com/NVIDIA/cutlass, 2024.

[5] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

[6] NVIDIA. TensorRT-LLM: An Open-Source Library for Optimizing LLM Inference. https://github.com/NVIDIA/TensorRT-LLM, 2024.

[7] NVIDIA. CUDA Tile Python Programming Guide. https://docs.nvidia.com/cuda/cutile-python/, 2025.

[8] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 10–19, 2019.

[9] NVIDIA. CUDA C++ Programming Guide: Warp Matrix Functions. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#wmma, 2024.

[10] NVIDIA. cuBLAS Library. https://developer.nvidia.com/cublas, 2024.

[11] NVIDIA. NVIDIA Hopper GPU Architecture Tuning Guide. https://docs.nvidia.com/cuda/hopper-tuning-guide/, 2024.

[12] NVIDIA. NVIDIA Ampere GPU Architecture Tuning Guide. https://docs.nvidia.com/cuda/ampere-tuning-guide/, 2024.

[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI*, 2018.

[14] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. TensorIR: An Abstraction for Automatic Tensorized Program Optimization. In *ASPLOS*, 2023.

[15] 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 *OSDI*, 2022.

[16] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, Peng Di, Kun Zhang, and Xuefeng Jin. AKG: Automatic Kernel Generation for Neural Processing Units using Polyhedral Transformations. In *PLDI*, 2021.

[17] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971, 2023.

[18] NVIDIA. NVIDIA H100 Tensor Core GPU Architecture. NVIDIA Whitepaper, 2022.

[19] NVIDIA. NVIDIA Blackwell Architecture Technical Brief. NVIDIA, 2024.

[20] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. In *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 5998–6008, 2017.

[21] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[22] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. PaLM: Scaling Language Modeling with Pathways. *Journal of Machine Learning Research (JMLR)*, 24(240):1–113, 2023.

[23] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053, 2020.

[24] 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.

[25] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. arXiv:1410.0759, 2014.

[26] 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 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 519–530, 2013.

[27] Vasily Volkov and James W. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In *Proceedings of the IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2008.

[28] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently Scaling Transformer Inference. In *Proceedings of Machine Learning and Systems (MLSys)*, 2023.

[29] Noam Shazeer. Fast Transformer Decoding: One Write-Head is All You Need. arXiv:1911.02150, 2019.

[30] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yinfei Yang, Sumit Sanghai, and Santiago Ontanón. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023.

[31] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast Inference from Transformers via Speculative Decoding. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2023.

[32] Maxim Milakov and Natalia Gimelshein. Online Normalizer Calculation for Softmax. arXiv:1805.02867, 2018.