CODA: Rewriting Transformer Blocks as GEMM-Epilogue Programs

Han Guo[1*] **Jack Zhang**[2] **Arjun Menon**[2]
Driss Guessous[4] **Vijay Thakkar**[4] **Yoon Kim**[1] **Tri Dao**[2,3]
[1]Massachusetts Institute of Technology [2]Princeton University [3]Together AI [4]Meta
hanguo@mit.edu

Abstract

Transformer training systems are built around dense linear algebra, yet a nontrivial fraction of end-to-end time is spent on surrounding memory-bound operators. Normalization, activations, residual updates, reductions, and related computations repeatedly move large intermediate tensors through global memory while performing little arithmetic, making data movement an increasingly important bottleneck in otherwise highly optimized training stacks. We introduce CODA, a GPU kernel abstraction that expresses these computations as GEMM-plus-epilogue programs. CODA is based on the observation that many Transformer operators exposed as separate framework kernels can be algebraically reparameterized to execute while a GEMM output tile remains on chip, before it is written to memory. The abstraction fixes the GEMM mainloop and exposes a small set of composable epilogue primitives for scaling, reductions, pairwise transformations, and accumulation. This constrained interface preserves the performance structure of expert-written GEMMs while remaining expressive enough to cover nearly all non-attention computation in the forward and backward pass of a standard Transformer block. Across representative Transformer workloads, both human- and LLM-authored CODA kernels achieve high performance, suggesting that GEMM-plus-epilogue programming offers a practical path toward combining framework-level productivity with hardware-level efficiency.[2]

1 Introduction

LLM training has become just as much of a systems problem as a modeling one. FLOPs in modern Transformer-based LLMs are dominated by matrix multiplications and attention, whose kernels have been heavily optimized for Tensor Core execution. Yet Transformers, and deep learning architectures more broadly, also contain normalization, activations, residual updates, reductions, and other bandwidth-limited operations that move large tensors through memory while doing little arithmetic. Prior work has shown that data movement is a central bottleneck in Transformer training [7]; as Figure 1 shows, when training a LLaMA-3-style 1B model on a single H100 using TorchTitan [11], these non-GEMM operations account for a nontrivial fraction

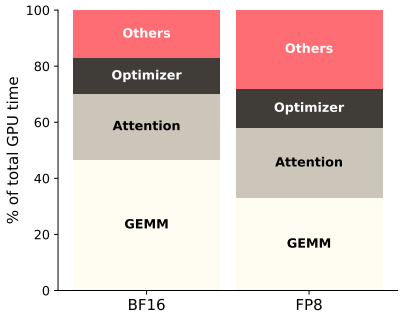

Figure 1: Runtime breakdown for LLaMA-3-style 1B model training on a single H100 using TorchTitan.

*Part of this work was done at Together AI.
[2]Code available at https://github.com/HanGuo97/coda-kernels.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

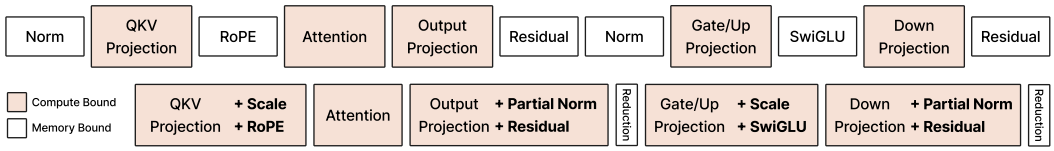

Figure 2: Forward pass of a standard Transformer layer. The top row shows the canonical formulation, which maps to a mix of compute- and memory-bound kernels. We reparameterize the computation so that most memory-bound operations are subsumed into the epilogues of compute-bound kernels.

of end-to-end runtime. As hardware increasingly accelerates low-precision matrix multiplication through formats such as FP8 and FP4, this bottleneck is becoming more important, as the cost of materializing intermediate tensors does not improve at the same rate.

Existing programming models make this issue difficult to address. High-level frameworks such as PyTorch express Transformer blocks as operator sequences, with autograd making the backward pass similarly convenient. This is productive, but operator boundaries often become materialization boundaries and obscure fusion opportunities across forward and backward computation. Production-level LLM systems therefore often bypass framework abstractions with hand-written backward passes or custom kernels, as in large-scale LLaMA training [5] and inference systems [9, 25]. This work asks whether there is a middle ground. That is, is it possible to recover much of the performance of custom kernels without giving up the structure needed for programmability and automation?

Our starting observation is that many Transformer computations that appear at the framework level as separate operators can be algebraically reparameterized as GEMM-plus-epilogue programs (Figure 2). In this form, a highly optimized GEMM mainloop produces output tiles, while a programmable epilogue performs tile-local transformations before the result is written to memory (Figure 3). This is efficient on GPUs because the epilogue operates on data that is already produced by the GEMM tile, avoiding additional global-memory round trips for intermediate tensors. With modern pipelined schedules, this epilogue work can often be placed in the shadow of other tiles' mainloops, as in Hopper Ping-Pong GEMM and Blackwell TMEM-based pipelines. Thus, we extend the epilogue beyond a place for simple post-processing such as scaling or bias addition, elevating it into a structured interface for fusing memory-bound computation into the lifetime of a GEMM tile.

Based on the above, we introduce `CODA`, a kernel abstraction prototype that realizes this interface. `CODA` keeps the GEMM mainloop fixed and exposes a small set of composable epilogue primitives for scaling, reductions, pairwise transformations, and accumulation. This programming model is deliberately constrained and yet expressive, as after reparameterization, these primitives cover nearly the entire forward and backward pass of a standard Transformer model while preserving efficiency. `CODA` inserts computation into the epilogue of a known high-performance GEMM before intermediate tensors are materialized in global memory, capturing a broad class of memory-bound computations surrounding dense linear algebra. Transformers are our primary application, but the same GEMM-plus-epilogue view applies more broadly whenever high-throughput matrix multiplication is surrounded by tile-expressible, data-movement-bound computations.

Finally, this structure makes automation more practical. Epilogue fusion is already established in high-performance GEMM libraries, but applying it to Transformer workloads remains a low-level engineering task. `CODA` targets this gap by providing Transformer-specific epilogue primitives on top of a tuned GEMM mainloop. Human or LLM-based authors can assemble these primitives into reparameterized Transformer kernels rather than synthesizing arbitrary CUDA. Across representative workloads, both authoring modes achieve high performance, suggesting that domain-specific epilogue abstractions can make established GEMM fusion techniques more programmable for LLM kernels.

2 Background and Related Works

2.1 Programming Models for LLM Systems

Modern LLM systems are programmed at multiple abstraction levels. Frameworks such as PyTorch and JAX express models as tensor-operator graphs and integrate naturally with automatic differentiation, but operator boundaries often become materialization ones.

Compiler systems lower tensor programs to optimized kernels through graph rewriting, scheduling, code generation, and autotuning [1, 2, 19]. Algebraic reformulation is another important source of performance, as shown by TASO [8] and Mirage [22]. However, rapidly evolving accelerators make peak performance a moving target for general-purpose compilers.

Closer to the hardware level, programmers use kernel DSLs and libraries such as Triton [19], ThunderKittens [14, 13, 17], TileLang [20], CuTeDSL [18], Gluon, and TLX, or rely on specialized LLM kernels in vLLM [9], SGLang [25], FlashInfer [23], and Liger Kernels [6]. These approaches deliver high performance, but extending them to new transformations or backward computations still requires substantial low-level engineering.

2.2 GEMM Mainloops and Epilogue Fusion

Matrix multiplication is the central compute primitive in modern LLM workloads. A high-performance GEMM kernel is typically divided into a mainloop and an epilogue. The mainloop performs the tiled matrix multiply-accumulate computation, while the epilogue transforms the computed output tile and efficiently writes it back to global memory.

The epilogue is a natural place to implement fusions because the output of the matmul is already present on chip close to compute cores. Practical epilogues commonly perform scaling, bias addition, activations, residual updates, data type conversions, tile-wise reductions and other output elementwise operations, avoiding separate kernel launches and extra global-memory round trips. Modern kernel libraries formalize this separation directly: CUTLASS [18] represents GEMM kernels as a composition of a collective mainloop and a collective epilogue, while Epilogue Visitor Trees further express epilogues as compositions of primitives [4].

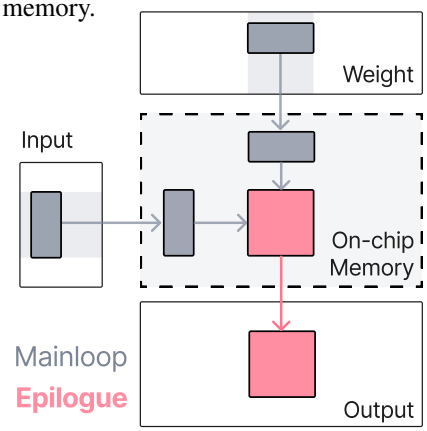

Figure 3: A GEMM mainloop computes output tiles; the epilogue transforms each tile before the final global-memory store.

This flexibility operates under a locality constraint. An epilogue sees only the local output tile, its accumulators, and consistently indexed auxiliary tensors, meaning that operations requiring global reductions or cross-tile communication must be reformulated into tile-local pieces or handled in a separate pass. CODA builds on this interface, keeping the high-performance GEMM mainloop fixed and using the epilogue as a programmable site for nearby memory-bound computation.

3 CODA

The previous section argued that GEMM epilogues are a natural place to fuse memory-bound computation into dense linear algebra. We now describe `CODA`, a GPU kernel abstraction that realizes this idea. Section 3.1 identifies a small set of epilogue primitives that map efficiently to GPU execution. Section 3.2 shows how the non-attention and non-embedding portions of the Transformer forward and backward pass can be reparameterized using these primitives. Finally, Section 3.3 describes their implementation and our LLM-oriented authoring workflow.

3.1 Efficient Epilogue Primitives

`CODA` programs the GEMM epilogue while keeping the mainloop fixed and highly optimized. For each output tile, an epilogue may load auxiliary data, transform accumulator values, emit auxiliary results, and store the final output. This interface is deliberately restricted to tile-local computation rather than arbitrary global communication. Our epilogue template, shown in Section B.1, is inspired by Epilogue Visitor Trees [4]. `CODA` provides five classes of epilogue primitives:

1. *Elementwise and pairwise maps:* apply local transformations to accumulator values, including residual updates, activations, RoPE-style rotations, and SwiGLU-style gates.

2. *Vector (Rank-1 Tensor) loads and stores:* load row or column vectors, broadcast them over an output tile, and optionally write vector-valued auxiliary results.
3. *Tile (Rank-2 Tensor) loads and stores:* load or store matrix tiles, such as residual streams, saved activations, or intermediate values needed by the backward pass.
4. *Tile (Rank-2 Tensor) reductions:* compute partial reductions over rows or columns of an output tile, to be combined later by a lightweight auxiliary kernel.
5. *Stateful transforms:* maintain running tile state, such as the max and sum-exp statistics used in online log-sum-exp and cross-entropy.

These primitives are intentionally narrow, operating at a level low enough to compile to efficient epilogue code and expressive enough to capture the memory-bound operations surrounding GEMMs in our Transformer reparameterizations, as shown next.

3.2 Reparameterizing Transformers as Epilogues

We now show that the primitive set above is sufficient for much of Transformer computation. After lightweight algebraic reparameterization, many non-attention and non-embedding components of a standard Transformer forward pass can be written as

$$\text{GEMM:} \quad \boldsymbol{h} = \boldsymbol{x}\boldsymbol{W}, \qquad \text{Epilogue:} \quad \boldsymbol{y}[i,j] = \boldsymbol{f}[i,j](\boldsymbol{h}[i,j]),$$

where $[i, j]$ indexes an output tile and $\boldsymbol{f}[i, j]$ is the tile function implemented in the GEMM epilogue. The epilogue is either fully tile-local, or tile-local up to partial results that are combined by a lightweight auxiliary reduction. We first apply this view to the forward pass, then show that independent tile functions preserve the same GEMM-epilogue structure in the backward pass.

3.2.1 GEMM-Residual-RMSNorm-GEMM Pattern

A recurring pattern in pre-normalized Transformers is a GEMM followed by a residual update and normalization, then another GEMM. This pattern appears across several adjacent sublayers:

1. attention output projection → residual stream → RMSNorm → MLP gate/up projection;
2. MLP down projection → residual stream → RMSNorm → attention QKV projection;
3. final MLP down projection → residual stream → final RMSNorm → language modeling head.

Although these cases are usually written as parts of different modules, they share the same computational structure:

$$\boldsymbol{y} = \mathrm{RMSNorm}(\boldsymbol{x}\boldsymbol{W}_0 + \boldsymbol{z}, \boldsymbol{\gamma})\boldsymbol{W}_1 = \Big(r\left(\boldsymbol{x}\boldsymbol{W}_0 + \boldsymbol{z}\right) \odot \boldsymbol{\gamma}\Big)\boldsymbol{W}_1,$$

where $\boldsymbol{z}$ denotes the residual stream and $r = 1/\,\mathrm{rms}(\boldsymbol{x}\boldsymbol{W}_0 + \boldsymbol{z})$ is the row-wise inverse RMS factor. This pattern crosses the usual module boundary: it couples the output projection of one sublayer with the input projection of the next.

Residual addition and multiplication by the RMSNorm weight $\boldsymbol{\gamma}$ are tile-local, so they can be fused into a GEMM epilogue. The row-wise factor r, however, requires a reduction across the hidden dimension, which is larger than a single output tile. In the canonical computation, r is applied before the second GEMM, creating an apparent dependency between normalization and the next projection.

We address the reduction by splitting it into two levels. The first GEMM epilogue computes tile-local partial reductions, and a small auxiliary kernel reduces these partials across tiles to obtain r. Since the auxiliary kernel reads a few partial values per tile rather than the full activation tensor, its memory traffic is much smaller than that of a standalone RMSNorm.

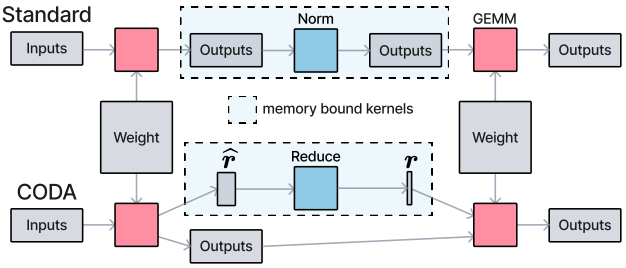

Figure 4: GEMM-RMSNorm-GEMM reparameterization.

The apparent dependency on r can be removed algebraically. Since r is shared across the row, it commutes with the second GEMM:

$$\boldsymbol{y} = \Big(r\left(\boldsymbol{x}\boldsymbol{W}_0 + \boldsymbol{z}\right) \odot \boldsymbol{\gamma}\Big)\boldsymbol{W}_1 = r\,\Big(\left(\boldsymbol{x}\boldsymbol{W}_0 + \boldsymbol{z}\right) \odot \boldsymbol{\gamma}\Big)\boldsymbol{W}_1.$$

Thus, the row-wise scale does not need to be applied before the second GEMM. It can instead be delayed to the epilogue of the second GEMM, after the projection has been computed.

Concretely, the computation decomposes into two GEMMs and one lightweight reduction (Figure 4):

$$\text{GEMM 1:} \quad \boldsymbol{h}_0 = \boldsymbol{x}\boldsymbol{W}_0, \qquad \text{Epilogue 1:} \quad \begin{aligned} \boldsymbol{h}_1[i,j] &= \boldsymbol{h}_0[i,j] + \boldsymbol{z}[i,j], \\ \boldsymbol{h}_2[i,j] &= \boldsymbol{h}_1[i,j] \odot \boldsymbol{\gamma}[j], \\ \widehat{\boldsymbol{r}}[i,j] &= \text{partialRMS}(\boldsymbol{h}_1[i,j]), \end{aligned}$$

$$\text{GEMM 2:} \quad \boldsymbol{h}_3 = \boldsymbol{h}_2\boldsymbol{W}_1, \qquad \text{Epilogue 2:} \quad \boldsymbol{y}[i,j] = r[i]\,\boldsymbol{h}_3[i,j].$$

Here $r = 1/\sqrt{\text{reduce}(\widehat{\boldsymbol{r}}) + \epsilon}$ is computed by a small auxiliary reduction over the tile partials. This decomposition replaces a standalone RMSNorm kernel with tile-local epilogue work around the two GEMMs, plus a lightweight auxiliary reduction.

In Figure 5, we benchmark this reparameterization against existing implementations on LLaMA-style configurations with a batch of 16K tokens. We vary the hidden dimension across representative model scales, with $d \in \{2048, 4096, 8192\}$ corresponding roughly to 1B, 7B, and 70B models, respectively. Our GEMM-Epilogue kernel is generated by an LLM provided with the above abstractions (explained in more detail in Section 3.3.1).

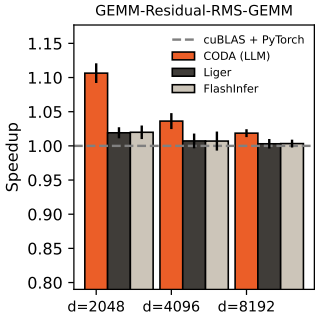

Figure 5: Benchmarks.

Numerics. The reparameterization changes where the RMSNorm scale is applied: the row-wise factor r is delayed from before the second GEMM to the second GEMM epilogue. We compare `BF16` GEMM-RMSNorm-GEMM outputs against an `FP32` reference on Llama-3 8B layers. We report the errors of `CODA` and QuACK, on which our GEMM template is based, normalized by the error of the standard PyTorch path. Figure 6 suggests that a more accurate GEMM mainloop can reduce numerical error, and that `CODA`'s reparameterized epilogue can reduce it further.

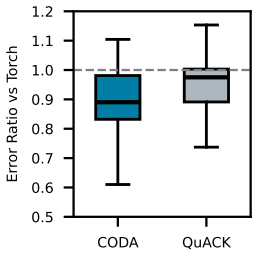

Figure 6: Relative error.

3.2.2 GEMM with Pairwise Activations

A second common pattern in Transformers is a GEMM followed by a *pairwise* activation. Unlike an elementwise activation, which transforms each feature independently, a pairwise activation consumes two adjacent feature values and produces one or two outputs.

$$\boldsymbol{h} = \boldsymbol{x}\boldsymbol{W}, \qquad \boldsymbol{h}_a[i,j], \boldsymbol{h}_b[i,j] = \text{split}\left(\boldsymbol{h}[i,j]\right), \qquad \boldsymbol{y}[i,j] = \boldsymbol{f}[i,j]\left(\boldsymbol{h}_a[i,j], \boldsymbol{h}_b[i,j]\right).$$

This form captures several operations in Transformer blocks:

- RoPE rotates each feature pair and return two outputs;
- SwiGLU combines gate and value stream into one output;
- SwiGLU backward pass maps one incoming gradient into gradients for both paired inputs.

Pairwise activations couple neighboring feature lanes and may change the feature dimension. A naive implementation materializes the GEMM output, splits it into paires, and applies the activation in a separate kernel. This adds memory traffic and sometimes materializes an expanded intermediate, as in SwiGLU.

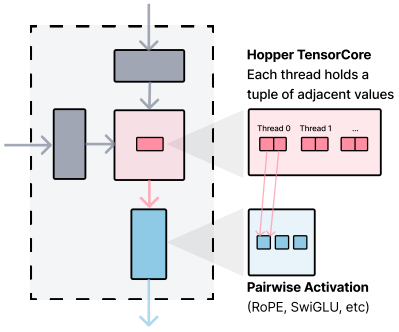

Figure 7: Pairwise activations operate on local feature pairs in the GEMM epilogue.

Instead, we arrange paired features to be adjacent along the output-feature dimension. This matches the Hopper Tensor Core accumulator layout exposed to the epilogue, where each thread holds a small tuple of adjacent output values in registers before they are stored. The epilogue can therefore apply f directly to each pair with register-level computation.

This removes the standalone activation kernel and avoids materializing the paired intermediate in global memory. The same idea applies to dimension-preserving operations such as RoPE, dimension-reducing operations such as SwiGLU, and dimension-expanding operations in the backward pass, as

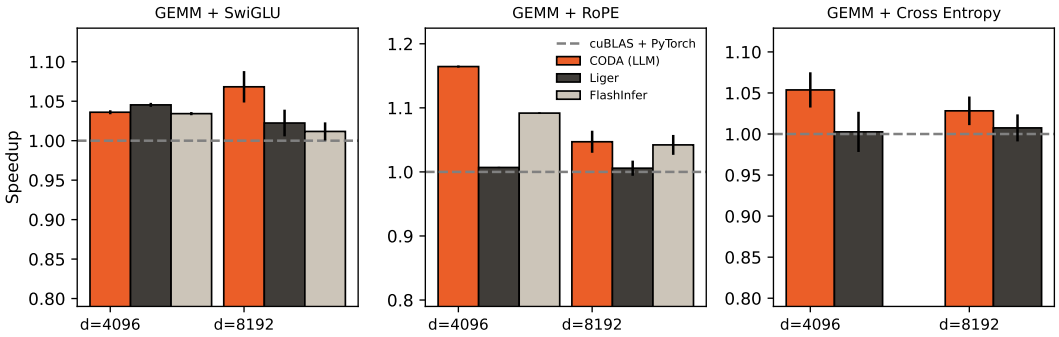

Figure 8: Kernel-level speedups for representative GEMM-plus-epilogue primitives across MNK sizes. RoPE uses an output dimension of $3N$ for QKV projections, and cross-entropy uses a 32K vocabulary. Speedups are relative to cuBLAS with `torch.compile`.

long as the pairing is reflected in the GEMM output layout. See Figure 8 for performance benchmarks.

3.2.3 GEMM with Cross-Entropy Loss

Cross-entropy loss can also be expressed as a GEMM with epilogue-side reductions, as shown by Cut Cross-Entropy [21]. Let $\boldsymbol{h}_i = \boldsymbol{x}_i \boldsymbol{W}_{\text{lm}}$ be the logits for token i, and let $\boldsymbol{y}_i$ be its target label. The per-token loss is

$$\ell_i = -\boldsymbol{h}_{i,\boldsymbol{y}_i} + \log \sum_k \exp(\boldsymbol{h}_{i,k}).$$

Thus, the loss decomposes into an indexed logit and a row-wise log-sum-exp over vocabulary entries.

Both terms fit the GEMM-plus-epilogue pattern. The indexed logit can be selected from the GEMM output tile using the target label, while the LSE can be accumulated as tile-local maximum and sum-exp statistics. A small auxiliary reduction then combines these statistics across tiles, avoiding a standalone memory-bound softmax over the full logits.[3] See Figure 8 for performance benchmarks.

3.2.4 Backward Pass

The preceding sections show that much of the Transformer forward pass can be reparameterized as GEMMs with epilogues, plus lightweight auxiliary reductions. We now show that the backward pass preserves the same structure.

GEMM with elementwise epilogue. Consider two GEMMs separated by an elementwise epilogue:

$$\boldsymbol{h} = \boldsymbol{x}\boldsymbol{W}_0, \qquad \boldsymbol{h}' = f(\boldsymbol{h}), \qquad \boldsymbol{y} = \boldsymbol{h}'\boldsymbol{W}_1,$$

where f is applied elementwise. Given an upstream gradient $\nabla_{\boldsymbol{y}}\mathcal{L}$, reverse-mode differentiation gives

$$\nabla_{\boldsymbol{h}'}\mathcal{L} = \nabla_{\boldsymbol{y}}\mathcal{L}\boldsymbol{W}_1^\top, \qquad \nabla_{\boldsymbol{h}}\mathcal{L} = \nabla_{\boldsymbol{h}'}\mathcal{L} \odot f'(\boldsymbol{h}), \qquad \nabla_{\boldsymbol{x}}\mathcal{L} = \nabla_{\boldsymbol{h}}\mathcal{L}\boldsymbol{W}_0^\top.$$

Thus, the backward computation has the same structure as the forward computation: GEMM, local transformation, GEMM. The only difference is the direction of fusion. In the forward pass, f is fused into the epilogue of the *preceding* GEMM that produces $\boldsymbol{h}$; in the backward pass, multiplication by $f'(\boldsymbol{h})$ is fused into the epilogue of the *following* GEMM that produces $\nabla_{\boldsymbol{h}'}\mathcal{L}$ (Figure 9).

Theorem 1. *Consider a sequence of GEMM-with-epilogue blocks followed by a final GEMM:*

$$\boldsymbol{h}_\ell = \boldsymbol{x}_{\ell-1}\boldsymbol{W}_\ell, \qquad \boldsymbol{x}_\ell[i,j] = \boldsymbol{f}_\ell[i,j](\boldsymbol{h}_\ell[i,j]), \qquad \ell = 1,\dots,L-1,$$
$$\boldsymbol{h}_L = \boldsymbol{x}_{L-1}\boldsymbol{W}_L.$$

Assume each tile function $\boldsymbol{f}_\ell[i,j]$ acts only on its corresponding GEMM output tile. Then the activation gradients can be computed with the same GEMM-with-epilogue structure:

$$\nabla_{\boldsymbol{x}_{\ell-1}}\mathcal{L} = \nabla_{\boldsymbol{h}_\ell}\mathcal{L}\boldsymbol{W}_\ell^\top, \qquad \nabla_{\boldsymbol{h}_{\ell-1}}\mathcal{L}[i,j] = \boldsymbol{g}_{\ell-1}[i,j]\big(\nabla_{\boldsymbol{x}_{\ell-1}}\mathcal{L}[i,j],\ \boldsymbol{h}_{\ell-1}[i,j]\big), \qquad \ell = L,\dots,2,$$
$$\nabla_{\boldsymbol{x}_0}\mathcal{L} = \nabla_{\boldsymbol{h}_1}\mathcal{L}\boldsymbol{W}_1^\top.$$

[3] We use a separate final reduction rather than atomics, and materialize logits to simplify the backward pass.

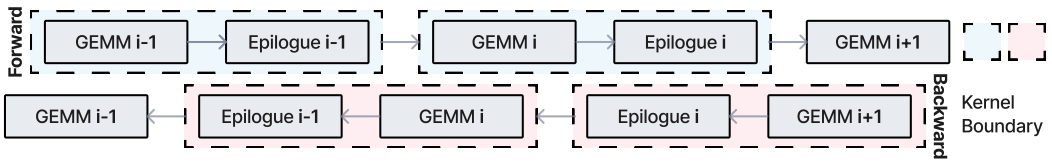

Figure 9: Forward and backward fusion for GEMM–epilogue blocks. Forward epilogues attach to the GEMM that produces their input, while backward epilogues attach to the GEMM that produces the gradient with respect to their output.

Here $\boldsymbol{g}_\ell[i,j]$ is the tile-local backward rule for $\boldsymbol{f}_\ell[i,j]$: it maps the gradient of the epilogue output tile $\boldsymbol{x}_\ell[i,j]$ to the gradient of the epilogue input tile $\boldsymbol{h}_\ell[i,j]$. The weight gradients are GEMMs:

$$\nabla_{\boldsymbol{W}_\ell}\mathcal{L} = \boldsymbol{x}_{\ell-1}^\top \nabla_{\boldsymbol{h}_\ell}\mathcal{L}, \qquad \ell = 1,\dots,L.$$

Thus, tile-local epilogues in the forward pass induce tile-local epilogues in the backward pass, while the surrounding linear maps remain GEMMs.

Proof sketch. For the GEMM $\boldsymbol{h}_\ell = \boldsymbol{x}_{\ell-1}\boldsymbol{W}_\ell$, reverse-mode differentiation gives

$$\nabla_{\boldsymbol{x}_{\ell-1}}\mathcal{L} = \nabla_{\boldsymbol{h}_\ell}\mathcal{L}\boldsymbol{W}_\ell^\top, \qquad \nabla_{\boldsymbol{W}_\ell}\mathcal{L} = \boldsymbol{x}_{\ell-1}^\top \nabla_{\boldsymbol{h}_\ell}\mathcal{L},$$

which are both GEMMs. For the epilogue $\boldsymbol{x}_\ell[i,j] = \boldsymbol{f}_\ell[i,j](\boldsymbol{h}_\ell[i,j])$, define the local backward rule by

$$\nabla_{\boldsymbol{h}_\ell}\mathcal{L}[i,j] = \boldsymbol{g}_\ell[i,j](\nabla_{\boldsymbol{x}_\ell}\mathcal{L}[i,j],\ \boldsymbol{h}_\ell[i,j])\,.$$

Since each $\boldsymbol{f}_\ell[i,j]$ depends only on its own tile, the corresponding $\boldsymbol{g}_\ell[i,j]$ also acts only on that tile. The backward pass therefore introduces no new cross-tile communication and preserves the GEMM-with-epilogue structure. □

RMSNorm backward. RMSNorm is the main case where the backward pass is not purely tile-local. Its backward rule introduces two reductions: a row-wise statistic needed for the input gradient, and a feature-wise reduction across rows for the RMSNorm weight gradient. A direct implementation computes both in a standalone RMSNorm backward kernel, requiring additional reads of activation-sized tensors. However, the row-wise statistic can be moved to a neighboring GEMM boundary. Consider

$$\boldsymbol{h}_0 = \boldsymbol{x}\boldsymbol{W}_0, \qquad \boldsymbol{h}_1 = f(\boldsymbol{h}_0), \qquad \boldsymbol{h}_2 = \mathrm{RMSNorm}(\boldsymbol{h}_1,\boldsymbol{\gamma}), \qquad \boldsymbol{y} = \boldsymbol{h}_2\boldsymbol{W}_1.$$

RMSNorm backward requires the row-wise inner product

$$\boldsymbol{s} = \frac{1}{d}\,\mathrm{sum}_{\mathrm{cols}}\left(\nabla_{\boldsymbol{h}_2}\mathcal{L} \odot \boldsymbol{h}_2\right).$$

Using $\nabla_{\boldsymbol{h}_2}\mathcal{L} = \nabla_{\boldsymbol{y}}\mathcal{L}\boldsymbol{W}_1^\top$ and $\boldsymbol{y} = \boldsymbol{h}_2\boldsymbol{W}_1$, this statistic can be equivalently written as

$$\boldsymbol{s} = \frac{1}{d}\,\mathrm{sum}_{\mathrm{cols}}\left(\nabla_{\boldsymbol{y}}\mathcal{L} \odot \boldsymbol{y}\right).$$

This identity changes where the statistic is computed. Instead of launching a standalone RMSNorm backward kernel to read $\boldsymbol{h}_2$ and $\nabla_{\boldsymbol{h}_2}\mathcal{L}$, we accumulate the same row-wise quantity at a boundary where $\boldsymbol{y}$ and $\nabla_{\boldsymbol{y}}\mathcal{L}$ are already available, thereby exposing the computation to epilogue fusion.

In consecutive Transformer patterns, each GEMM epilogue can therefore accumulate the row-wise statistic needed by the preceding RMSNorm backward. The RMSNorm weight gradient is handled similarly by emitting tile partials for the reduction across rows. Overall, RMSNorm backward becomes GEMM–epilogue kernels plus lightweight auxiliary reductions over tile partials. We give the full derivation and kernel organization in Section A.2, with benchmarks in Figure 11.

3.3 Implementation

We implement CODA on top of CuTeDSL, which provides Python-level kernel authoring while retaining low-level control over details such as layouts and memory movement.

Data movement. Vector loads handle small broadcast operands such as RMSNorm weights. These values are staged once in shared memory and reused across subtiles. Tile loads handle larger operands such as residual activations, and uses Tensor Memory Accelerator transfers between global memory and shared memory, allowing data movement to overlap with epilogue computation. Stores follow similar tile-granular path for transformed outputs, saved intermediates, and reduction partials.

Local computation. Pairwise maps act on neighboring features, covering dimension-preserving operations such as RoPE, dimension-reducing operations such as SwiGLU, and dimension-expanding operations in the backward pass. When a map changes the feature dimension, the epilogue performs the corresponding local layout adjustment, such as compacting dimension-reducing outputs or packing pairs of 16-bit values for dimension-expanding outputs.

Reductions. Tile-wise reductions follow the ownership of GEMM output fragments. Row-wise reductions are accumulated by the warp that owns the row. Column-wise reductions may span multiple warps, so each warp first produces a partial result, and these partials are combined through shared memory.

3.3.1 LLM-Oriented Authoring

CODA is designed both for LLM workloads and for LLM-assisted authoring. Rather than asking a model to synthesize arbitrary CUDA or discover a hardware schedule from scratch, CODA exposes a constrained space of epilogue programs around expert-designed GEMM mainloops. Its primitives already encode efficient implementation strategies, so LLM-based authoring becomes a problem of composing vector loads, tile loads, pairwise maps, reductions, and stores for a given Transformer computation. This lightweight use of LLMs is complementary to prior work on kernel generation, which often relies on orchestration, search, execution feedback, or post-training [12, 16, 3, 10, 24].

Because CuTeDSL is relatively new, current models have limited exposure to its idioms. We therefore provide curated demonstrations for each abstraction. In practice, the repository itself acts as a growing demonstration set, with new kernels being written by adapting and composing existing examples.

Compositions. Transformer kernels often combine several epilogue operations, such as residual addition and RMSNorm scaling. Monolithic fused epilogues lead to large, repetitive implementations that are difficult to place in context. CODA instead represents each epilogue as a composition of reusable primitives: the LLM specifies the local epilogue program, while the library supplies the fixed GEMM mainloop and implementation pattern for each primitive. New fused kernels are therefore assembled from reusable building blocks instead of being rewritten from scratch.

4 Experiments

After the reparameterizations in Section 3.2, we obtain a compact benchmark suite of GEMM-plus-epilogue kernels spanning the Transformer++ forward and backward pass (Section C.1). The suite covers nearly all computation outside attention, embeddings, auxiliary reductions, and lightweight glue operations. We evaluate two implementations. `CODA (LLM)` uses Claude Code to generate most kernels from a written specification, curated examples, and a running log of implementation tips, with lightweight human supervision. `CODA (human)` is written by human programmers using the same high-level reparameterizations, but without access to the exact CODA primitive set.

We compare against cuBLAS with `torch.compile`, as well as optimized LLM kernel libraries including Liger Kernel [6] and FlashInfer [23]. Because our reparameterized kernels do not always have one-to-one counterparts in existing libraries, we compose the closest available optimized primitives and fall back to PyTorch operators as needed. We apply `torch.compile` to each method when compatible. Additional setup details are given in Section C.2.

Kernel Benchmarks. We first evaluate CODA at the individual-kernel level. Unless otherwise noted, we use square GEMM shapes with $M{=}N{=}K \in \{4096, 8192\}$. For cross-entropy kernels, we set

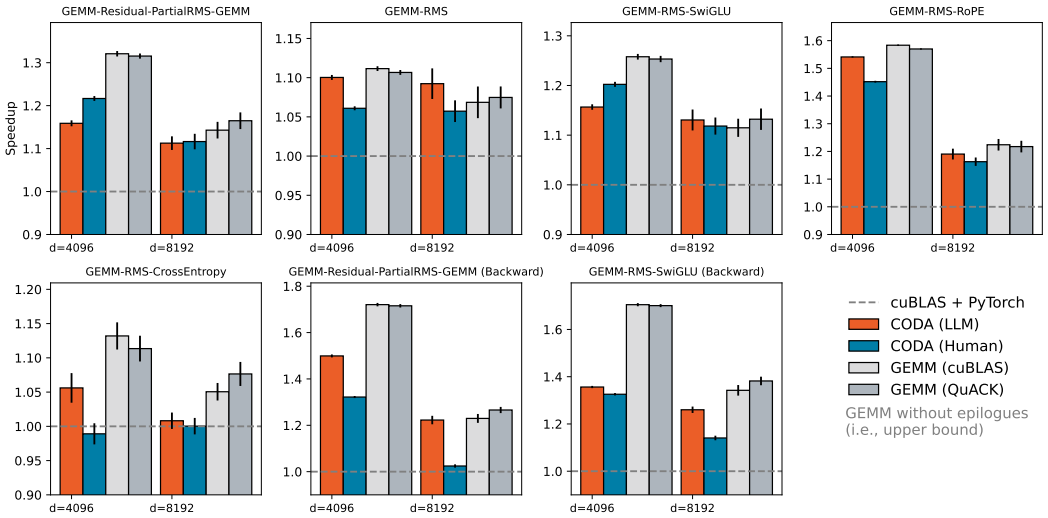

Figure 10: Kernel-level speedups on reparameterized Transformer kernels relative to cuBLAS with `torch.compile`. Raw GEMM baselines using PyTorch/cuBLAS and QuACK are included as reference ceilings, since they execute only the matrix multiplication and no epilogue work.

the vocabulary dimension to 32768. For RoPE kernels, we use $N_{\text{rope}}{=}3N$ to account for QKV-style projections and use precomputed cos and sin tables.[4] For kernels that emit partial reductions, we benchmark only the fused GEMM kernel with reduction tile size 128; auxiliary reductions are included in the block-level benchmarks below. We benchmark functions using Triton's `do_bench` and show means and standard deviations across 30 runs.

We evaluate two groups of kernels from Section C.1. The first group consists of standard Transformer-style kernels, such as GEMM with RoPE, SwiGLU, or cross-entropy epilogues. These kernels have close counterparts in existing libraries, so we compare against cuBLAS with `torch.compile`, Liger Kernels, and FlashInfer when applicable. Results are shown in Figure 8.

The second group consists of reparameterized Transformer forward and backward kernels, which generally do not have one-to-one equivalents in existing libraries. For these kernels, we primarily compare against cuBLAS with `torch.compile`, and additionally report raw GEMM from PyTorch/cuBLAS and QuACK.[5] These raw GEMM omit epilogue work and therefore serve as reference ceilings for the attainable throughput. Results are shown in Figure 10.

Block Benchmarks. We next benchmark kernel sequences corresponding to Transformer sublayers and full layers, which we call *blocks*. We use hidden sizes in $\{2048, 4096, 8192\}$, roughly matching 1B, 7B, and 70B model scales, with FFN expansion rate $8/3$ rounded to multiples of 256 and vocabulary size 32768. Unlike isolated kernel benchmarks, these measurements include auxiliary reductions and lightweight glue operations.

For the forward pass, we compare each reparameterized sequence against the closest available sequence of optimized operators. For the backward pass, the reparameterization changes the dependency structure: each sublayer emits partial statistics needed by the preceding RMSNorm backward. Individual backward sublayers therefore do not have direct PyTorch counterparts, so we report backward results at the full-layer level. In `CODA`, a layer consists of two consecutive GEMM-Residual-RMSNorm-GEMM blocks covering the SwiGLU and RoPE paths. Results are shown in Figure 11.

[4] For `CODA`, we additionally pre-broadcast and extend these tables across batch, head, and QKV dimensions to avoid in-kernel branching, at the cost of additional input traffic.

[5] https://github.com/Dao-AILab/quack

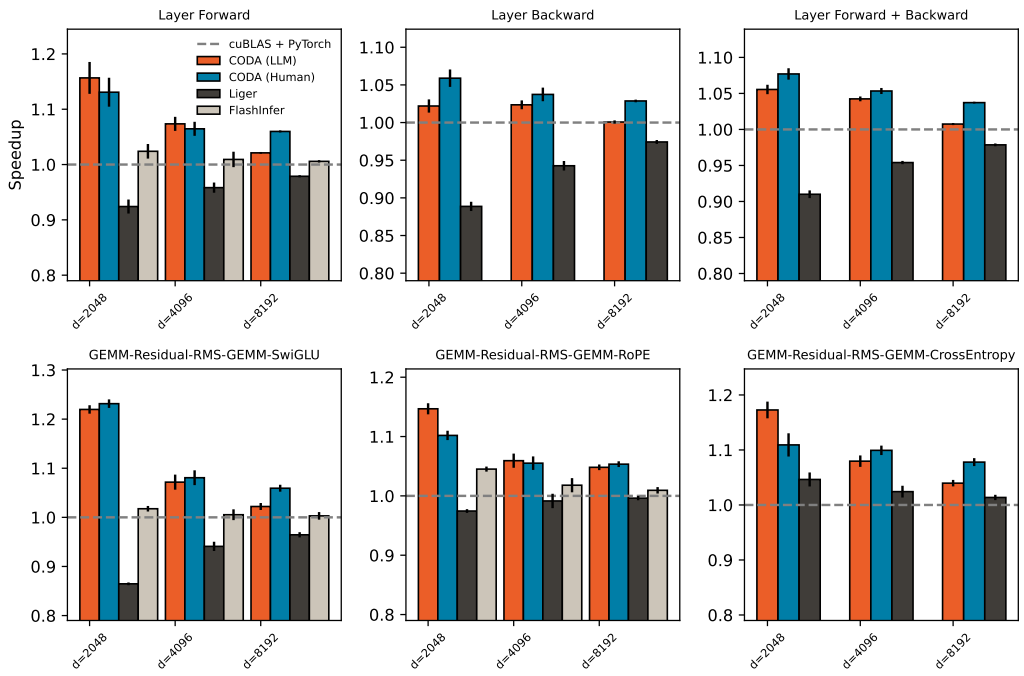

Figure 11: Block-level speedups for reparameterized Transformer kernel sequences, including auxiliary reductions and lightweight glue operations. Here, a layer denotes two consecutive GEMM-Residual-RMSNorm-GEMM blocks with the SwiGLU and RoPE activations, respectively.

5 Conclusion and Limitations

CODA reparameterizes much of Transformer computation as GEMM epilogues, reducing memory-bound overhead while preserving GEMM efficiency. Its constrained abstraction supports high-performance kernels authored by both humans and LLMs.

Limitations. Our reparameterizations target a common Transformer architecture; extending them to broader model families is future work. CODA currently focuses on single-GPU kernels and does not yet address distributed execution. Finally, while reparameterization improves efficiency, it can obscure module boundaries and algorithmic semantics, making integration with framework-level abstractions more challenging.

Acknowledgment

We thank Beshr Islam Bouli, Kaiming Cheng, Xinle Cheng, Ryan Chin, Tarushii Goel, Wentao Guo, Lucas Torroba Hennigen, Alicia Li, Mayank Mishra, Jyothish Pari, Caiming Xiong, Nicholas Yap, Tianyuan Zhang, and Adam Zweiger for helpful discussion. We gratefully acknowledge the support of the Schmidt Sciences AI2050 fellowship, the Google ML and Systems Junior Faculty Awards, the Google Research Scholar program, and the National Science Foundation (Award #2441872).

References

[1] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM international conference on architectural support for programming languages and operating systems, volume 2*, pages 929–947, 2024.

[2] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th*

USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 578–594, 2018.

[3] W. Chen, J. Zhu, Q. Fan, Y. Ma, and A. Zou. Cuda-llm: Llms can write efficient cuda kernels. *arXiv preprint arXiv:2506.09092*, 2025.

[4] Z. Chen, A. Kerr, R. Cai, J. Kosaian, H. Wu, Y. Ding, and Y. Xie. Evt: Accelerating deep learning training with epilogue visitor tree. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 301–316, 2024.

[5] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[6] P.-L. Hsu, Y. Dai, V. Kothapalli, Q. Song, S. Tang, S. Zhu, S. Shimizu, S. Sahni, H. Ning, and Y. Chen. Liger kernel: Efficient triton kernels for llm training. *arXiv preprint arXiv:2410.10989*, 2024.

[7] A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, and T. Hoefler. Data movement is all you need: A case study on optimizing transformers. *Proceedings of Machine Learning and Systems*, 3:711–732, 2021.

[8] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.

[9] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626, 2023.

[10] R. T. Lange, Q. Sun, A. Prasad, M. Faldor, Y. Tang, and D. Ha. Towards robust agentic cuda kernel benchmarking, verification, and optimization. *arXiv preprint arXiv:2509.14279*, 2025.

[11] W. Liang, T. Liu, L. Wright, W. Constable, A. Gu, C.-C. Huang, I. Zhang, W. Feng, H. Huang, J. Wang, et al. Torchtitan: One-stop pytorch native solution for production ready llm pre-training. *arXiv preprint arXiv:2410.06511*, 2024.

[12] A. Ouyang, S. Guo, S. Arora, A. L. Zhang, W. Hu, C. Ré, and A. Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*, 2025.

[13] B. Spector, J. Juravsky, S. Sul, O. Dugan, D. Lim, D. Fu, S. Arora, and C. Ré. Look ma, no bubbles! designing a low-latency megakernel for llama-1b, 2025.

[14] B. F. Spector, S. Arora, A. Singhal, D. Y. Fu, and C. Ré. Thunderkittens: Simple, fast, and adorable ai kernels. *arXiv preprint arXiv:2410.20399*, 2024.

[15] J. Su, M. Ahmed, Y. Lu, S. Pan, W. Bo, and Y. Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

[16] S. Su, X. Sun, X. Li, A. Wang, J. Li, and C. Shum. Cuda-l2: Surpassing cublas performance for matrix multiplication through reinforcement learning. *arXiv preprint arXiv:2512.02551*, 2025.

[17] S. H. Sul, S. Arora, B. F. Spector, and C. Ré. Parallelkittens: Systematic and practical simplification of multi-gpu ai kernels. *arXiv preprint arXiv:2511.13940*, 2025.

[18] V. Thakkar, P. Ramani, C. Cecka, A. Shivam, H. Lu, E. Yan, J. Kosaian, M. Hoemmen, H. Wu, A. Kerr, M. Nicely, D. Merrill, D. Blasig, A. Atluri, F. Qiao, P. Majcher, P. Springer, M. Hohnerbach, J. Wang, and M. Gupta. CUTLASS, Jan. 2023. URL https://github.com/NVIDIA/cutlass.

[19] P. Tillet, H.-T. Kung, and D. 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*, pages 10–19, 2019.

[20] L. Wang, Y. Cheng, Y. Shi, Z. Tang, Z. Mo, W. Xie, L. Ma, Y. Xia, J. Xue, F. Yang, et al. Tilelang: A composable tiled programming model for ai systems. *arXiv preprint arXiv:2504.17577*, 2025.

[21] E. Wijmans, B. Huval, A. Hertzberg, V. Koltun, and P. Krähenbühl. Cut your losses in large-vocabulary language models. In *International Conference on Learning Representations*, 2025.

[22] M. Wu, X. Cheng, S. Liu, C. Shi, J. Ji, M. K. Ao, P. Velliengiri, X. Miao, O. Padon, and Z. Jia. Mirage: A {Multi-Level} superoptimizer for tensor programs. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 21–38, 2025.

[23] Z. Ye, L. Chen, R. Lai, W. Lin, Y. Zhang, S. Wang, T. Chen, B. Kasikci, V. Grover, A. Krishnamurthy, et al. Flashinfer: Efficient and customizable attention engine for llm inference serving. *Proceedings of Machine Learning and Systems*, 7, 2025.

[24] M. Yuksekgonul, D. Koceja, X. Li, F. Bianchi, J. McCaleb, X. Wang, J. Kautz, Y. Choi, J. Zou, C. Guestrin, et al. Learning to discover at test time. *arXiv preprint arXiv:2601.16175*, 2026.

[25] L. Zheng, L. Yin, Z. Xie, C. L. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

A Backward Pass

A.1 Tile-wise Epilogue

Partition the GEMM output $\boldsymbol{h}$ into tiles $\boldsymbol{h}_{[i,j]}$. A tile-wise epilogue applies an independent transformation to each tile:

$$\boldsymbol{h} = \boldsymbol{x}\boldsymbol{W}_0, \qquad \boldsymbol{h}' = \begin{bmatrix} \boldsymbol{f}[0,0]\left(\boldsymbol{h}[0,0]\right) & \cdots & \boldsymbol{f}[0,N]\left(\boldsymbol{h}[0,N]\right) \\ \vdots & \ddots & \vdots \\ \boldsymbol{f}[M,0]\left(\boldsymbol{h}[M,0]\right) & \cdots & \boldsymbol{f}[M,N]\left(\boldsymbol{h}[M,N]\right) \end{bmatrix}, \qquad \boldsymbol{y} = \boldsymbol{h}'\boldsymbol{W}_1.$$

The backward pass has the same block structure.

$$\nabla_{\boldsymbol{h}'}\mathcal{L} = \nabla_{\boldsymbol{y}}\mathcal{L}\boldsymbol{W}_1^\top, \qquad \nabla_{\boldsymbol{h}}\mathcal{L} = \begin{bmatrix} \boldsymbol{g}[0,0]\left(\nabla_{\boldsymbol{h}'}\mathcal{L}[0,0]\right) & \cdots & \boldsymbol{g}[0,N]\left(\nabla_{\boldsymbol{h}'}\mathcal{L}[0,N]\right) \\ \vdots & \ddots & \vdots \\ \boldsymbol{g}[M,0]\left(\nabla_{\boldsymbol{h}'}\mathcal{L}[M,0]\right) & \cdots & \boldsymbol{g}[M,N]\left(\nabla_{\boldsymbol{h}'}\mathcal{L}[M,N]\right) \end{bmatrix}, \qquad \nabla_{\boldsymbol{x}}\mathcal{L} = \nabla_{\boldsymbol{h}}\mathcal{L}\boldsymbol{W}_0^\top.$$

Here each $\boldsymbol{g}[i,j]$ is the local backward rule for the corresponding tile:

$$\boldsymbol{g}[i,j](\Delta) = \mathrm{unvec}\Big(\boldsymbol{J}_{[i,j]}^\top \mathrm{vec}(\Delta)\Big), \qquad \boldsymbol{J}_{[i,j]} = \frac{\partial \,\mathrm{vec}\left(\boldsymbol{f}[i,j]\left(\boldsymbol{h}[i,j]\right)\right)}{\partial \,\mathrm{vec}\left(\boldsymbol{h}[i,j]\right)}.$$

Thus, each gradient tile depends only on the corresponding forward tile and upstream gradient tile. No cross-tile communication is introduced, so the backward operation remains tile-local and can be implemented as a GEMM epilogue. As shown in Figure 9, the only structural change is the direction of fusion: forward epilogues are fused into the GEMM that produces their input, while backward epilogues are fused into the GEMM that produces the gradient with respect to their output.

A.2 GEMM-RMSNorm-GEMM Backward Pass

We now describe the backward pass for the GEMM–epilogue–RMSNorm–GEMM pattern. RMSNorm is the first case where the backward pass is not purely tile-local. The reason is simple: RMSNorm contains a row-wise normalization factor, so its backward pass needs a row-wise statistic. In addition, the RMSNorm weight $\boldsymbol{\gamma}$ is shared across rows, so its gradient requires a reduction across the row dimension. The goal of this section is to show that these are the only non-local pieces. Everything else can still be fused into GEMM epilogues, with the non-local pieces handled by lightweight reductions over tile partials.

Consider the forward computation

$$\boldsymbol{h}_0 = \boldsymbol{x}\boldsymbol{W}_0, \qquad \boldsymbol{h}_1 = f(\boldsymbol{h}_0), \qquad \boldsymbol{h}_2 = \mathrm{RMSNorm}(\boldsymbol{h}_1, \boldsymbol{\gamma}), \qquad \boldsymbol{y} = \boldsymbol{h}_2\boldsymbol{W}_1.$$

Let $\boldsymbol{r}$ be the row-wise inverse RMS factor. We write $\overline{\boldsymbol{r}} = \boldsymbol{r}\mathbf{1}^\top$ and $\overline{\boldsymbol{\gamma}} = \mathbf{1}\boldsymbol{\gamma}^\top$ for the broadcasts of $\boldsymbol{r}$ and $\boldsymbol{\gamma}$ to the shape of $\boldsymbol{h}_1$. Then

$$\boldsymbol{h}_2 = \overline{\boldsymbol{r}} \odot \boldsymbol{h}_1 \odot \overline{\boldsymbol{\gamma}}.$$

Given the upstream gradient $\nabla_{\boldsymbol{y}}\mathcal{L}$, the first backward operation is the GEMM

$$\nabla_{\boldsymbol{h}_2}\mathcal{L} = \nabla_{\boldsymbol{y}}\mathcal{L}\boldsymbol{W}_1^\top.$$

The RMSNorm backward can be written as

$$\begin{aligned}
\nabla_{\boldsymbol{h}_1}\mathcal{L} &= \overline{\boldsymbol{r}} \odot \left(\nabla_{\boldsymbol{h}_2}\mathcal{L} \odot \overline{\boldsymbol{\gamma}} - \overline{\boldsymbol{r}} \odot \boldsymbol{h}_1 \odot \overline{\boldsymbol{s}}\right), \\
\nabla_{\boldsymbol{\gamma}}\mathcal{L} &= \mathrm{sum}_{\mathrm{rows}}\left(\nabla_{\boldsymbol{h}_2}\mathcal{L} \odot \boldsymbol{h}_1 \odot \overline{\boldsymbol{r}}\right),
\end{aligned}$$

where $\overline{\boldsymbol{s}} = \boldsymbol{s}\mathbf{1}^\top$ broadcasts one scalar per row. The row-wise statistic $\boldsymbol{s}$ is

$$\begin{aligned}
\boldsymbol{s} &= \frac{1}{d} \odot \boldsymbol{r} \odot \mathrm{sum}_{\mathrm{cols}}\left(\nabla_{\boldsymbol{h}_2}\mathcal{L} \odot \overline{\boldsymbol{\gamma}} \odot \boldsymbol{h}_1\right), \\
&= \frac{1}{d}\,\mathrm{sum}_{\mathrm{cols}}\left(\nabla_{\boldsymbol{h}_2}\mathcal{L} \odot \overline{\boldsymbol{r}} \odot \overline{\boldsymbol{\gamma}} \odot \boldsymbol{h}_1\right), \\
&= \frac{1}{d}\,\mathrm{sum}_{\mathrm{cols}}\left(\nabla_{\boldsymbol{h}_2}\mathcal{L} \odot \boldsymbol{h}_2\right),
\end{aligned}$$

where d is the hidden dimension. This expression identifies the two non-local operations in RMSNorm backward. The statistic $\boldsymbol{s}$ is a reduction across columns, producing one scalar per row. The weight gradient $\nabla_{\boldsymbol{\gamma}}\mathcal{L}$ is a reduction across rows, producing one scalar per hidden feature.

A standalone RMSNorm backward kernel would compute these reductions by reading activation-sized tensors. The key observation is that the row-wise statistic $\boldsymbol{s}$ can be moved to a different boundary. Using $\nabla_{\boldsymbol{h}_2}\mathcal{L} = \nabla_{\boldsymbol{y}}\mathcal{L}\,\boldsymbol{W}_1^\top$ and $\boldsymbol{y} = \boldsymbol{h}_2\boldsymbol{W}_1$, we have

$$\begin{aligned}
\boldsymbol{s} &= \frac{1}{d}\,\mathrm{sum}_{\mathrm{cols}}\left(\nabla_{\boldsymbol{h}_2}\mathcal{L} \odot \boldsymbol{h}_2\right), \\
&= \frac{1}{d}\,\mathrm{sum}_{\mathrm{cols}}\left((\nabla_{\boldsymbol{y}}\mathcal{L}\boldsymbol{W}_1^\top) \odot \boldsymbol{h}_2\right) \\
&= \frac{1}{d}\,\mathrm{diag}\left((\nabla_{\boldsymbol{y}}\mathcal{L}\boldsymbol{W}_1^\top)\,\boldsymbol{h}_2^\top\right) \\
&= \frac{1}{d}\,\mathrm{diag}\left(\nabla_{\boldsymbol{y}}\mathcal{L}\,(\boldsymbol{h}_2\boldsymbol{W}_1)^\top\right) \\
&= \frac{1}{d}\,\mathrm{sum}_{\mathrm{cols}}\left(\nabla_{\boldsymbol{y}}\mathcal{L} \odot (\boldsymbol{h}_2\boldsymbol{W}_1)\right) \\
&= \frac{1}{d}\,\mathrm{sum}_{\mathrm{output}}\left(\nabla_{\boldsymbol{y}}\mathcal{L} \odot \boldsymbol{y}\right).
\end{aligned}$$

Intuitively, the RMSNorm backward needs the inner product between an activation and its gradient along each row. The identity above says that this inner product can be computed either before or after the following GEMM. This lets us compute the statistic at a boundary where $\boldsymbol{y}$ and $\nabla_{\boldsymbol{y}}\mathcal{L}$ are already available.

This is useful because Transformer layers contain consecutive GEMM–epilogue–RMSNorm–GEMM patterns. During the backward pass of one pattern, the GEMM that produces $\nabla_{\boldsymbol{x}}\mathcal{L}$ already has access to both $\boldsymbol{x}$ and $\nabla_{\boldsymbol{x}}\mathcal{L}$. Since this $\boldsymbol{x}$ is the output of the preceding pattern, the epilogue of the current pattern can accumulate the RMSNorm statistic needed by the preceding pattern:

$$\widehat{\boldsymbol{s}}_{\mathrm{prev}} = \mathrm{reduceTile}_{\mathrm{cols}}\left(\boldsymbol{x} \odot \nabla_{\boldsymbol{x}}\mathcal{L}\right), \qquad \boldsymbol{s}_{\mathrm{prev}} = \frac{1}{d}\,\mathrm{reduce}\left(\widehat{\boldsymbol{s}}_{\mathrm{prev}}\right).$$

Thus, each pattern computes the row-wise RMSNorm backward statistic required by the pattern before it. The reduction is still present, but it is now a small reduction over tile partials rather than a standalone activation-sized RMSNorm backward kernel.

The RMSNorm weight gradient is handled similarly, except that its reduction is across rows rather than columns. We accumulate tile partials in the RMSNorm backward epilogue:

$$\widehat{\nabla_{\boldsymbol{\gamma}}\mathcal{L}} = \mathrm{reduceTile}_{\mathrm{rows}}\left(\nabla_{\boldsymbol{h}_2}\mathcal{L} \odot \boldsymbol{h}_1 \odot \overline{\boldsymbol{r}}\right), \qquad \nabla_{\boldsymbol{\gamma}}\mathcal{L} = \mathrm{reduce}_{\mathrm{rows}}\left(\widehat{\nabla_{\boldsymbol{\gamma}}\mathcal{L}}\right).$$

Putting these pieces together, the backward pass is organized as follows:

$$
\begin{aligned}
\text{GEMM 1:} \quad & \nabla_{\boldsymbol{h}_2}\mathcal{L} = \nabla_{\boldsymbol{y}}\mathcal{L}\boldsymbol{W}_1^\top, \\
\text{Epilogue 1:} \quad & \nabla_{\boldsymbol{h}_1}\mathcal{L} = \overline{\boldsymbol{r}} \odot \left(\nabla_{\boldsymbol{h}_2}\mathcal{L} \odot \overline{\boldsymbol{\gamma}} - \boldsymbol{h}_1 \odot \overline{\boldsymbol{r}} \odot \overline{\boldsymbol{s}}\right), \\
& \nabla_{\boldsymbol{h}_0}\mathcal{L} = g(\nabla_{\boldsymbol{h}_1}\mathcal{L}), \\
& \widehat{\nabla_{\boldsymbol{\gamma}}\mathcal{L}} = \mathrm{reduceTile}_{\mathrm{rows}}\left(\nabla_{\boldsymbol{h}_2}\mathcal{L} \odot \boldsymbol{h}_1 \odot \overline{\boldsymbol{r}}\right), \\
\\
\text{GEMM 2:} \quad & \nabla_{\boldsymbol{x}}\mathcal{L} = \nabla_{\boldsymbol{h}_0}\mathcal{L}\boldsymbol{W}_0^\top, \\
\text{Epilogue 2:} \quad & \widehat{\boldsymbol{s}}_{\mathrm{prev}} = \mathrm{reduceTile}_{\mathrm{cols}}\left(\boldsymbol{x} \odot \nabla_{\boldsymbol{x}}\mathcal{L}\right), \\
\\
\text{Auxiliary reductions:} \quad & \boldsymbol{s}_{\mathrm{prev}} = \frac{1}{d}\,\mathrm{reduce}_{\mathrm{cols}}\left(\widehat{\boldsymbol{s}}_{\mathrm{prev}}\right), \\
& \nabla_{\boldsymbol{\gamma}}\mathcal{L} = \mathrm{reduce}_{\mathrm{rows}}\left(\widehat{\nabla_{\boldsymbol{\gamma}}\mathcal{L}}\right).
\end{aligned}
$$

Here g denotes the tile-local backward rule for the epilogue f. The statistic $\boldsymbol{s}$ used in Epilogue 1 is assumed to have already been accumulated by the following pattern in the backward order.

Finally, the output of a pattern often passes through another epilogue before the next pattern begins:

$$
\boldsymbol{h}_0 = \boldsymbol{x}\boldsymbol{W}_0, \qquad \boldsymbol{h}_1 = f_0(\boldsymbol{h}_0), \qquad \boldsymbol{h}_2 = \mathrm{RMSNorm}(\boldsymbol{h}_1, \boldsymbol{\gamma}), \qquad \boldsymbol{h}_3 = \boldsymbol{h}_2\boldsymbol{W}_1 \qquad \boldsymbol{y} = f_1(\boldsymbol{h}_3).
$$

In this case, the statistic should be accumulated using the pre-epilogue tensor $\boldsymbol{h}_3$ and its gradient. The backward rule for f_1 is tile-local, so it can be fused before accumulating the statistic:

$$
\begin{aligned}
\text{GEMM 2:} \quad & \nabla_{\boldsymbol{x}}\mathcal{L} = \nabla_{\boldsymbol{h}_0}\mathcal{L}\boldsymbol{W}_0^\top, \\
\text{Epilogue 2:} \quad & \nabla_{\overleftarrow{\boldsymbol{h}_3}}\mathcal{L} = \overleftarrow{f_1}(\nabla_{\boldsymbol{x}}\mathcal{L}), \\
& \widehat{\boldsymbol{s}}_{\mathrm{prev}} = \mathrm{reduceTile}_{\mathrm{cols}}\left(\overleftarrow{\boldsymbol{h}_3} \odot \nabla_{\overleftarrow{\boldsymbol{h}_3}}\mathcal{L}\right)
\end{aligned}
$$

Here $\overleftarrow{\boldsymbol{h}_3}$ denotes the pre-epilogue GEMM output from the preceding pattern, and $\overleftarrow{f_1}$ denotes the local backward rule for its following epilogue. This case preserves the same structure: apply the local backward epilogue first, then accumulate the row-wise statistic from the pre-epilogue activation and its gradient.

Overall, this organization removes the activation-sized RMSNorm backward kernel. The remaining non-local work consists only of reductions over tile partials: a column reduction for the row-wise RMSNorm statistic, and a row reduction for the RMSNorm weight gradient. The GEMMs and local backward updates are fused into GEMM epilogues, preserving the same GEMM-plus-epilogue structure used in the forward pass.

B CODA

B.1 Epilogue Template

```
# --- Mainloop ---
# ...
# --- Pre-loop: visitor tree entry points ---
epilogue.consumer_begin(...)
epilogue.producer_begin(...)

# --- Main loop: epilogue tiles ---
for epi_idx in range(num_epi_tiles):
    # Per-tile visitor entry
    epilogue.consumer_begin_loop(...)
    # Producer-side async load (e.g., TMA)
    epilogue.producer_tma_load(...)
    # Core epilogue computation
    rD = load_accumulator_fragment(...)
    epilogue.consumer_visit(rD, ...)
    # Registers -> shared memory
    store_regs_to_smem(...)
    # Pre-store visitor hook
```

```
    epilogue.consumer_smem_store(...)
    tma_store_from_smem_to_gmem(...)
    epilogue.consumer_tma_store(...)
    # Per-tile visitor exit
    epilogue.consumer_end_loop(gmem_coord)

# --- Post-loop: visitor tree finalization ---
epilogue.consumer_end(...)
```

Listing 1: Epilogue Kernel Abstraction.

B.2 Epilogue Example

```
def _create_mean_sq_reduction_op(element_type, inv_block_size):
    """Create a reduction op that accumulates mean of squares: acc + val^2 *
     inv_block_size.

    The combine_fn squares each new element and scales by 1/block_size before adding
    to the accumulator. The warp-level reduction uses standard addition since partial
    sums are already accumulated and scaled.
    """
    init_value = element_type(0.)
    inv_bs = element_type(inv_block_size)

    _sq_combine = lambda x, y: x + y * y * inv_bs
    _add_wrp = lambda tree_x, tree_y: pytree.tree_map(operator.add, tree_x, tree_y)

    return BlockReductionOp(
        combine_fn=lambda tree_x, tree_y: pytree.tree_map(_sq_combine, tree_x, tree_y),
        reduce_ssa=None,
        reduce_wrp=lambda xs: pytree.tree_map(
            lambda x: cute.arch.warp_reduction(
                x,
                op=_add_wrp,
                threads_in_group=HOPPER_WARP_REDUCTION_WIDTH,
            ),
            xs,
        ),
        init_value=init_value,
    )


class EVTRowVecMulPostAct(EpilogueVisitorTree):
    """
    Loads a per-N row vector W (cp.async to smem, then s2r), multiplies the
    accumulator by W into a separate register tile, and stores that scaled
    tile to a side output mPostAct via TMA. tRS_rD itself is left unchanged
    so the main D output (the unscaled GEMM result) is unaffected.

    This mirrors the rowvec=norm_weight side-output path of trainstation's
    'gemm_partial_rms_fwd', kept local to this kernel rather than as a
    general-purpose rapier EVT.

    Inputs:
        - GEMM output (in registers): [M x N], unchanged by this op
        - mRowVec: [L, N]  - RMSNorm weight, broadcast along M

    Outputs:
        - mPostAct: [M x N] = D * W  (side output, written via TMA)
    """

    @struct_utils.mlir_namedtuple
    class EpilogueArguments(NamedTuple):
        mPostAct: cute.Tensor | None
        mRowVec: cute.Tensor | None

    @struct_utils.register_pytree_dataclass
    @dataclass
    class EpilogueParams(EpilogueVisitorTree.EpilogueParams):
        mPostAct: cute.Tensor | None
        mRowVec: cute.Tensor | None
        epi_tma_atom: cute.CopyAtom
        epi_gmem_layout: cutlass.utils.LayoutEnum
        epi_smem_layout_staged: cute.Layout

    @struct_utils.register_pytree_dataclass
    @dataclass
    class EpilogueTensorsSMem(EpilogueVisitorTree.EpilogueTensorsSMem):
        sPostAct: cute.Tensor | None
```

```
        sRowVec: cute.Tensor | None

    @struct_utils.register_pytree_dataclass
    @dataclass
    class EpilogueTensors(EpilogueVisitorTree.EpilogueTensors):
        tDsPostAct: cute.Tensor
        tDgPostAct: cute.Tensor
        tRS_sPostAct: cute.Tensor
        epi_tma_atom: cute.CopyAtom
        tiled_copy_postact_r2s: cute.TiledCopy
        tDsRowVec: cute.Tensor | None

    @struct_utils.register_pytree_dataclass
    @dataclass
    class EpilogueTensorsLoop(EpilogueVisitorTree.EpilogueTensorsLoop):
        tDsPostAct: cute.Tensor
        tDgPostAct: cute.Tensor
        tRS_rPostAct: cute.Tensor | None
        tRS_sPostAct: cute.Tensor
        epi_tma_atom: cute.CopyAtom
        tiled_copy_postact_r2s: cute.TiledCopy
        tDrRowVec_epi: cute.Tensor | None

    @struct_utils.register_pytree_dataclass
    @dataclass
    class EpiloguePipelines(EpilogueVisitorTree.EpiloguePipelines):
        pass

    def __init__(
        self,
        acc_dtype: type[cute.Numeric],
        post_act_dtype: type[cute.Numeric],
        tile_shape_mnk: tuple[int, int, int],
        buffer_align_bytes: int,
    ) -> None:
        super().__init__()
        self.arch = 90
        self.acc_dtype = acc_dtype
        self.post_act_dtype = post_act_dtype
        self.container_dtype = post_act_dtype
        self.tile_shape_mnk = tile_shape_mnk
        self.buffer_align_bytes = buffer_align_bytes

    @cute.jit
    def to_underlying_arguments(
        self,
        epi_tile: cute.Tile,
        epi_stage: int,
        epi_load_stage: int,
        epi_args: EpilogueArguments,
    ) -> EpilogueParams:

        if cutlass.const_expr(epi_args.mPostAct is not None):
            mPostAct = misc_utils.static_assert_is_Tensor(epi_args.mPostAct)
            misc_utils.static_assert(get_dtype(mPostAct) is self.container_dtype)
            (
                epi_gmem_layout,
                epi_smem_layout_staged,
                epi_tma_atom,
                epi_tma_tensor,
            ) = epilogue_utils.prepare_tma(
                tma_op="s2g",
                epi_tile=epi_tile,
                epi_stage=epi_stage,
                epi_tensor=mPostAct,
            )

        if cutlass.const_expr(epi_args.mRowVec is not None):
            misc_utils.static_assert(epi_args.mPostAct is not None)
            mRowVec = misc_utils.static_assert_is_Tensor(epi_args.mRowVec)
            mRowVec = layout_utils.assumed_align_stride(
                mRowVec,
                assumed_align=4,
            )
        else:
            mRowVec = None

        return self.EpilogueParams(
            mPostAct=epi_tma_tensor,
            mRowVec=mRowVec,
            epi_tma_atom=epi_tma_atom,
```

```
            epi_gmem_layout=epi_gmem_layout,
            epi_smem_layout_staged=epi_smem_layout_staged,
        )

    @cute.jit
    def prefetch_tma_descriptors(
        self,
        epi_params: EpilogueParams,
    ) -> None:
        cute.nvgpu.cpasync.prefetch_descriptor(epi_params.epi_tma_atom)

    @cute.jit
    def consumer_begin(
        self,
        tiled_copy_r2s: cute.TiledCopy,
        tile_coord_mnkl: cute.Coord,
        tidx: cute.Int32,
        tiled_mma: cute.TiledMma,
        tRS_rD_layout: cute.Layout,
        epi_tile: cute.Tile,
        epi_num_threads: int,
        epi_num_matrices: int,
        epi_barrier: cutlass.pipeline.NamedBarrier,
        epi_params: EpilogueParams,
        epi_tensors_smem: EpilogueTensorsSMem,
    ) -> EpilogueTensors:

        tile_M = self.tile_shape_mnk[0]
        tile_N = self.tile_shape_mnk[1]
        m_idx, n_idx, _, batch_idx = tile_coord_mnkl
        thr_copy_r2s = tiled_copy_r2s.get_slice(tidx)

        # Side output (PostAct) TMA setup
        mPostAct = misc_utils.static_assert_is_Tensor(epi_params.mPostAct)
        sPostAct = misc_utils.static_assert_is_Tensor(epi_tensors_smem.sPostAct)
        tiled_copy_postact_r2s, _, tRS_sPostAct = epilogue_utils.prepare_copy_r2s_sm90(
            tiled_copy_r2s=tiled_copy_r2s,
            tidx=tidx,
            dst=sPostAct,
            epi_layout=epi_params.epi_gmem_layout,
            epi_dtype=self.container_dtype,
            acc_dtype=self.acc_dtype,
        )
        gPostAct = mPostAct[None, None, batch_idx]
        gPostAct = cute.local_tile(gPostAct, (tile_M, tile_N), (m_idx, n_idx))
        gPostAct = cute.zipped_divide(gPostAct, epi_tile)

        tDsPostAct, tDgPostAct = cute.nvgpu.cpasync.tma_partition(
            atom=epi_params.epi_tma_atom,
            cta_coord=0,
            cta_layout=cute.make_layout(1),
            smem_tensor=cute.group_modes(sPostAct, 0, cute.rank(sPostAct) - 1),
            gmem_tensor=cute.group_modes(gPostAct, 0, cute.rank(gPostAct) - 1),
        )

        # RowVec cp.async load (per-N broadcast across M)
        if cutlass.const_expr(epi_params.mRowVec is not None):
            mRowVec = misc_utils.static_assert_is_Tensor(epi_params.mRowVec)
            sRowVec = misc_utils.static_assert_is_Tensor(epi_tensors_smem.sRowVec)
            mRowVec = mRowVec[batch_idx, None]
            gRowVec = cute.local_tile(mRowVec, (tile_N,), (n_idx,))
            cRowVec = cute.make_identity_tensor(tile_N)
            limit_n = min(mRowVec.shape[0] - n_idx * tile_N, tile_N)
            memory_utils.g2s_copy_1d(
                src=gRowVec,
                dst=sRowVec,
                crd=cRowVec,
                shape=(limit_n,),
                num_threads=epi_num_threads,
                thread_index=tidx,
            )
            sRowVec_view_layout = cute.make_layout(
                shape=(tile_M, tile_N),
                stride=(0, 1),
            )
            sRowVec_view = cute.make_tensor(
                iterator=sRowVec.iterator,
                layout=sRowVec_view_layout,
            )
            tDsRowVec = thr_copy_r2s.partition_S(
                cute.flat_divide(sRowVec_view, epi_tile)
```

```
            )
            cute.arch.cp_async_commit_group()
            cute.arch.cp_async_wait_group(0)
            epi_barrier.arrive_and_wait()
        else:
            tDsRowVec = None

        return self.EpilogueTensors(
            tDsPostAct=tDsPostAct,
            tDgPostAct=tDgPostAct,
            tRS_sPostAct=tRS_sPostAct,
            epi_tma_atom=epi_params.epi_tma_atom,
            tiled_copy_postact_r2s=tiled_copy_postact_r2s,
            tDsRowVec=tDsRowVec,
        )

    @cute.jit
    def consumer_end(
        self,
        tiled_copy_r2s: cute.TiledCopy,
        tile_coord_mnkl: cute.Coord,
        tidx: cute.Int32,
        shape_mnk: cute.Shape,
        epi_tile: cute.Tile,
        epi_num_threads: int,
        epi_barrier: cutlass.pipeline.NamedBarrier,
        epi_params: EpilogueParams,
        epi_tensors: EpilogueTensors,
        epi_tensors_smem: EpilogueTensorsSMem,
    ) -> None:
        pass

    @cute.jit
    def consumer_begin_loop(
        self,
        epi_coord: cute.Coord,
        epi_params: EpilogueParams,
        epi_tensors: EpilogueTensors,
        epi_pipelines: EpiloguePipelines,
    ) -> tuple[EpilogueTensorsLoop, EpiloguePipelines]:

        if cutlass.const_expr(epi_tensors.tDsRowVec is not None):
            tDsRowVec = misc_utils.static_assert_is_Tensor(epi_tensors.tDsRowVec)
            tDsRowVec_cur = cute.group_modes(tDsRowVec, 3, cute.rank(tDsRowVec))
            tDsRowVec_cur = tDsRowVec_cur[None, None, None, epi_coord]
            tDrRowVec_cvt = memory_utils.s2r_copy_1d(tDsRowVec_cur, dtype=self.acc_dtype)
        else:
            tDrRowVec_cvt = None

        return (
            self.EpilogueTensorsLoop(
                tDsPostAct=epi_tensors.tDsPostAct,
                tDgPostAct=epi_tensors.tDgPostAct,
                tRS_rPostAct=None,
                tRS_sPostAct=epi_tensors.tRS_sPostAct,
                epi_tma_atom=epi_tensors.epi_tma_atom,
                tiled_copy_postact_r2s=epi_tensors.tiled_copy_postact_r2s,
                tDrRowVec_epi=tDrRowVec_cvt,
            ),
            self.EpiloguePipelines(),
        )

    @cute.jit
    def consumer_visit(
        self,
        tRS_rD: cute.Tensor,
        shape_mnk: cute.Shape,
        epi_params: EpilogueParams,
        epi_tensors_loop: EpilogueTensorsLoop,
    ) -> EpilogueTensorsLoop:

        tRS_rPostAct = creation_utils.allocate_tensor_like(
            tensor=tRS_rD,
            memspace="rmem",
            smem_allocator=None,
            dtype=self.acc_dtype,
        )
        if cutlass.const_expr(self.arch < 100):
            if cutlass.const_expr(epi_tensors_loop.tDrRowVec_epi is not None):
                tDrRowVec_epi = misc_utils.static_assert_is_Tensor(epi_tensors_loop.
    tDrRowVec_epi)
```

```
                for i in cutlass.range_constexpr(cute.size(tRS_rPostAct)):
                    tRS_rPostAct[i] = tRS_rD[i] * tDrRowVec_epi[i]
            else:
                for i in cutlass.range_constexpr(cute.size(tRS_rPostAct)):
                    tRS_rPostAct[i] = tRS_rD[i]
        else:
            raise NotImplementedError

        tRS_rPostAct = dtype_utils.convert(
            tRS_rPostAct,
            dtype=self.post_act_dtype,
        )

        return self.EpilogueTensorsLoop(
            tDsPostAct=epi_tensors_loop.tDsPostAct,
            tDgPostAct=epi_tensors_loop.tDgPostAct,
            tRS_rPostAct=tRS_rPostAct,
            tRS_sPostAct=epi_tensors_loop.tRS_sPostAct,
            epi_tma_atom=epi_tensors_loop.epi_tma_atom,
            tiled_copy_postact_r2s=epi_tensors_loop.tiled_copy_postact_r2s,
            tDrRowVec_epi=epi_tensors_loop.tDrRowVec_epi,
        )

    @cute.jit
    def consumer_smem_store(
        self,
        epi_coord: cute.Coord,
        epi_buffer: cute.Int32,
        epi_params: EpilogueParams,
        epi_tensors_loop: EpilogueTensorsLoop,
    ) -> None:
        tiled_copy = epi_tensors_loop.tiled_copy_postact_r2s
        tRS_rPostAct = misc_utils.static_assert_is_Tensor(epi_tensors_loop.tRS_rPostAct)
        tRS_sPostAct = misc_utils.static_assert_is_Tensor(epi_tensors_loop.tRS_sPostAct)
        src = tiled_copy.retile(tRS_rPostAct)
        dst = tRS_sPostAct[None, None, None, epi_buffer]
        cute.copy(atom=tiled_copy, src=src, dst=dst)

    @cute.jit
    def consumer_tma_store(
        self,
        epi_coord: cute.Coord,
        epi_buffer: cute.Int32,
        epi_params: EpilogueParams,
        epi_tensors_loop: EpilogueTensorsLoop,
    ) -> None:
        atom = epi_tensors_loop.epi_tma_atom
        tDsPostAct = misc_utils.static_assert_is_Tensor(epi_tensors_loop.tDsPostAct)
        tDgPostAct = misc_utils.static_assert_is_Tensor(epi_tensors_loop.tDgPostAct)
        src = tDsPostAct[None, epi_buffer]
        dst = tDgPostAct[None, epi_coord]
        cute.copy(atom=atom, src=src, dst=dst)

    @cute.jit
    def get_smem_struct(
        self,
        epi_load_stage: int,
        epi_num_threads: int,
        epi_params: EpilogueParams,
    ) -> type[EpilogueSharedStorage]:

        if cutlass.const_expr(epi_params.mPostAct is not None):
            post_act_smem_size = cute.cosize(epi_params.epi_smem_layout_staged)
        else:
            post_act_smem_size = 0

        if cutlass.const_expr(epi_params.mRowVec is not None):
            mRowVec = misc_utils.static_assert_is_Tensor(epi_params.mRowVec)
            row_vec_dtype = get_dtype(mRowVec)
            row_vec_smem_size = epilogue_utils.get_smem_size_vector(
                mTensor=mRowVec,
                epi_tile=self.tile_shape_mnk[1],
                epi_num_threads=epi_num_threads,
            )
        else:
            row_vec_dtype = cute.Float32
            row_vec_smem_size = 0

        @cute.struct
        class SharedStorage(EpilogueSharedStorage):
```

```
            sPostAct: cute.struct.Align[cute.struct.MemRange[self.container_dtype,
    post_act_smem_size], self.buffer_align_bytes]
            sRowVec: cute.struct.Align[cute.struct.MemRange[row_vec_dtype,
    row_vec_smem_size], 16]

        return SharedStorage

    @cute.jit
    def get_smem_tensors(
        self,
        storage: EpilogueSharedStorage,
        epi_num_threads: int,
        epi_params: EpilogueParams,
    ) -> EpilogueTensorsSMem:

        if cutlass.const_expr(epi_params.mPostAct is not None):
            sPostAct = storage.sPostAct.get_tensor(
                epi_params.epi_smem_layout_staged.outer,
                swizzle=epi_params.epi_smem_layout_staged.inner,
            )
        else:
            sPostAct = None

        if cutlass.const_expr(epi_params.mRowVec is not None):
            sRowVec_layout = cute.make_layout(self.tile_shape_mnk[1])
            sRowVec = storage.sRowVec.get_tensor(sRowVec_layout)
        else:
            sRowVec = None

        return self.EpilogueTensorsSMem(
            sPostAct=sPostAct,
            sRowVec=sRowVec,
        )

    @cute.jit
    def get_smem_bytes_per_stage(
        self,
        epi_tile: cute.Tile,
        epi_num_threads: int,
        epi_args: EpilogueArguments,
    ) -> tuple[int, int, int]:
        epi_smem_bytes_fixed = 0
        epi_smem_bytes_per_stage_cst = 0
        epi_smem_bytes_per_stage_pld = 0

        if cutlass.const_expr(epi_args.mPostAct is not None):
            mPostAct = misc_utils.static_assert_is_Tensor(epi_args.mPostAct)
            misc_utils.static_assert(get_dtype(mPostAct) is self.container_dtype)
            epi_smem_bytes_per_stage_cst = epi_smem_bytes_per_stage_cst + (
                epilogue_utils.get_epi_smem_bytes_per_stage_matrix(
                    mTensor=mPostAct,
                    epi_tile=epi_tile,
                )
            )

        if cutlass.const_expr(epi_args.mRowVec is not None):
            mRowVec = misc_utils.static_assert_is_Tensor(epi_args.mRowVec)
            epi_smem_bytes_fixed = epi_smem_bytes_fixed + (
                epilogue_utils.get_epi_smem_bytes_per_stage_fixed_vector(
                    mTensor=mRowVec,
                    epi_tile=self.tile_shape_mnk[1],
                    epi_num_threads=epi_num_threads,
                )
            )

        return (
            epi_smem_bytes_fixed,
            epi_smem_bytes_per_stage_cst,
            epi_smem_bytes_per_stage_pld,
        )


def prepare_epilogue(
    shape_mnkl: tuple[int, int, int, int],
    tile_shape_mn: tuple[int, int],
    C: torch.Tensor,
    S: torch.Tensor,
    W: torch.Tensor,
    O: torch.Tensor,
) -> tuple[
    Callable[..., EpilogueVisitorTree],
```

```
    EpilogueVisitorTree.EpilogueArguments,
    dict,
    tuple,
]:
    """Prepare epilogue for GEMM with residual, partial mean-of-squares, and
    fused per-N RMSNorm-weight scaling - mirrors trainstation's `gemm_partial_rms_fwd`.

    Composes three EVT visitors:
        1. EVTResidual: D = acc + C
        2. EVTColBlockReductionStore: S[m, nb] = mean(D[m, nb*bs:(nb+1)*bs]^2)
        3. EVTRowVecMulPostAct (local): O[m, n] = D[m, n] * W[n], side output via TMA

    The partial sum-of-squares is computed on the *unscaled* D, so a downstream
    rstd reduction sees the GEMM output before W is applied. tRS_rD is preserved
    so the main D output is also unscaled.

    Args:
        shape_mnkl: Problem shape (M, N, K, L) where L is batch dimension.
        tile_shape_mn: CTA tile shape (tile_M, tile_N).
        C: Residual matrix of shape (M, N).
        S: Output for partial mean-of-squares of shape (M, num_blocks) in fp32.
        W: RMSNorm weight of shape (N,), broadcast across M.
        O: Output of shape (M, N) for D * W.

    Returns:
        Tuple of (epi_cls, epi_args, epi_outs, epi_keys).
    """
    M, N, K, L = shape_mnkl

    epi_dtype = torch2cute_dtype_map[C.dtype]
    post_act_dtype = torch2cute_dtype_map[O.dtype]

    epi_cls = lambda acc_dtype, tile_shape_mnk, buffer_align_bytes: EVTList([
        EVTResidual(
            acc_dtype=acc_dtype,
            epi_dtype=epi_dtype,
            tile_shape_mnk=tile_shape_mnk,
            buffer_align_bytes=buffer_align_bytes,
        ),
        EVTColBlockReductionStore(
            reduction_op=_create_mean_sq_reduction_op(
                element_type=acc_dtype,
                inv_block_size=1.0 / tile_shape_mnk[1],
            ),
            tile_shape_mnk=tile_shape_mnk,
        ),
        EVTRowVecMulPostAct(
            acc_dtype=acc_dtype,
            post_act_dtype=post_act_dtype,
            tile_shape_mnk=tile_shape_mnk,
            buffer_align_bytes=buffer_align_bytes,
        ),
    ])

    epi_args = EVTList.EpilogueArguments([
        EVTResidual.EpilogueArguments(
            mMatrix=C,
        ),
        EVTColBlockReductionStore.EpilogueArguments(
            mColVec=S,
        ),
        EVTRowVecMulPostAct.EpilogueArguments(
            mPostAct=O,
            mRowVec=W,
        ),
    ])

    epi_keys = (
        C.dtype,
        S.dtype,
        W.dtype,
        O.dtype,
        EVTResidual,
        EVTColBlockReductionStore,
        EVTRowVecMulPostAct,
    )

    epi_outs = {}

```

```
    return epi_cls, epi_args, epi_outs, epi_keys
```

Listing 2: Kernel Example.

C Experiments

C.1 List of Kernels

We summarize the kernels implemented in CODA. Each kernel is a GEMM followed by an epilogue program.

C.1.1 Basic Epilogue Kernels

We first list three basic GEMM-plus-epilogue kernels. These are useful for isolating individual epilogue primitives, although they are not always used directly in the Transformer forward pass.

Kernel 1: GEMM with RoPE. This kernel applies RoPE [15] to pairs of adjacent features in the GEMM output:

$$\begin{aligned} \boldsymbol{D} &= \boldsymbol{A}\boldsymbol{B}, \\ \boldsymbol{O} &= \mathrm{RoPE}(\boldsymbol{D}). \end{aligned}$$

Kernel 2: GEMM with SwiGLU. This kernel applies a fused SwiGLU activation to an interleaved GEMM output:

$$\begin{aligned} \boldsymbol{D} &= \boldsymbol{A}\boldsymbol{B}, \\ [\boldsymbol{G}, \boldsymbol{U}] &= \mathrm{interleavedSplit}(\boldsymbol{D}), \\ \boldsymbol{O} &= \mathrm{silu}(\boldsymbol{G}) \odot \boldsymbol{U}. \end{aligned}$$

Kernel 3: GEMM with partial cross-entropy. This kernel computes logits, selects the target logit, and emits block-wise log-sum-exp statistics for the cross-entropy loss:

$$\begin{aligned} \boldsymbol{Z} &= \boldsymbol{A}\boldsymbol{B}, \\ \boldsymbol{z}_{\mathrm{tgt}} &= \boldsymbol{Z}[\boldsymbol{y}], \\ \widehat{\boldsymbol{l}}_{\mathrm{lse}} &= \mathrm{reduceTile}_{\log\sum\exp}(\boldsymbol{Z}). \end{aligned}$$

C.1.2 Forward-Pass Kernels

The following kernels implement the reparameterized Transformer forward pass. They compose the basic epilogue primitives with RMSNorm scaling, residual updates, and partial reductions.

Kernel 4: GEMM with residual, partial RMSNorm, and weight scaling. This kernel implements the first stage of the GEMM–Residual–RMSNorm–GEMM pattern. It forms the residual-updated activation, emits partial RMS statistics, and applies the RMSNorm weight:

$$\begin{aligned} \boldsymbol{D} &= \boldsymbol{A}\boldsymbol{B} + \boldsymbol{C}, \\ \widehat{\boldsymbol{r}} &= \mathrm{reduceTile}_{\mathrm{cols}}(\boldsymbol{D} \odot \boldsymbol{D}), \\ \boldsymbol{O} &= \boldsymbol{D} \odot \boldsymbol{\gamma}. \end{aligned}$$

Kernel 5: GEMM with RMSNorm scaling. This kernel consumes a precomputed row-wise normalization factor and applies it in the GEMM epilogue:

$$\begin{aligned} \boldsymbol{D} &= \boldsymbol{A}\boldsymbol{B}, \\ \boldsymbol{O} &= \boldsymbol{D} \odot \boldsymbol{r}. \end{aligned}$$

Kernel 6: GEMM with RMSNorm and SwiGLU. This kernel composes row-wise RMSNorm scaling with SwiGLU, corresponding to the MLP gate/up projection:

$$\begin{aligned} \boldsymbol{D} &= \boldsymbol{A}\boldsymbol{B}, \\ \boldsymbol{D}' &= \boldsymbol{D} \odot \boldsymbol{r}, \\ [\boldsymbol{G}, \boldsymbol{U}] &= \mathrm{interleavedSplit}(\boldsymbol{D}'), \\ \boldsymbol{O} &= \mathrm{silu}(\boldsymbol{G}) \odot \boldsymbol{U}. \end{aligned}$$

Kernel 7: GEMM with RMSNorm and RoPE. This kernel composes row-wise RMSNorm scaling with RoPE, corresponding to QKV projection followed by rotary positional embedding:

$$\begin{aligned}\boldsymbol{D} &= \boldsymbol{A}\boldsymbol{B},\\ \boldsymbol{D}' &= \boldsymbol{D} \odot \boldsymbol{r},\\ \boldsymbol{O} &= \mathrm{RoPE}(\boldsymbol{D}').\end{aligned}$$

Kernel 8: GEMM with RMSNorm and partial cross-entropy. This kernel adds row-wise RMSNorm scaling before target-logit selection and partial log-sum-exp reduction, corresponding to the language-modeling head:

$$\begin{aligned}\boldsymbol{Z} &= (\boldsymbol{A}\boldsymbol{B}) \odot \boldsymbol{r},\\ \boldsymbol{z}_{\mathrm{tgt}} &= \boldsymbol{Z}[\boldsymbol{y}],\\ \widehat{\boldsymbol{l}}_{\mathrm{lse}} &= \mathrm{reduceTile}_{\log\sum\exp}(\boldsymbol{Z}).\end{aligned}$$

C.1.3 Backward-Pass Kernels

Finally, we list the backward kernels. These kernels mirror the forward structure: each performs a GEMM, applies the local backward rule in the epilogue, and emits partial reductions needed by neighboring RMSNorm backward computations.

Kernel 9: GEMM with residual and RMSNorm backward. This kernel implements the local part of RMSNorm backward. Let $\boldsymbol{C}$ denote the RMSNorm input, $\boldsymbol{r}$ the row-wise inverse RMS factor, $\boldsymbol{\gamma}$ the RMSNorm weight, and $\boldsymbol{z}_{\Delta z}$ the row-wise RMSNorm backward statistic:

$$\begin{aligned}\boldsymbol{D} &= \boldsymbol{A}\boldsymbol{B}^{\top},\\ \boldsymbol{C}_{\mathrm{norm}} &= \boldsymbol{C} \odot \boldsymbol{r},\\ \boldsymbol{O}_{\mathrm{out}} &= \boldsymbol{O}_{\mathrm{in}} + (\boldsymbol{D} \odot \boldsymbol{\gamma} - \boldsymbol{C}_{\mathrm{norm}} \odot \boldsymbol{z}_{\Delta z}) \odot \boldsymbol{r},\\ \boldsymbol{C}_{\mathrm{out}} &= \boldsymbol{C}_{\mathrm{norm}} \odot \boldsymbol{\gamma},\\ \widehat{\nabla_{\boldsymbol{\gamma}}\mathcal{L}} &= \mathrm{reduceTile}_{\mathrm{rows}}\left(\boldsymbol{D} \odot \boldsymbol{C}_{\mathrm{norm}}\right).\end{aligned}$$

Kernel 10: GEMM with SwiGLU backward. This kernel computes the backward pass of a fused SwiGLU epilogue and emits the row-wise statistic needed by the preceding RMSNorm backward. Let $\boldsymbol{Z}$ be the saved interleaved pre-activation tensor:

$$\begin{aligned}\boldsymbol{D} &= \boldsymbol{A}\boldsymbol{B}^{\top},\\ [\boldsymbol{G}, \boldsymbol{U}] &= \mathrm{interleavedSplit}(\boldsymbol{Z}),\\ \boldsymbol{O} &= \mathrm{silu}(\boldsymbol{G}) \odot \boldsymbol{U},\\ \nabla_{\boldsymbol{U}}\mathcal{L} &= \boldsymbol{D} \odot \mathrm{silu}(\boldsymbol{G}),\\ \nabla_{\boldsymbol{G}}\mathcal{L} &= \boldsymbol{D} \odot \boldsymbol{U} \odot (\sigma(\boldsymbol{G}) + \mathrm{silu}(\boldsymbol{G}) \odot (1 - \sigma(\boldsymbol{G}))),\\ \nabla_{\boldsymbol{Z}}\mathcal{L} &= \mathrm{interleavedConcat}\left(\nabla_{\boldsymbol{G}}\mathcal{L}, \nabla_{\boldsymbol{U}}\mathcal{L}\right),\\ \widehat{\boldsymbol{z}_{\Delta z}} &= \mathrm{reduceTile}_{\mathrm{cols}}\left(\boldsymbol{G} \odot \nabla_{\boldsymbol{G}}\mathcal{L} + \boldsymbol{U} \odot \nabla_{\boldsymbol{U}}\mathcal{L}\right).\end{aligned}$$

C.2 Setup Details

Experiments are conducted using a single H100 GPU. We use the following package versions.

1. PyTorch 2.10.0
2. CuTeDSL 4.4.2
3. Liger Kernels 0.8.0
4. FlashInfer 0.6.10.post1
5. QuACK Kernels 0.4.1

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: See abstract and introduction.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: See the section on conclusion and limitations

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

Justification: NA

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [N/A]

   Justification: Code will be open-sourced

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [N/A]

Justification: Will be open-sourced

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [N/A]

   Justification: No model training experiments.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: See experiments

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.

- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: See appendix.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics https://neurips.cc/public/EthicsGuidelines?

   Answer: [Yes]

   Justification: NA

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    Justification: This work discusses GPU kernels

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.

- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: This work discusses GPU kernels

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [No]

    Justification: We will open-source.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.

- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [N/A]

    Justification: We will open-source.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: NA

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: NA

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: See methods section

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.