Unweight: Lossless MLP Weight Compression for LLM Inference

IVAN NIKULIN, Cloudflare, Inc.,

Unweight is a research program on lossless compression of LLM weight tensors, with active work on dense inference, model distribution, and Mixture-of-Experts serving. This report presents intermediate results from the engineering stream: a composable GPU toolkit whose components can be assembled for these deployment scenarios on NVIDIA Hopper GPUs (H100, H200). It is well established in prior work (DFloat11, ZipServ, ZipNN) that BF16 exponent fields in trained LLM weights carry ~2.6 bits of Shannon entropy in their 8-bit allocation, while sign and mantissa fields are near-incompressible. While this report focuses on BF16 MLP weights, the research program also explores compression of other weight formats.

Unweight separates each BF16 value into sign+mantissa and exponent, Huffman-codes the exponents over a per-tensor 16-value palette, and handles rare exponents through verbatim rows rather than inline escape symbols. The compressed representation, execution pipelines, and runtime scheduling are independently configurable: a model can be Huffman-encoded for distribution and transcoded to a palette intermediate representation on load for inference.

Three execution pipelines—full decode to cuBLAS, exponent decode with reconstructive matmul, and palette transcode with reconstructive matmul—are selected per projection and batch-size bucket via coordinate-descent autotuning on end-to-end throughput.

The central compute primitive is a persistent ThunderKittens LCF kernel that reconstructs BF16 tiles in shared memory immediately before Hopper WGMMA consumption, eliminating a full HBM round-trip for the weight matrix. A hard/easy layer alternation schedule extends preprocess-compute overlap across layers with different encoding profiles.

On Llama-3.1-8B, Unweight achieves ~30% compression on MLP weights (~20% total model size reduction) with lossless numerical equivalence. Research is ongoing; this is a technical report, not a final pre-print.

Author's Contact Information: Ivan Nikulin, Cloudflare, Inc.,, inikulin@cloudflare.com.

Cloudflare Technical Report Cf-TR-2026.04.v1,

Contents

Kernel source code: https://github.com/cloudflareresearch/unweight-kernels

1 Introduction

1.1 The Problem

Dense BF16 inference on datacenter GPUs is memory-bandwidth bound during autoregressive decode: each generated token reads the full model weight matrix from HBM. For a Llama-3 class model [9], MLP projections (gate, up, down) account for roughly two-thirds of the parameter footprint and dominate per-token HBM traffic. Reducing this traffic directly improves decode throughput.

The same weight traffic problem manifests in two other settings. First, model **distribution**: shipping multi-gigabyte checkpoints over networks is slow, and any lossless compression that can be applied and removed at GPU speed reduces transfer time without numerical compromise. Second, **Mixture-of-Experts** models: frontier MoE architectures are enormous and require GPU clusters; cold experts that are not resident in HBM must be fetched on demand, and compressed storage reduces the fetch cost.

Why BF16. Quantized formats (FP8, INT4) reduce memory footprint through lossy precision reduction but sacrifice numerical fidelity. Full-precision BF16 remains the standard for workloads where output quality cannot be traded for throughput — notably agentic applications (multi-step tool calling, code generation, complex reasoning chains) where errors compound across steps, and fine-tuned domain-specific models where quantization degrades task-specific performance. These use cases drive sustained demand for dense BF16 inference at scale. Lossless compression of lower-precision formats is also viable: recent work (Hershcovitch et al. [13]) demonstrates that FP8 E4M3 weights are compressible via the same exponent-separation technique, with the 4-bit exponent field showing strong concentration (compression ratios of 0.20–0.30 on the exponent stream). The overall savings are more modest than BF16 because FP8's 8-bit total leaves less absolute redundancy to exploit. Extending Unweight to FP8 and other formats is ongoing research; this report focuses on BF16 as the primary target.

1.2 Why Lossless Compression on Hopper is Hard

Lossless inference-time weight compression on datacenter GPUs is fundamentally harder than the consumer-GPU or offline settings targeted by most prior work. NVIDIA Hopper GPUs (H100, H200) [15] are throughput machines: their tensor cores and high-bandwidth HBM (3.35 TB/s on H100, 4.8 TB/s on H200) are designed to be simultaneously saturated, leaving almost no free compute cycles or memory bandwidth to hide decompression work. Any decode latency that is not perfectly overlapped with the matmul critical path becomes directly additive to token latency. This is unlike consumer GPUs (where compute is the bottleneck and spare memory bandwidth is available for decompression) or offline decompression (where latency is irrelevant). The challenge is making compression pay for itself on hardware that is already well-balanced between compute and memory.

Unweight targets Hopper specifically because it is the dominant deployed inference architecture in datacenter fleets as of 2026, and will remain so for years — the capital investment cycle for GPU infrastructure spans multiple hardware generations. NVIDIA's newer Blackwell architecture (B200, B300) has been shipping since late 2025 but remains supply-constrained, and most cloud inference capacity runs on Hopper. The kernel designs in this report (WGMMA, TMA, asymmetric warpgroup scheduling, 228 KB SMEM per SM) [10, 11] are Hopper-specific; adapting them to Blackwell's different SM architecture is future work. Importantly, H100 and H200 share identical

compute architecture — same 132 SMs, same SMEM capacity, same ISA — differing only in HBM capacity and bandwidth. All Unweight kernels run unmodified on both.

1.3 Unweight: A Composable Compression Toolkit

Unweight addresses these problems through a composable set of GPU-accelerated encoding, decoding, transcoding, and fused-matmul kernels. The key architectural property is that **encoding, decoding, and execution decisions are independently configurable**:

- A whole model can be Huffman-encoded for **distribution** purposes, maximizing compression ratio without any inference-engine coupling.
- For **dense inference**, projections can be individually configured as raw, palette-encoded, or Huffman-encoded. Infire [12], Cloudflare's proprietary LLM inference engine, selects among five execution paths per projection per batch-size bucket. Huffman-encoded projections can be transcoded to palette form on model load, enabling the most efficient runtime scheduling without constraining the distribution format.
- For **MoE cold experts**, compressed MLP weights reduce the per-expert memory footprint and transfer cost when experts are loaded on demand.

The components—encoder, decoder, palette transcoder, reconstructive matmul, runtime scheduler, and autotuner—are designed to be assembled per deployment scenario rather than used as a monolithic pipeline.

1.4 Scope of This Report

This report presents the **engineering stream** of the Unweight research program: kernel design, system architecture, and runtime integration. A separate research stream investigates new compression methodologies for LLM weights and is not covered here.

The core technical contributions are:

(1) **Verbatim rows**: outlier handling is lifted from per-symbol escape codes to row-level metadata, removing branches from the decode and matmul hot paths.
(2) **Branchless Huffman decode**: the coded alphabet is restricted to a 16-value palette, making every decode-table entry terminal and enabling a single-lookup branchless bitreader.
(3) **Reconstructive matmul**: a persistent ThunderKittens [8] LCF kernel that reconstructs BF16 from compressed inputs in shared memory and issues WGMMA without an HBM round-trip.
(4) **Hard/easy layer alternation**: a runtime scheduling strategy that prefetches Huffman preprocessing across intervening palette/raw layers to maximize compute-decode overlap.
(5) **Composable configuration**: encoding format, execution pipeline, and scheduling are orthogonal choices connected by runtime transcoding.
(6) **Coordinate-descent autotuning**: per-projection, per-batch-bucket pipeline selection driven by end-to-end throughput measurement.

On Llama-3.1-8B, these yield ~30% compression on MLP weights, ~20% total model VRAM reduction, and a production runtime that adapts its decode/compute tradeoff to the current batch size.

2 Prior Work and Differentiation

2.1 Landscape of Lossless BF16 Weight Compression

All systems in this space share the same foundational observation: BF16 exponent fields in trained LLM weights have Shannon entropy of ~2.6 bits versus their 8-bit allocation, while sign and mantissa fields are near-maximal entropy and essentially incompressible. The systems diverge

in how they encode this redundancy and—critically—how they integrate decoding into the GPU inference pipeline.

Deep Compression (Han et al. [1]). The earliest work applying Huffman coding to neural network weights. Applied to quantized CNNs, achieving ~22% additional compression on top of pruning and quantization. No GPU inference support; decoding was CPU-only. Established the principle but predated the memory-bandwidth-bound regime of modern LLM inference.

DietGPU (Johnson [2]). A GPU-native implementation of range ANS (rANS) for lossless floating-point compression. General-purpose (not LLM-specific), targeting HPC/ML communication and checkpointing. Achieves only ~43.7% of peak memory bandwidth on L40S during decompression (per ZipServ's evaluation), because ANS's state-machine decoding introduces heavy thread divergence and serialized bit parsing on SIMT architectures.

ZipNN (Hershcovitch et al. [3]). A lossless compression library optimized for neural network tensors. Uses Huffman coding with byte-level separation of exponents. Targets **storage and distribution only**—compression/decompression happens on the host for model download and checkpointing. No GPU inference integration or runtime decompression.

NeuZip (Hao et al. [4]). Compresses BF16 exponents using ANS, decompressing layer-by-layer during inference. Uniquely targets both training (lossless) and inference (lossy mantissa truncation). Relies on NVIDIA's nvCOMP library for GPU ANS, which is closed-source and binary-only, limiting portability and optimization. Higher latency and lower throughput than DFloat11 due to nvCOMP's general-purpose ANS implementation not being tuned for BF16 exponent distributions.

DFloat11 (Zhang et al. [5]). The closest prior work to Unweight. Key design choices:

- *Encoding*: Per-model Huffman coding of BF16 exponents. Sign+mantissa stored uncompressed. ~30% total compression (~70% size), matching Unweight's compression ratio.
- *Decode table*: Hierarchical cascaded 8-bit LUTs ($k = 4$–8 sub-tables, ~2.3 KB total). Each 256-entry sub-table decodes 8 bits; misses redirect to a child sub-table via repurposed unused exponent values (240–255) as internal pointers. This keeps SMEM usage low but introduces dependent memory-latency divergence on multi-level misses.
- *Thread model*: Two-phase kernel. Phase 1: each thread decodes its 8-byte chunk and counts output elements (no writes). Block-wide prefix sum via Blelloch scan computes per-thread output offsets. Phase 2: re-decode the same chunk and write to a SMEM staging buffer, then batch-flush to HBM via coalesced writes.
- *Auxiliary metadata*: 5-bit gap array (per-thread starting bit offset) + block-level output position array. These are precomputed during encoding and stored alongside the compressed bitstream.
- *Pipeline*: Fully decoupled—decompress entire weight matrices to HBM in BF16 format, then run standard cuBLAS GEMM. Decompression is batched across all matrices in a transformer block.
- *Integration*: HuggingFace Transformers framework. Targets consumer GPUs (RTX 3090/4090) and CPU-offloading scenarios where the model exceeds single-GPU HBM.

Huff-LLM (Yubeaton et al. [6]). Applies Huffman coding to both exponents and mantissa fields. Designed for FPGA-based inference accelerators (systolic arrays, vector accelerators) with custom hardware decoders. Not applicable to GPUs—requires dedicated Huffman decode hardware that doesn't exist on NVIDIA SMs.

ZipServ (Fan et al. [7]). The most architecturally distinct approach, and the only prior work to fuse decompression with GEMM on GPUs. Key innovations:

- *Encoding*: Abandons variable-length entropy coding entirely. Exploits the observation that in 99.6% of LLM weight matrices, the top-7 most frequent exponents form a **numerically contiguous** range. Encodes each element as a fixed-length 3-bit codeword (001–111 for in-range, 000 for outlier), stored as three 64-bit bitmaps per 8×8 tile (Triple Bitmap Encoding, TCA-TBE).
- *Decode*: Completely branch-free. Each thread reads 2 bits from the spatial indicator bitmap, loads from either the compressed (sign+mantissa) or fallback (full BF16) buffer using `__popc()`-based dynamic addressing, and reconstructs the exponent via `base_exp + codeword` (single integer add, no LUT).
- *Fused ZipGEMM*: Loads compressed tiles from HBM to SMEM, decompresses in registers, and feeds directly to Tensor Core `mma.sync` instructions. Eliminates the HBM round-trip for decompressed weights entirely.
- *Pipeline*: Stage-aware—uses fused ZipGEMM for memory-bound decode phase, decoupled decompression + cuBLAS for compute-bound prefill.
- *Results*: 1.31× average speedup over cuBLAS on RTX 4090, 1.36× on L40S. However, admits limitations on datacenter GPUs (A100, H800) where abundant HBM bandwidth reduces the benefit, and the intense ALU workload from decompression is harder to hide at lower clock frequencies.

2.2 How Unweight Differs

Unweight occupies a unique position in this landscape across several dimensions:

1. Selective MLP-only compression. All prior work compresses the entire model (all linear layers, embeddings, LM head). Unweight deliberately compresses only MLP projections (gate, up, down), leaving attention weights, embeddings, and layer norms uncompressed. This is a production-oriented choice: MLP weights are 2/3 of Llama-3's parameter count and dominate decode-phase HBM traffic. Attention weights have different access patterns (K/V cached, only Q/O are per-token) and compress slightly less well. Selective compression avoids decode overhead on weights where the benefit is marginal.

2. Three-pipeline architecture with per-projection adaptive dispatch. No prior work offers multiple execution paths. DFloat11 has one path (decode → cuBLAS). ZipServ has two (fused ZipGEMM for decode, decoupled for prefill). Unweight has three (full decode → cuBLAS, exp-decode → reconstructive matmul, palette transcode → reconstructive matmul), each independently selectable per-projection per-batch-size-bucket via coordinate descent tuning. This captures non-monotonic performance crossovers that single-path systems miss—e.g., full decode may win at batch=1 where cuBLAS launch overhead dominates, while palette + reconstructive matmul wins at batch=32 where HBM traffic dominates.

3. Reconstructive matmul with WGMMA on Hopper. Unweight's reconstructive matmul is conceptually similar to ZipServ's ZipGEMM (both fuse decompression with GEMM), but differs architecturally (Table 1):

Dimension	ZipServ ZipGEMM	Unweight Reconstructive Matmul
Tensor Core ISA	`mma.sync` (Ampere/Ada)	WGMMA (Hopper)
Grid model	Standard split-K tiling	Persistent LCF (ThunderKittens [8])
Reconstruction location	Register file	Shared memory (SMEM)
Producer/consumer	Single-phase (all warps decode + compute)	Asymmetric: 1 producer WG + 2 consumer WGs
Register allocation	Uniform across warps	Asymmetric: producer 40, consumer 232 regs
Pipeline stages	2-level double buffer	Configurable 2–4 input pipe stages
Target hardware	Consumer GPUs (RTX 4090, L40S)	Datacenter H100/H200 (132 SMs, 228 KB SMEM/SM)

Table 1. Architectural comparison: ZipServ ZipGEMM vs. Unweight reconstructive matmul.

ZipServ [7] decompresses in registers and feeds `mma.sync` directly—possible because TCA-TBE's fixed-length bitmap encoding produces deterministic per-thread data without thread divergence. Unweight cannot do this because Huffman decoding is inherently sequential per-thread (variable-length symbols), so it decouples decode from matmul and reconstructs in SMEM where the data can be swizzled for conflict-free WGMMA access.

4. Palette pipeline: 4-bit intermediate representation. Unique to Unweight. The 16-entry high-frequency exponent palette with 4-bit packed indices reduces the transcoder's HBM write traffic by 50% compared to writing full u8 exponents, and reduces the matmul's HBM read traffic correspondingly. Rows whose exponents all belong to the palette are transcoded to packed nibbles; rows containing any non-palette exponent are stored verbatim and bypassed during transcoding. The palette resolve in the matmul uses branchless `__byte_perm`-based lookup—entirely in ALU, no SMEM table. No prior work has an analogous intermediate representation between the entropy-coded bitstream and the matmul.

5. Huffman with flat table vs. cascaded LUTs. DFloat11 [5] uses cascaded 8-bit sub-tables to minimize SMEM usage (~2.3 KB). Unweight uses a single flat decode table sized dynamically to the tensor's maximum code length (up to 32768 entries for 15-bit codes). On Hopper with 228 KB SMEM, even the largest table fits comfortably—there is no pressure to minimize it. Because the coded alphabet is restricted to palette symbols and non-palette rows are handled as verbatim rows, every decode-table entry is terminal: one lookup yields the symbol, code length, and palette index with no dependent second-stage lookup. This avoids the 60+ cycle dependent-load penalty of cascaded misses.

6. Work-stealing vs. two-phase decode. DFloat11's [5] two-phase kernel decodes each chunk twice (once to count, once to write) with an intervening Blelloch prefix sum for output placement. This requires precomputed gap arrays and block-level position arrays as auxiliary metadata. Unweight's decoder is single-pass: each thread atomically claims a row via `atomicAdd`, decodes it, and writes directly to the output. No auxiliary metadata beyond per-tile bit offsets (computed during encoding). The atomic work-stealing naturally load-balances across tiles with varying compression ratios.

7. Compression ratio: comparable but not identical. DFloat11 [5] and ZipServ [7] both report ~30% compression (~70% size) across entire models. Unweight achieves the same ~30% compression ratio on MLP weights specifically, translating to ~20% total model size reduction (since only MLP weights are compressed). This is a deliberate design tradeoff: less total compression, but zero overhead on non-MLP layers.

2.3 Summary Comparison Table

	DFloat11	ZipServ	NeuZip	DietGPU	Unweight
Workflow	Decode→cuBLAS	Fused GEMM	Layer-wise decode	Transfer/storage	Both
Encoding	Huffman (cascade)	Fixed 3-bit bitmap	ANS (nvCOMP)	rANS	Huffman (flat)
Decode table	8-bit cascade	None (arith.)	ANS state	ANS state	12-bit flat LUT
Fused GEMM	No	Yes (ZipGEMM)	No	No	Yes (recon. matmul)
TC ISA	N/A	mma.sync	N/A	N/A	WGMMA
Scope	All layers	All layers	All layers	All tensors	MLP only
Palette IR	No	No	No	No	Yes (4-bit)
Adaptive tuning	No	No	No	No	Yes (coord. desc.)
Target GPU	Consumer (4090)	Consumer (4090)	Any (nvCOMP)	Any	Hopper (H100/H200)
Framework	HF Transformers	vLLM	PyTorch	Library	Infire [12]

Table 2. Summary comparison of lossless BF16 weight compression systems.

Not shown in the table: **ZipNN** (Hershcovitch et al. [3]) targets storage and distribution only — host-side compression/decompression with no GPU inference integration. **Deep Compression** (Han et al. [1]) was CPU-only offline compression for quantized CNNs. **Huff-LLM** (Yubeaton et al. [6]) targets FPGA-based inference with custom hardware decoders.

3 Compressed Representation

3.1 BF16 Decomposition

Each BF16 value b is split into two bytes:

- $\sigma(b) \in \mathsf{u8}$: the sign+mantissa byte (1 sign bit + 7 mantissa bits)
- $\varepsilon(b) \in \mathsf{u8}$: the exponent byte (8 bits)

Sign+mantissa fields are near-maximal entropy and stored verbatim. Exponent fields exhibit strong clustering (~2.6 bits of Shannon entropy) and are the compression target (Figure 1).

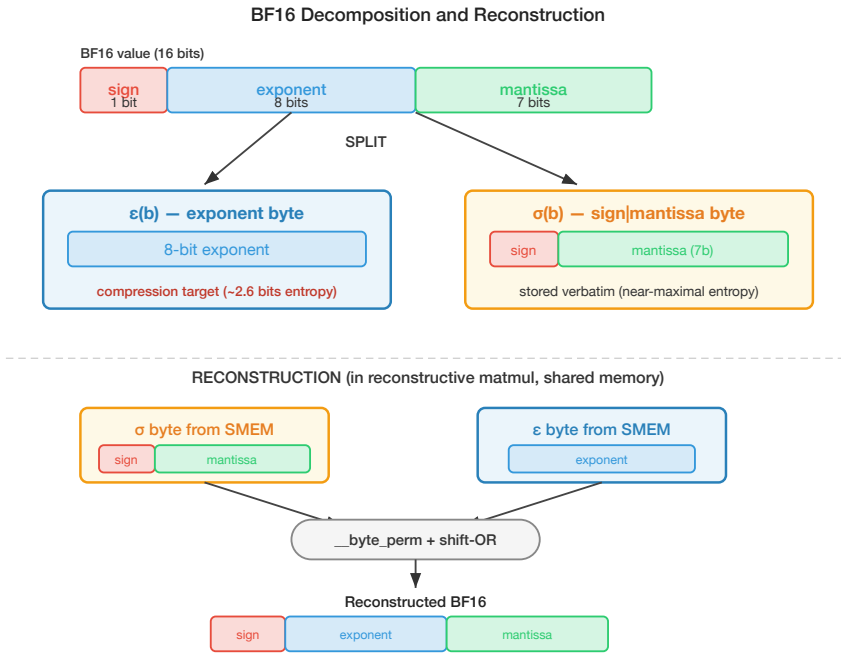

Fig. 1. BF16 decomposition and reconstruction.

3.2 Tiled Layout

The weight matrix $B \in \mathrm{BF16}^{N \times K}$ (with N, K divisible by 64) is partitioned into 64×64 tiles $B_{u,v}$. Tiles are stored in row-major order `[N/64, K/64, 64, 64]`, aligning with cuBLAS B-operand expectations and requiring no transpose for matmul consumption.

3.3 Palette and Verbatim-Row Model

For each tensor, the encoder selects a **palette** of the 16 most frequent exponent values:

$$P = \{p_0, p_1, \ldots, p_{m-1}\}, \quad m \leq 16$$

A tile row r is classified as **verbatim** if any of its 64 exponents falls outside the palette:

$$q(u, v, r) = 1 \iff \exists\, c \text{ such that } \varepsilon(B_{u,v}[r, c]) \notin P$$

This produces two row classes:

- **Coded row** ($q = 0$): all exponents are palette members; the row is Huffman-coded into the bitstream.
- **Verbatim row** ($q = 1$): the row is removed from the Huffman stream and its 64 raw exponent bytes are copied to a dense `verbatim_exponents` tensor.

This formulation lifts outlier handling from per-symbol control flow to row-level metadata, eliminating branches from the decode and matmul hot paths while preserving exact reconstruction (Figure 2).

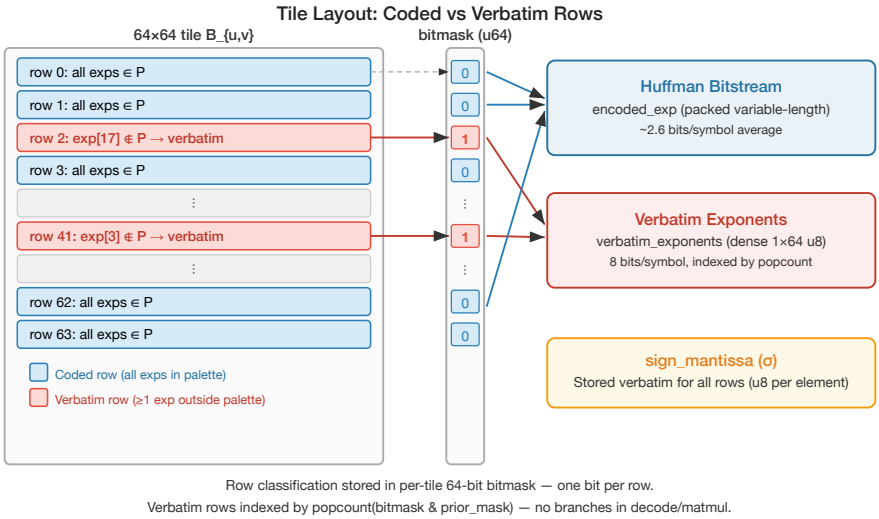

Fig. 2. Tile layout: coded vs verbatim rows.

3.4 Huffman Coding Over Palette Symbols

A single Huffman code [14] is built per tensor, but the coded alphabet contains **only** palette symbols. Let ℓ_i be the code length assigned to palette symbol p_i, and define $L = \max_i \ell_i$. The decoder constructs a flat table:

$$T \in \text{Entry}^{2^L}, \quad \text{Entry} = (\text{symbol}, \text{length}, \text{palette\_idx})$$

The table is sized dynamically to 2^L where L is the tensor's maximum code length (the runtime supports L up to 15, i.e. up to $2^{15} = 32768$ entries). Every entry is terminal: one lookup yields the decoded exponent, its code length, and its palette index. There is no escape marker and no dependent second-stage lookup, because verbatim rows are routed out before bit parsing begins.

The branchless bitreader hot path uses `__funnelshift_r` to form a sliding 64-bit window from two adjacent 32-bit words, making word-boundary crossings free of branches:

```
u32 bits = __funnelshift_r(current_word, next_word, bit_idx);
HuffmanDecodeEntry entry = decode_table[bits & (DECODE_TABLE_SIZE - 1)];
consume_bits(entry.length);
```

3.5 Per-Tensor Storage Layout

The complete compressed representation of one tensor consists of:

```
sign_mantissa      : u8 [N, K]              sigma bytes, stored verbatim
encoded_exp        : packed Huffman bits     coded rows only
row_end_offsets    : u16 [num_tiles, 64]     cumulative coded bits per row
tile_metadata      : TileMetadata [num_tiles] per-tile routing info
verbatim_exponents : u8 [verb_rows, 64]      dense verbatim row store
high_freq_palette  : u8 [m]                  palette values, m <= 16
decode_table       : HuffmanDecodeEntry [2^L] flat decode LUT
```

where each `TileMetadata` record contains:

```
TileMetadata(t) = (
  tile_start_bit      : u32,  // first coded bit for this tile
  verbatim_row_offset : u32,  // index into verbatim_exponents
  verbatim_bitmask    : u64   // which of the 64 rows are verbatim
)
```

Two invariants hold by construction:

- `encoded_exp` contains only palette symbols; non-palette exponents never enter the bitstream.
- `row_end_offsets` remains contiguous even in the presence of verbatim rows, because a verbatim row contributes zero coded bits.

3.6 Palette Intermediate Representation

For the palette-transcoded execution pipeline (§6), coded rows are further transformed into packed 4-bit palette indices (two indices per byte), halving the transient exponent-side HBM traffic. The encoder computes a reverse mapping `symbol_to_palette_idx[256]` so that each palette exponent maps to a nibble `0..15`. Verbatim rows bypass this transform entirely and are loaded directly from `verbatim_exponents` by the consumer kernel.

4 Encoding Pipeline

Encoding runs offline on GPU. The algorithmic flow is summarized below; the subsequent subsections describe each stage.

Algorithm 1 Offline encoding of one BF16 tensor B

Require: $B \in \text{BF16}^{N \times K}$
Ensure: sign_mantissa, encoded_exp, row_end_offsets, tile_metadata, verbatim_exponents, high_freq_palette, decode_table

Extract $\sigma(b)$ for every element → sign_mantissa. Accumulate a tensor-global 256-bin histogram of $\varepsilon(b)$.
Sort histogram descending. Select top-m exponents ($m \leq 16$) as palette P. Build Huffman tree over P. Extract encode_table and decode_table.
for each tile row r **do**
    if $\exists\, c$ such that encode_table$[\varepsilon(r, c)]$.length = 0 **then**
        Mark r as verbatim via atomicOr on verbatim_bitmask. coded_length$(r) \leftarrow 0$.
    else
        coded_length$(r) \leftarrow \sum_c$ encode_table$[\varepsilon(r, c)]$.length
    end if
    Warp-level inclusive prefix sum → row_end_offsets per tile.
    Cross-warp fixup and TileMetadata assembly.
end for
Global prefix sum of tile bit lengths → tile_start_bit offsets.
for each non-verbatim row **do** Huffman-encode into encoded_exp.
end for
for each verbatim row **do** copy 64 raw exponent bytes into verbatim_exponents.
end for

The pipeline is implemented as six GPU kernel launches (Figure 3):

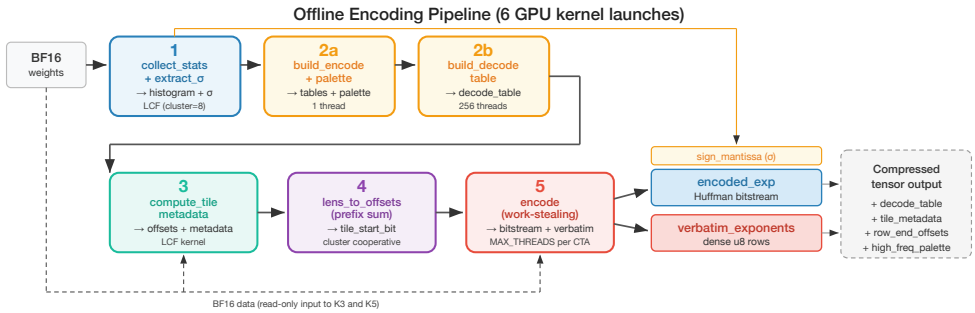

Fig. 3. Offline encoding pipeline.

4.1 Statistics Collection and Sign+Mantissa Extraction

A ThunderKittens [8] LCSF (Load-Compute-Store-Finish) kernel with `cluster_size = 8`. Each Cooperative Thread Array (CTA) streams 64×64 BF16 tiles via TMA. Two consumer warps (64 threads = one thread per tile row) simultaneously extract σ bytes and accumulate a 256-bin exponent histogram in shared memory. The σ extraction uses `__byte_perm` to separate the sign+mantissa byte from each BF16 pair in three instructions:

```
mantissas      = __byte_perm(bf16_pair, 0, 0x0020);
signs          = __byte_perm(bf16_pair, 0, 0x0031);
sign_mantissa  = (mantissas & 0x7F7F) | (signs & 0x8080);
```

After all tiles are processed, the finish stage reduces per-CTA histograms across the 8-block cluster using distributed shared memory (`cluster.map_shared_rank`) and writes the accumulated totals to the global histogram.

4.2 Table Construction

Table construction is split into two kernels. The first (`build_encode_table_and_palette`) runs on a **single thread** and performs:

(1) Sort the 256-bin histogram descending by frequency.
(2) Select the top-m symbols ($m \leq 16$) as the palette P. Build `high_freq_palette[`m`]` and the reverse mapping `symbol_to_palette_idx[256]`.
(3) Construct leaf nodes for palette symbols only (up to $16 \times 2 = 32$ tree nodes), ordered ascending by frequency.
(4) Build a Huffman tree via the classic two-queue merge.
(5) Extract `encode_table` by top-down DFS, assigning LSB-first codes. Non-palette symbols receive `length = 0`, which serves as an offline classification predicate: any row containing such a symbol is routed to the verbatim path.
(6) Output `decode_table_size` $= 2^L$ where L is the maximum code length.

The second kernel (`build_decode_table`) launches one thread per data symbol. Each thread reads its encode entry and, for symbols with non-zero code length, fills $2^{L-\ell}$ slots in the flat decode table (where L is the max code length and ℓ is the symbol's code length). Every resulting `HuffmanDecodeEntry` is terminal: `{symbol, length, palette_idx}`.

4.3 Tile Metadata Computation

An LCF kernel processing batches of 64 tiles per task. The CTA-level algorithm is:

Algorithm 2 Tile metadata computation — one CTA

Require: BF16 tiles via TMA, encode_table in shared memory
Ensure: row_end_offsets[tile, 64], TileMetadata per tile, global max_verbatim_per_tile

Producer: TMA load one 64×64 BF16 tile per iteration.
Consumer (64 threads = 1 per tile row):
for each of 64 elements in the row **do**
    if encode_table[ε(element)].length = 0 **then**
        atomicOr(verbatim_bitmask, $1 \ll$ row); row_bit_length $\leftarrow$ 0; **break**
    end if
    row_bit_length += encode_table[ε(element)].length
end for
row_end_offsets[row] $\leftarrow$ warp_inclusive_prefix_sum(row_bit_length)
Finish (warp 0 only):
    Cross-warp fixup: add warp 0's total to all of warp 1's offsets.
    tile_bit_length $\leftarrow$ warp0_total + warp1_local_total
    Assemble TileMetadata(tile_bit_length, popcount(bitmask), bitmask).
    atomicMax(max_verbatim_per_tile, popcount(bitmask))
    TMA store row_end_offsets and metadata.

The cross-warp fixup (step 12 in Algorithm 2) uses packed u32 arithmetic to update two u16 offsets per u32 simultaneously, avoiding scalar operations:

```
u32 total_packed = warp0_total | ((u32)warp0_total << 16);
val.x += total_packed; val.y += total_packed; // vectorized as u32x4
```

The global `max_verbatim_per_tile` (step 16 in Algorithm 2) is used at inference time to select the `MAX_VERBATIM_ROWS` dispatch bucket for the reconstructive matmul kernel (§8.4).

4.4 Tile Length to Offset Conversion

A cluster-cooperative prefix-sum kernel (cluster size 8, 1024 threads per CTA) that converts per-tile bit lengths into global bit offsets via a four-level hierarchical exclusive prefix sum: per-thread slice → warp → CTA → cluster. The cluster level uses distributed shared memory (`cluster.map_shared_rank`) to accumulate totals from lower-ranked blocks. The tile-length array is overwritten in place, avoiding an extra allocation.

4.5 Encoding

The encoding kernel uses work-stealing rather than the LCF template used by the earlier stages. The reason is that Huffman-coded row lengths vary significantly across tiles: a tile with many short codes finishes much faster than one with long codes. LCF assigns fixed-size batches to each CTA, which would leave some CTAs idle while others are still working. Work-stealing with per-row atomic dispatch naturally balances this variable workload.

Algorithm 3 Work-stealing Huffman encode

Require: encode_table in shared memory, tile_metadata, row_end_offsets, BF16 weights
Ensure: encoded_exp bitstream, verbatim_exponents

```
Load encode_table from HBM to shared memory (once per CTA).
loop
    row_idx ← atomicAdd(global_counter, 1)
    if row_idx ≥ total_rows then exit
    end if
    tile_idx ← row_idx ≫ 6; row_in_tile ← row_idx & 63
    Load TileMetadata(tile_idx).
    Extract 64 exponent bytes from BF16 row.
    if row is verbatim (from bitmask) then
        verbatim_index ← popcount(bitmask bits below row_in_tile)
        Copy 64 exponent bytes to verbatim_exponents[verbatim_row_offset + verbatim_index].
    else
        bit_offset ← tile_start_bit + row_start_bit
        for each of 64 exponents do
            Look up Huffman code from encode_table. Accumulate code bits into register word.
            On word boundary: flush to encoded_exp (first word uses atomicOr; mid-row words use direct store).
        end for
        Flush final partial word with atomicOr.
    end if
end loop
```

The first word of each row requires `atomicOr` because it may share a u32 word with the previous row's final bits. Subsequent mid-row words are written directly since each row owns exclusive bit ranges within those words.

5 Composable Execution Model

A key property of Unweight is that **encoding format, execution pipeline, and runtime scheduling are orthogonal choices**. This section describes the configuration space; subsequent sections describe the individual components.

5.1 Bundle-Time Encoding Choices

Each MLP projection (gate, up, down) is independently assigned one of three encoding modes at bundle time:

- **Raw**: uncompressed BF16 weights, no Unweight involvement.
- **Huffman**: full Huffman-coded representation (§3). Maximizes compression ratio and is the natural choice for distribution bundles.
- **Palette**: pre-transcoded representation where coded rows are already stored as packed 4-bit palette indices (§3.6). Eliminates the need for runtime Huffman decoding, at the cost of slightly larger on-disk size than Huffman.

These choices are per-projection and per-layer. A bundle can mix all three within a single model.

5.2 Inference-Time Execution Choices

At inference time, each projection is assigned an execution configuration per batch-size bucket. Five execution paths are available (Table 3):

Path	Encoding	Preprocess	Consumer
RawCublas	Raw	none	cuBLAS
PaletteReconstructive	Palette	none	recon. matmul
HuffmanFullDecodeCublas	Huffman	full decode to BF16	cuBLAS
HuffmanExpDecodeRecon.	Huffman	exp-only decode	recon. matmul
HuffmanPaletteTranscodeRecon.	Huffman	palette transcode	recon. matmul

Table 3. Five execution paths available per projection.

The three Huffman paths differ in how much preprocessing they perform before handing data to the consumer kernel. The autotuner (§10) selects among compatible paths by measuring end-to-end throughput.

5.3 Runtime Transcoding: Bridging Distribution and Inference Formats

A Huffman-encoded distribution bundle does not need to be re-encoded as palette to use the palette execution path. The `HuffmanPaletteTranscodeReconstructive` path performs Huffman→palette transcoding as its preprocess step, converting coded rows to packed nibbles on the fly. This means the same Huffman-only bundle can serve both distribution (maximum compression, fast GPU decode) and inference (palette-based execution where beneficial).

Conversely, a bundle can be pre-transcoded to palette encoding at model-load time if the deployment scenario favors eliminating all runtime Huffman preprocessing. This is the `PaletteReconstructive` path, which requires no preprocess kernels at all.

5.4 Layer Plan and Scheduling Independence

The bundle manifest stores a per-layer plan containing the layer index, a mode $\in$ {Bootstrap, Easy, Hard}, and a pointer to the next hard layer. This plan governs the runtime scheduler's slot-preserving prefetch strategy (§9) but does not constrain the encoding format. A layer classified as "Hard" (because it contains Huffman-encoded projections) can coexist with "Easy" layers (palette or raw) in the same model, and the scheduler exploits this heterogeneity to overlap preprocessing with computation.

6 Execution Pipelines and Work Distribution

6.1 The Core Scheduling Problem

Huffman decoding is inherently sequential per symbol: variable-length codes must be parsed one at a time within a row. This is a well-known limitation of entropy coding on SIMT architectures (see §2.1). Warp-cooperative approaches stall on the slowest thread, making Huffman poorly parallelizable at GPU scale.

Unweight's strategy is to **scatter decompression work across pipeline stages** so the sequential Huffman parsing can be overlapped with other-layer compute, and the matmul consumer only deals with fixed-length inputs. The three pipelines differ in how they split work between the preprocess kernel (which handles the hard, sequential part) and the consumer kernel (which completes reconstruction from uniform pieces).

6.2 Three Pipelines as a Work-Distribution Spectrum

Table 4 compares what each pipeline writes to HBM during preprocessing and what the consumer must read and reconstruct.

Pipeline	Preprocess writes	Consumer reads	Consumer work
Full decode + cuBLAS	BF16 (16b/elem)	BF16 from scratch	none (cuBLAS)
Exp decode + recon. matmul	exp only (8b/elem)	exp + σ (8b each)	combine $\varepsilon + \sigma \rightarrow$ BF16
Palette transcode + recon. matmul	palette idx (4b/elem, coded rows)	palette (4b) + σ (8b) + verbatim	resolve palette $\rightarrow \varepsilon$, combine $\rightarrow$ BF16

Table 4. Three pipelines as a work-distribution spectrum.

Moving down the table, the preprocess kernel writes less to HBM but shifts more reconstruction work to the matmul consumer:

- **Full decode** writes complete BF16 (sign+mantissa + exponent combined). The consumer (cuBLAS) is trivial — it reads a standard dense matrix.
- **Exp-only decode** writes only exponent bytes, skipping sign+mantissa entirely. This halves preprocess HBM writes. The reconstructive matmul combines exponent bytes with the permanent sign+mantissa tensor to produce BF16 in shared memory.
- **Palette transcode** writes packed 4-bit palette indices for coded rows — a further 2× reduction over exp-only. However, the reconstructive matmul must now resolve palette nibbles to exponent bytes before combining with sign+mantissa, and must handle verbatim rows separately via the tile bitmask.

6.3 Why Multiple Pipelines Are Necessary

A faster preprocess (fewer HBM writes) shifts more work to the matmul consumer. Whether this tradeoff pays off depends on the inference setup:

- At **small batch sizes**, fixed kernel launch and synchronization costs are significant relative to the matmul work, changing the balance between preprocess traffic and matmul overhead.
- At **large batch sizes**, sustained HBM bandwidth and tensor-core utilization dominate, shifting the balance toward different pipeline choices.
- **Gate/up and down projections** have different output dimensions, so they hit different reconstructive-matmul variant frontiers.
- **CTA count** is itself a tradeoff: more preprocess CTAs reduce standalone decode time but can starve the persistent matmul of SMs. On Hopper, a reconstructive matmul CTA requires ~227 KB of shared memory and a decode CTA requires ~16 KB. Since 227 + 16 > 228 KB (the per-SM limit), these CTAs cannot co-reside on the same SM. The available matmul SMs are therefore 132 − decode_CTAs (Figure 4).

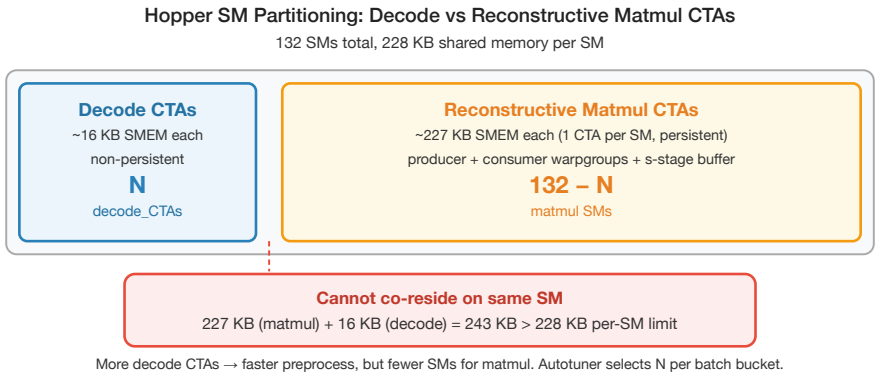

Fig. 4. Hopper SM partitioning constraint.

Different inference engines, deployment configurations, and workload profiles benefit from different balance points. This is why the autotuner (§10) selects per projection and per batch-size bucket rather than hard-coding a single pipeline.

7 Decode and Transcode Kernels

The decode and transcode kernels are the preprocess stage of the two reconstructive-matmul pipelines (§6.2). They parse the variable-length Huffman bitstream and produce fixed-length outputs that the reconstructive matmul can consume without any entropy-coding logic.

7.1 Design Rationale: Work-Stealing Over Warp-Cooperative Decode

Huffman decoding is sequential per row because variable-length codes must be parsed one at a time. The natural question is how to parallelize across the tensor.

Warp-cooperative decode (assigning one warp to one tile, with threads sharing the bitstream) would stall the entire warp on the thread processing the longest row. Since coded-row lengths vary significantly — a row of frequent exponents may use 2-bit codes while a row of rare exponents uses 10-bit codes — this creates severe load imbalance.

Work-stealing at row granularity avoids this: each thread independently claims one row at a time via a global atomic counter. The row is the minimal schedulable unit that matches the metadata structure (one start bit, one end bit, one verbatim decision, one output contract). This lets fast rows complete and claim new work while slow rows are still being parsed elsewhere.

The same work-stealing design is used for the encode kernel (§4.5) for the same reason: variable-length Huffman data creates uneven workloads that are poorly served by fixed-partition scheduling.

7.2 Unified Row Worker Algorithm

All three preprocess modes (full decode, exp-only decode, palette transcode) share the same kernel structure (Figure 5). The mode determines only what the output write produces.

Algorithm 4 Work-stealing decode / transcode — one thread

Require: decode_table in shared memory, tile_metadata, row_end_offsets, encoded_exp bitstream, verbatim_exponents, σ tensor
Ensure: depends on mode — BF16 scratch, u8 exponents, or packed 4-bit palette indices

Load decode_table from HBM to shared memory (once per CTA).
loop
    row_idx ← atomicAdd(global_counter, 1)
    if row_idx ≥ total_rows **then exit**
    end if
    tile_idx ← row_idx ≫ 6; row_in_tile ← row_idx & 63
    Load TileMetadata(tile_idx).
    if row is verbatim (bitmask test) **then**
        verbatim_index ← popcount(bitmask bits below row_in_tile)
        src ← verbatim_exponents[verbatim_row_offset + verbatim_index]
        if full-decode mode **then** combine src with σ → write 64 BF16 values.
        end if
        if exp-only mode **then** copy 64 exponent bytes to output.
        end if
        if palette mode **then** skip — verbatim rows handled by matmul consumer.
        end if
    else ▷ coded row
        Initialize HuffmanBitReader at tile_start_bit + row_start_bit.
        for each of 64 symbols **do**
            prefix ← __funnelshift_r(current_word, next_word, bit_idx)
            entry ← decode_table[prefix & (table_size − 1)]
            Emit entry.symbol / entry.palette_idx based on mode.
            bit_idx += entry.length; slide window if needed.
        end for
        Write 64 output values (BF16, u8, or packed nibbles) to HBM.
    end if
end loop

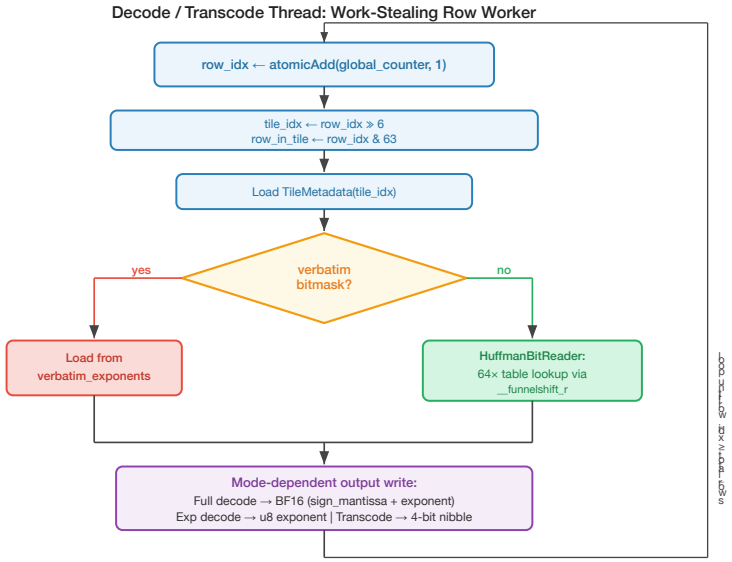

Fig. 5. Decode/transcode thread flow.

7.3 Branchless Bitreader

The `HuffmanBitReader` maintains a two-word sliding window over the bitstream. The key low-level optimization is `__funnelshift_r`, which extracts an aligned bit prefix from two adjacent u32 words in a single instruction, making word-boundary crossings branch-free:

```
u32 bits = __funnelshift_r(current_word, next_word, bit_idx);
HuffmanDecodeEntry entry = decode_table[bits & (DECODE_TABLE_SIZE - 1)];
```

Because the coded alphabet contains only palette symbols (verbatim rows are dispatched before the bitreader runs), every table entry is terminal. There is no escape path and no conditional second-stage lookup.

7.4 Why Flat Tables Over Hierarchical Alternatives

The flat shared-memory decode table (up to 32768 entries for 15-bit codes) is preferred over hierarchical cascaded tables (as used by DFloat11 [5]) for a straightforward reason: on Hopper with 228 KB of shared memory per SM, even the largest table is only 7% of capacity, so minimizing table size has no benefit. Meanwhile, cascaded lookups introduce dependent shared-memory loads on the slow path, which stall the warp on memory latency. We tested both a DFloat11-style cascade and a two-level fast-LUT optimization; both regressed throughput because the kernel is dominated by total work and HBM output traffic, not lookup latency.

8 Reconstructive Matmul

Reconstructive matmul is the consumer-side kernel in Unweight's fused pipelines. Its role is to complete BF16 reconstruction from fixed-length preprocess outputs — either exponent bytes or palette nibbles plus sign+mantissa — and feed the result directly to Hopper's WGMMA tensor-core instructions without materializing a dense BF16 weight matrix in HBM.

8.1 Why Reconstruction Happens in Shared Memory

Hopper WGMMA reads its B operand from shared memory in a swizzled layout. This is a hardware constraint, not a design choice. Given that B must end up in shared memory regardless, reconstructing it there is the most direct path: the producer loads compressed inputs via TMA, and the consumer assembles BF16 fragments in place. Register-side reconstruction would either duplicate work across warps or require cross-warp shuffling to reform the WGMMA operand layout.

8.2 Kernel Organization

The kernel uses the ThunderKittens [8] Load-Compute-Finish (LCF) pipeline template with asymmetric warpgroup roles:

- **Producer warpgroup** (1 warpgroup, reduced register allocation): stages A tiles, sign+mantissa tiles, and B-side source data (exponent tiles or palette-index tiles + verbatim rows) into a circular shared-memory buffer via TMA.
- **Consumer warpgroups** (2 warpgroups, expanded register allocation): reconstruct BF16 from the staged inputs and issue WGMMA.
- **Finish stage**: stores or accumulates the output C tile via TMA.

The following algorithm gives the CTA-level view. Let K be the inner dimension, s the number of pipeline stages, and σ / ε denote sign+mantissa and exponent bytes respectively.

Algorithm 5 Persistent reconstructive matmul — one CTA

Require: CTA assigned to output tile-group, s-stage circular SMEM buffer, $A \in$ BF16 in HBM, permanent σ tensor in HBM, and either (a) decoded ε tensor in HBM, or (b) packed palette-index tensor + TileMetadata + verbatim ε store

Initialize barriers, stage pointers, accumulator fragments.
if producer warpgroup **then**
  Deallocate consumer-only registers (setmaxnreg).
  for $t = 0 \ldots \lceil K/64 \rceil + s - 1$ **do**
    Wait for stage (t mod s) to be released by consumers.
    if $t < \lceil K/64 \rceil$ **then**
      TMA load A tile for k-step t. Load σ tile for k-step t.
      if direct-exponent mode **then** Load ε tile for k-step t.
      else ▷ palette mode
        Load palette-index tile and TileMetadata for k-step t.
        $v \leftarrow$ popcount(verbatim_bitmask) for this tile.
        if $v > 0$ **then** load exactly v verbatim ε rows (1×64 each).
        end if
      end if
      Commit stage (t mod s), notify consumers.
    end if
  end for
else ▷ consumer warpgroups (2×4 warps = 8 warps)
  Reallocate producer registers as reconstruction temporaries (setmaxnreg 232).
  Zero accumulator fragments.
  for $t = 0 \ldots \lceil K/64 \rceil - 1$ **do**
    Wait for stage (t mod s) to become visible.
    total_chunks $\leftarrow$ TILES_PER_CTA_N $\times$ 64 $\times$ 4
    for each chunk assigned to this warp's thread (strided) **do**
      Decompose chunk index $\rightarrow$ (n_tile_idx, local_row, chunk_idx)
      Load σ chunk (4× u32) from staged sign+mantissa tile via SMEM.
      if direct-exponent mode **then**
        Load ε chunk (4× u32) from staged exponent tile via SMEM.
      else if row is verbatim (bitmask test) **then**
        verbatim_row_idx $\leftarrow$ popcount(bitmask bits below local_row)
        Load ε chunk from staged verbatim buffer.
      else ▷ palette-coded row
        Load packed 4-bit palette indices (2× u32) from staged palette tile.
        for each u32 of packed indices **do**
          Expand nibbles; resolve via `__byte_perm`; mux $\rightarrow \varepsilon$ word.
        end for
      end if
      Interleave ε and σ via `__byte_perm` $\rightarrow$ BF16 pairs.
      Reconstruct BF16: signs | (exponents $\gg 1$) | mantissas.
      Store 8 BF16 values to swizzled WGMMA B layout in SMEM.
    end for
    reconstruction_barrier.arrive_and_wait()
    Issue WGMMA(A_tile, B_reconstructed). Wait for completion.
    Release stage (t mod s) to producer.
  end for
  Epilogue: TMA store / store_add of C.
end if

The producer absorbs all irregular memory access — including the variable-count verbatim-row loads — so that the consumer warpgroups see only a deterministic reconstruction task followed by WGMMA issue (Figure 6).

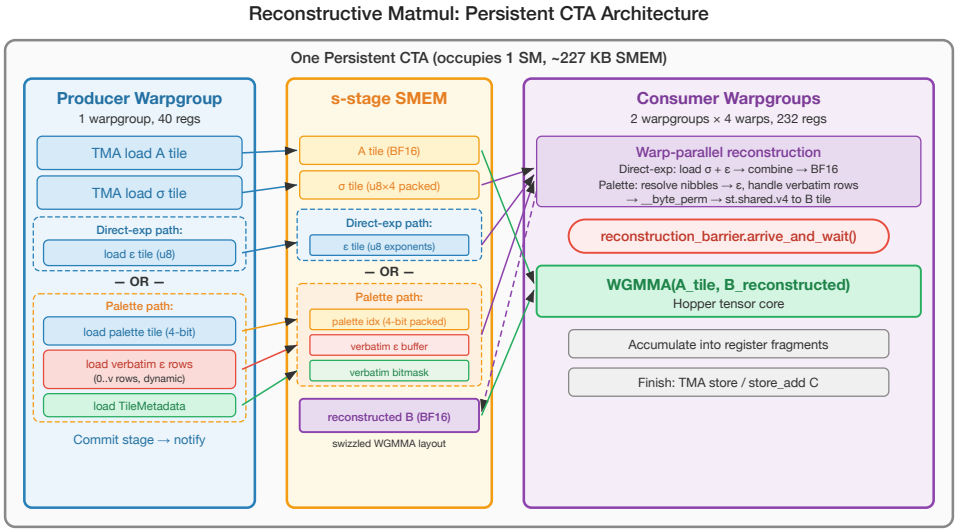

Fig. 6. Reconstructive matmul CTA architecture.

8.3 Two B-Side Source Contracts

The kernel supports two source models, matching the two reconstructive pipelines from §6:

Direct-exponent path. Each weight is reconstructed from one exponent byte ε (written by the exp-decode preprocess) plus one sign+mantissa byte σ (from the permanent tensor). The consumer reads both from staged shared-memory tiles and assembles BF16 with a shift-and-OR.

Palette path. Coded rows arrive as packed 4-bit palette indices. The consumer unpacks each nibble and resolves it to an exponent byte through a 16-entry palette cache using two `__byte_perm`-based 8-way lookups and a one-bit mux to select `palette[0:7]` versus `palette[8:15]` (Figure 7). Verbatim rows bypass the palette entirely: the producer stages their raw exponent bytes, and the consumer reads them directly. Row classification is a single bitmask check, not a per-element branch.

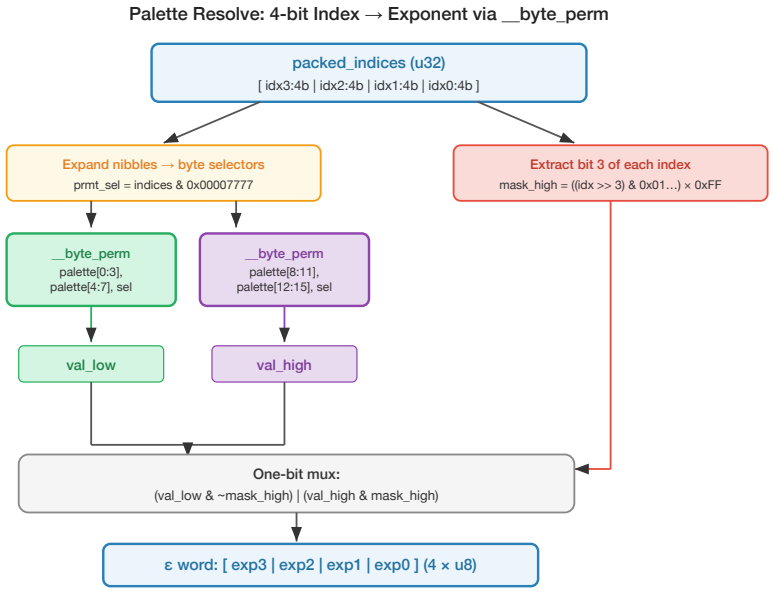

Fig. 7. Palette resolve via `__byte_perm`.

8.4 Variant Family and Shared-Memory Budget

Each variant is parameterized by two dimensions: the number of output-N tiles per CTA (how many 64-column B tiles a CTA handles per task iteration) and the number of pipeline stages s (the depth of the circular shared-memory buffer). All variants use `TILES_PER_CTA_M = 2` (two 64-row output tiles) and two consumer warpgroups.

More N tiles improve output locality per CTA. Deeper staging hides TMA load latency by allowing the producer to run ahead of the consumer. Both consume shared memory. Hopper provides 228 KB per SM. Table 5 lists the available variants.

Variant	N tiles	Stages	Tradeoff
N1P6	1	6	Deepest staging, narrowest output. Best latency hiding.
N1P4	1	4	Moderate staging depth at narrow output.
N1P2	1	2	Minimal staging at narrow output.
N2P4	2	4	Default small-batch. Good staging/width balance.
N2P3	2	3	Less staging than N2P4. Frees SMEM for verbatim.
N2P2	2	2	Minimal staging, standard width. Most SMEM headroom for large verbatim budgets.
N3P2	3	2	Wide output, minimal staging. Req. $N \bmod 192 = 0$.
N4P2	4	2	Widest output, large-batch best. Req. $N \bmod 256 = 0$.

Table 5. Reconstructive matmul variant family.

Dimension compatibility. A variant with n output-N tiles requires $N \bmod (n \times 64) = 0$. For example, N4P2 requires N divisible by 256. The autotuner filters candidates by shape compatibility before benchmarking.

Default batch-size heuristic. Before autotuning, the system uses a simple heuristic: for batch sizes below 256, prefer deep-staging variants (N2P4, N2P3, N2P2) because producer-ahead latency hiding matters more; for batch sizes ≥ 256, prefer wide-output variants (N4P2, N3P2, N2P4) because output-tile locality matters more.

Palette path: verbatim-row staging. The palette path adds a third parameter: `MAX_VERBATIM_ROWS` $\in \{0, 4, 8, 16, 32, 64\}$, selected from the tensor's observed maximum verbatim count per tile. Each staged verbatim row is a 1×64 byte tile in shared memory. As this budget grows, the feasible variant frontier shifts toward fewer pipeline stages:

MAX_VERBATIM_ROWS	Feasible variants (palette path)
0–8	All variants
16	Variants with $s \leq 4$
32	Variants with $s \leq 2$
64	Variants with $s \leq 2$, narrow N only

Table 6. Feasible variants by verbatim-row budget.

The shared-memory budget must simultaneously accommodate the WGMMA accumulator tiles, the s-stage circular input buffer (A tiles, sign+mantissa tiles, exponent/palette tiles), and the verbatim staging area. When the verbatim budget is large, only shallow-pipeline variants fit.

There is no single best variant. The optimal choice is a point on the shared-memory vs. latency-hiding frontier, and that point changes with batch size, projection shape, and verbatim-row statistics. The autotuner (§10) selects the variant per projection per batch bucket.

8.5 Known Limitations and Future Improvements

The current reconstructive matmul design has three known inefficiencies that represent active areas of improvement.

Down-projection dimension asymmetry. In a SwiGLU MLP, gate and up projections have shape `B[intermediate_size, hidden_size]` while the down projection has shape `B[hidden_size, intermediate_size]`. For Llama 3.1 8B this means gate/up have N=14336, K=4096 and down has N=4096, K=14336. The reconstructive matmul iterates over K in its inner loop, reconstructing `TILES_PER_CTA_N` B tiles per k-step. Down therefore performs 3.5× more reconstruction steps per CTA (224 vs 64 k-steps), while having fewer output-N tiles to parallelize across. The ratio of reconstruction work to output parallelism is significantly worse.

	Gate/Up	Down
Dimensions	$N = 14336, K = 4096$	$N = 4096, K = 14336$
k-steps per CTA	64	224
Output N-tiles	112 (N2P4)	32 (N2P4)
Reconstruction	low K, wide output	high K, narrow output

Table 7. Down-projection dimension asymmetry (Llama 3.1 8B).

Potential mitigations include dimension-aware kernel specialization for transposed shapes, or restructuring the grid schedule to better exploit the narrower output dimension.

Redundant B-tile reconstruction at large batch sizes. The persistent grid is capped at 132 CTAs. When the number of output tiles exceeds 132, CTAs cycle through multiple task iterations. Different output tiles in the M dimension share the same B-column tiles, but each CTA reconstructs its B tiles from compressed data independently — there is no cross-CTA reuse of decoded B data.

For gate/up projections with variant N2P4, redundant reconstruction begins at padded_m ≥ 256 (batch size ~256). For N4P2 at large batches, it begins around batch ~384. Two mitigation strategies are under consideration:

- **HBM decode cache**: the first CTA to reconstruct a B tile publishes the decoded BF16 data to an HBM cache. Subsequent CTAs that need the same tile load the cached version instead of re-decoding. This trades one HBM write + read for the reconstruction work, which is profitable when many CTAs share the tile.
- **N-column-locality scheduling**: restructure the grid schedule so that output tiles sharing the same B columns are assigned to consecutive task iterations on the same CTA. This keeps the decoded B tile resident in shared memory across iterations, avoiding both re-decode and HBM round-trips. This requires replacing the current M-row-grouped schedule with a schedule that prioritizes B-column reuse.

Small-batch overhead. The LCF pipeline and WGMMA design impose fixed costs that do not pay off at small matrix dimensions:

- The producer/consumer warpgroup split wastes the producer's compute capability when the matmul is small.
- All variants pad M to at least 128 (two 64-row tiles). At batch size 1, this means 127 of 128 rows are zero padding.
- Pipeline barriers, stage rotation, and register reallocation (`setmaxnreg`) have fixed costs that dominate when the actual compute is tiny.
- The 132-CTA persistent grid is wasteful when there are only a few output tiles.

For batch sizes below ~32, a simpler ALU-based decode+matmul kernel — without warpgroup specialization, TMA, or WGMMA — would likely perform better by avoiding this overhead entirely. This is an active area of development.

9 Runtime Orchestration

This section describes how the compressed MLP runtime orchestrates preprocess and consumer kernels across layers, streams, and double-buffered slots. We first describe the baseline (uncompressed) MLP computation, then the compressed variant.

9.1 Baseline SwiGLU MLP

The MLP block in SwiGLU [16] transformer models computes:

$$\mathrm{MLP}(x) = W_{\text{down}} \cdot \big(\mathrm{SiLU}(W_{\text{gate}} \cdot x) \odot (W_{\text{up}} \cdot x)\big) + x$$

where $W_{\text{gate}}, W_{\text{up}} \in \mathbb{R}^{d_{\text{ff}} \times d}$, $W_{\text{down}} \in \mathbb{R}^{d \times d_{\text{ff}}}$, and $\mathrm{SiLU}(z) = z \cdot \sigma(z)$. For Llama 3.1 8B [9], $d = 4096$ and $d_{\text{ff}} = 14336$.

The gate and up projections are independent matmuls over the same input x and can execute concurrently. The down projection depends on both and must wait for their completion. The runtime maps this to two CUDA streams:

(1) gate $\leftarrow W_{\text{gate}} \cdot x$ on stream 0
(2) up $\leftarrow W_{\text{up}} \cdot x$ on stream 1
(3) sync(stream 0, stream 1)

(4) gate ← SiLU(gate) ⊙ up on stream 0
(5) output ← W_{down} · gate + residual on stream 0

9.2 Layer Classification

The bundle plan classifies each layer into one of three modes:

- **Bootstrap**: the first layer, which uses uncompressed weights and seeds the overlap pipeline by launching preprocess for the first hard layer.
- **Hard**: at least one projection requires Huffman preprocessing (decode or transcode). Hard layers consume a preprocess slot and launch preprocess for the next hard layer.
- **Easy**: all projections are palette-encoded or raw. No preprocess kernels are needed; any in-flight preprocess slot from a prior hard layer is preserved.

9.3 Stream Topology and Slot Management

The runtime uses 8 CUDA streams:

```
Consumer streams:   stream 0 (gate, down, SiLU)
                    stream 1 (up)

Preprocess streams: slot 0: stream 2 (gate), stream 3 (up), stream 4 (down)
                    slot 1: stream 5 (gate), stream 6 (up), stream 7 (down)
```

Two preprocess **slots** provide double-buffering: while one slot's preprocess results are being consumed by the current hard layer, the other slot can be filled with preprocess results for the next hard layer. Slots are assigned deterministically:

$$\text{slot\_for\_hard\_layer}(\ell) = \lfloor \ell/2 \rfloor \bmod 2$$

Each slot contains separate output buffers for each projection (gate, up, down) in each format that may be needed (decoded BF16, decoded exponents, or palette indices), plus per-projection atomic row counters for work-stealing.

9.4 Compressed MLP Forward Pass

Algorithm 6 Compressed MLP forward pass — all layers

Require: layer plan, compressed weights, 2 consumer streams, 6 preprocess streams, 2 double-buffered preprocess slots, hard/easy execution configs per batch bucket

for each layer ℓ **do**
  config ← resolve_exec_config(batch_size)
  if mode(ℓ) = Bootstrap **then**
    if next_hard_layer(ℓ) exists **then**
      next_slot ← slot_for_hard_layer(next_hard_layer(ℓ))
      LAUNCH_PREPROCESS(next_hard weights, next_slot, config.hard)
    end if
    Execute baseline SwiGLU MLP (§9.1) on raw weights.
  else if mode(ℓ) = Easy **then**
    Execute compressed MLP with easy config. Preserve in-flight preprocess slot.
  else ▷ Hard
    slot ← slot_for_hard_layer(ℓ)
    if next_hard_layer(ℓ) exists **then**
      next_slot ← slot_for_hard_layer(next_hard_layer(ℓ))
      LAUNCH_PREPROCESS(next_hard weights, next_slot, config.hard)
    end if
    sync(consumer_stream_1, up_preprocess_stream(slot))
    up ← $W_{\text{up}} \cdot x$ on stream 1
    sync(consumer_stream_0, gate_preprocess_stream(slot))
    gate ← $W_{\text{gate}} \cdot x$ on stream 0
    sync(stream 0, stream 1)
    gate ← SiLU(gate) $\odot$ up on stream 0
    sync(consumer_stream_0, down_preprocess_stream(slot))
    output ← W_{down}· gate + residual on stream 0
  end if
end for

Algorithm 7 LAUNCH_PREPROCESS(W, slot, config)

for each projection $p \in \{\text{gate}, \text{up}, \text{down}\}$ **do**
  if W[p] is Huffman-encoded **then**
    preprocess_stream ← stream_for(p, slot)
    sync(preprocess_stream, consumer_stream_for(p)) ▷ buffer protection
    Reset atomic row counter for (slot, p).
    match config[p]:
      FullDecode → launch decode_tiles(FULL)
      ExpDecode → launch decode_tiles(EXP)
      PaletteTranscode → launch transcode
  end if
end for

9.5 Key Orchestration Properties

Buffer protection. Before writing to a slot's preprocess buffer, the preprocess stream synchronizes against the consumer stream that last read from it (`sync(preprocess_stream, consumer_stream_for(p))`). This prevents overwriting data that an in-flight matmul is still reading.

Concurrent preprocess. Gate, up, and down preprocess kernels for the same slot launch on three separate streams and run concurrently — they have no cross-dependencies.

Down projection gets maximum overlap. The down projection is consumed last in the MLP sequence (after gate, SiLU, and up), so its preprocess has the longest time to complete before the consumer stream waits on it.

Row counter reset. Each preprocess kernel gets a fresh atomic counter per (slot, projection) pair. This counter drives work-stealing across tile rows within the decode/transcode kernel.

Multi-layer timeline. Figure 8 shows how preprocess and consumer work overlap across layers.

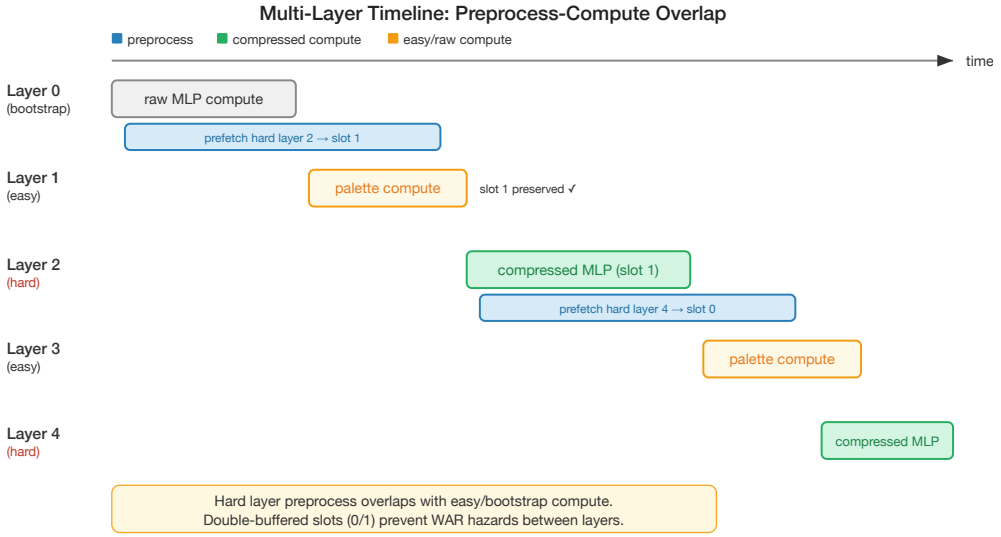

Fig. 8. Multi-layer timeline: preprocess-compute overlap.

Pipeline tradeoffs cannot be discussed in isolation from this orchestration. A hard layer may tolerate a heavier preprocess stage because that work is overlapped under the compute of intervening easy layers. An easy layer cannot assume the same slack. This is why the autotuner (§10) tunes separate hard and easy execution profiles.

10 Autotuning Methodology

The autotuner selects the best execution configuration for each projection and batch-size bucket by measuring end-to-end decode throughput on Infire [12].

10.1 Search Space

The tuner operates on a set of batch-size buckets (e.g. $\{1, 64, 256, 512, 1024\}$). For each bucket it tunes separate **hard** and **easy** layer profiles. Candidate projection configurations are generated from the encoding template: raw projections map to cuBLAS; palette projections map to compatible reconstructive-matmul variants; Huffman projections map to all three pipeline options (full-decode + cuBLAS, exp-decode + reconstructive, palette-transcode + reconstructive) with compatible matmul variants.

Candidate generation is dimension-aware: gate/up projections use the intermediate dimension, while down uses the hidden dimension.

10.2 Coordinate-Descent Search

Exhaustive search over the Cartesian product of gate, up, down, and CTA count choices is too expensive when every point requires real GPU measurement. The tuner uses coordinate descent:

Algorithm 8 Autotuning — one batch bucket

Require: candidate sets C_{gate}, C_{up}, C_{down}, CTA count candidates C_{cta}, throughput oracle Bench

for profile $p \in \{\text{hard}, \text{easy}\}$ **do**
  repeat
    Sweep C_{gate} while holding up/down fixed. Keep best.
    Sweep C_{up} while holding gate/down fixed. Keep best.
    Sweep C_{down} while holding gate/up fixed. Keep best.
  until no improvement.
  Revisit per-projection alternatives; accept only improvements $> 2\%$.
  if profile uses Huffman preprocessing **then**
    Sweep preprocess CTA counts from C_{cta}. Keep best.
  end if
end for
Emit best hard/easy configurations with measured throughput.

10.3 Benchmark Protocol

The tuner benchmarks the complete Infire [12] inference engine, not isolated kernels. This is critical because pipeline selection affects overlap, SM contention, and synchronization — effects that kernel microbenchmarks cannot capture.

Each configuration change triggers a CUDA graph re-capture and warmup phase before measurement. The throughput score is tokens/s = (batch_size × decode_steps)/median_time. Intermediate results are cached to allow resumption from partial runs.

11 Evaluation

This is a technical report for ongoing research, not a final pre-print. The evaluation presented here is preliminary and not yet comprehensive.

11.1 Testbed

The primary development and validation model is **Llama 3.1 8B Instruct** [9]. This choice is deliberate:

- Llama 3.1 uses a SwiGLU MLP architecture whose exponent distributions are statistically representative of SwiGLU models at other scales. Fan et al. [7] demonstrate that MLP exponent distributions share the same statistical properties across models with this architecture — the top exponents and their frequency ranks are consistent regardless of model size.
- The 8B parameter count fits on a single Hopper GPU, enabling fast engineering iteration without multi-GPU coordination overhead.
- Compression ratios are expected to be similar across SwiGLU models due to shared exponent statistics, but kernel configurations and autotuning results are model- and inference-stack-dependent and would need to be re-tuned for each deployment. Llama 3.1 8B serves as the engineering iteration testbed before scaling to larger models.

11.2 Compression Ratios

Two bundle configurations are evaluated on Llama 3.1 8B Instruct (14.97 GB original), shown in Table 8:

Inference configuration (HHR-PPR): hard layers use Huffman/Huffman/Raw for gate/up/down; easy layers use Palette/Palette/Raw. Down projection is excluded from compression due to the dimension-asymmetry limitation described in §8.5.

Distribution configuration (full Huffman): all projections (gate, up, down) use Huffman encoding across all layers. This maximizes compression and can be transcoded to the inference configuration on model load.

Configuration	Gate/Up enc.	Down enc.	Size	Ratio
Uncompressed baseline	—	—	14.97 GB	100%
Inference (HHR-PPR)	Huff. ~68% / Pal. ~75%	Raw 100%	13.07 GB	87.3%
Distribution (all Huff.)	Huffman ~68%	Huffman ~68%	11.74 GB	78.4%

Table 8. Compression ratios for Llama 3.1 8B Instruct (14.97 GB original).

Per-projection compression ratios are consistent across layers: Huffman-encoded MLP projections compress to ~68.1–68.6% of original size, palette-encoded projections to ~75.4–75.6%. The difference reflects the palette IR overhead (packed 4-bit indices plus verbatim metadata vs. raw Huffman bitstream).

The overall bundle ratios (87.3% and 78.4%) are higher than the per-projection ratios because Unweight compresses only MLP projections. Attention weights, embeddings, layer norms, and the language model head — which together account for roughly one-third of the model — are stored uncompressed. The MLP projections themselves achieve ~32% compression (Huffman) or ~25% compression (palette), but this is diluted by the uncompressed remainder of the model.

11.3 End-to-End Inference Throughput

Measured on H100 SXM5 with the inference configuration (HHR-PPR). Down projection uses uncompressed cuBLAS. Throughput is measured as decode tokens/s across batch-size buckets (Table 9). Each measurement is the median of 5 runs; standard deviations were below 2% of the median for all batch sizes and are omitted from this table (see § 11.6 limitation 4):

Batch	Baseline (tok/s)	Unweight (tok/s)	Penalty
1	168	100	−40.7%
64	7,752	5,136	−33.7%
256	18,097	11,677	−35.4%
512	21,685	14,406	−33.5%
1024	23,464	16,520	−29.6%

Table 9. End-to-end inference throughput (tokens/s) on H100 SXM5.

The throughput penalty narrows at larger batch sizes (from ~41% at batch 1 to ~30% at batch 1024), consistent with the expectation that preprocess overlap improves as compute work grows relative to fixed decode costs. The batch-1 penalty is disproportionately large due to the small-batch overhead described in §8.5: the LCF/WGMMA pipeline's fixed costs dominate when the matmul is tiny.

These numbers reflect the current state without down-projection compression, without the small-batch kernel optimization, and without the B-tile caching or N-column-locality scheduling

improvements described in §8.5. All three are active areas of improvement that are expected to narrow the throughput gap.

11.4 Individual Kernel Throughput

Standalone kernel benchmarks on a single gate_proj tensor (14336 × 4096, layer 2). These measure isolated kernel performance, not end-to-end pipeline throughput. All times are medians of 100 runs after 50 warmup iterations. Run-to-run variance was below 1% for all kernel measurements.

Preprocess kernels (128 CTAs). Input throughput is measured against the exponent tensor size (14336 × 4096 = 58.7 M elements, 1 byte each). Output throughput is computed from the actual bytes written, which differs per kernel:

Kernel	Time	In thrpt	Out size	Out thrpt
Exp-only decode	130.0 μs	420.8 GiB/s	1 B/elem (u8)	420.8 GiB/s
Full BF16 decode	193.3 μs	282.9 GiB/s	2 B/elem (bf16)	565.8 GiB/s
Palette transcode	116.5 μs	469.3 GiB/s	0.5 B/elem (4-bit)	234.6 GiB/s

Table 10. Preprocess kernel throughput (gate_proj, 128 CTAs).

Palette transcode is the fastest preprocess option by wall time — it performs the same Huffman parsing as exp-only decode but writes half the bytes (packed 4-bit indices vs u8 exponents). Full decode is slowest by wall time but has the highest output bandwidth because it writes 2-byte BF16 values (combining exponents with sign+mantissa).

Matmul comparison (gate_proj, 14336 × 4096). Throughput is in mega-elements/s (Me/s) or giga-elements/s (Ge/s):

Batch	cuBLAS	Recon. (exp)	Recon. (palette)
1	86.1 μs / 166.5 Me/s	109.1 μs / 131.4 Me/s	128.1 μs / 111.9 Me/s
64	87.7 μs / 10.5 Ge/s	108.2 μs / 8.48 Ge/s	128.7 μs / 7.13 Ge/s
256	93.8 μs / 39.1 Ge/s	149.6 μs / 24.5 Ge/s	209.9 μs / 17.5 Ge/s
512	135.2 μs / 54.3 Ge/s	265.8 μs / 27.6 Ge/s	387.6 μs / 18.9 Ge/s
1024	213.1 μs / 68.9 Ge/s	458.6 μs / 32.0 Ge/s	700.2 μs / 21.0 Ge/s

Table 11. Matmul throughput comparison (gate_proj, 14336 × 4096).

At small batch sizes (1–64), the reconstructive matmul overhead is dominated by the LCF pipeline's fixed costs (§8.5). At larger batch sizes (256+), the gap widens further because the persistent grid (132 CTAs) begins reconstructing the same B tiles redundantly across multiple task iterations — the redundant B-tile reconstruction problem described in §8.5. cuBLAS does not have this overhead because it reads a pre-materialized dense matrix. The palette variant is consistently slower than the direct-exponent variant because the consumer must additionally resolve palette nibbles to exponent bytes before combining with sign+mantissa.

Both the B-tile caching/scheduling improvements and the small-batch ALU kernel described in §8.5 are expected to narrow this gap significantly. In the end-to-end pipeline, the matmul overhead is also partially hidden by overlapping preprocess with adjacent-layer compute.

11.5 Encoding Throughput

Encoding is an offline operation that runs on GPU. The following measures the full end-to-end encoding pipeline per tensor (gate_proj, 14336 × 4096, 112 MB BF16 input):

Encoder	Time	Input throughput
Huffman encode	5.2 ms	21.0 GiB/s
Palette encode	3.3 ms	33.2 GiB/s

Table 12. Encoding throughput per tensor (gate_proj, 14336 × 4096).

These are end-to-end times (median of 100 runs) including tensor allocation and host↔device memory transfers, not just kernel execution. Huffman encoding includes all six kernel stages: statistics collection, table construction, tile metadata computation, global prefix sum, and bitstream encoding. Palette encoding shares the first four stages but replaces Huffman bitstream writing with direct palette-index packing, which writes fewer bytes and avoids the variable-length bitstream logic.

At these speeds, encoding the full Llama 3.1 8B MLP weights (~10 GB of BF16 projections) takes roughly 0.5–1 second on Hopper — fast enough for on-the-fly bundle preparation. These numbers are relevant for the distribution use case (how fast a model can be compressed for transfer) and for understanding the one-time offline cost.

11.6 Limitations

The following limitations apply to the current system and evaluation:

(1) **Single-model evaluation.** All results are measured on Llama 3.1 8B Instruct. While compression ratios are expected to generalize across SwiGLU models due to shared exponent statistics (§ 11.1), kernel configurations and end-to-end throughput are model- and inference-stack-dependent. Evaluation on larger models (70B+) and non-SwiGLU architectures is pending.

(2) **Down-projection exclusion.** The inference configuration leaves down projections uncompressed due to the dimension-asymmetry limitation described in § 8.5. This reduces the effective compression ratio from ~32% on compressed projections to ~20% total model size reduction. Addressing this requires dimension-aware kernel specialization.

(3) **Throughput overhead.** The current end-to-end throughput overhead ranges from ~30% at batch 1024 to ~41% at batch 1. Three known sources are being addressed: small-batch LCF/WGMMA fixed costs, redundant B-tile reconstruction at large batches, and the down-projection exclusion (§ 8.5).

(4) **No variance reporting.** Kernel and end-to-end measurements in this report are median values from repeated runs. Standard deviations and confidence intervals are not yet reported. Future revisions will include error bars on all performance measurements.

(5) **Hopper-only.** All kernels target NVIDIA Hopper (H100/H200) specifically. Adapting to Blackwell or other architectures is future work.

(6) **MLP-only compression.** Attention weights, embeddings, layer norms, and the language model head are stored uncompressed. These account for roughly one-third of the model and dilute the overall compression ratio.

(7) **No comparison with quantization.** This report evaluates lossless compression in isolation. Head-to-head throughput comparisons with lossy quantization approaches (FP8, INT4) that achieve higher compression at the cost of numerical fidelity are not included.

12 Active Research Directions

All aspects of Unweight are under active development. This report presents intermediate results; the following directions are being pursued concurrently:

Dense inference pipeline iteration. The kernel implementations, variant selection heuristics, and runtime orchestration described in this report continue to evolve. Ongoing work includes further decode-granularity tuning, additional reconstructive-matmul variants, and improved overlap scheduling.

MoE cold-expert compression. Frontier Mixture-of-Experts models are enormous and require GPU clusters. Most experts are infrequently activated and not worth keeping permanently resident in HBM. Compressed MLP weights reduce the per-expert memory footprint and transfer cost when experts must be loaded on demand. The composable nature of Unweight's kernels makes this feasible: the same reconstructive matmul that serves dense inference can consume compressed expert weights fetched just-in-time.

Distribution compression. The encoder and decoder can be used standalone for fast GPU-accelerated model bundle compression, independent of any inference engine. A whole model can be Huffman-encoded for distribution, reducing checkpoint transfer time without numerical compromise.

Compression methodology research. A separate research stream investigates new compression methodologies for LLM weights beyond exponent-domain Huffman coding, including quantized weight formats. This engineering report covers only the systems side of the program.

13 Summary

This report presented intermediate results from the engineering stream of the Unweight research program: a composable toolkit for lossless LLM weight compression targeting dense inference, model distribution, and MoE serving on NVIDIA Hopper (H100/H200).

The current system design is organized around five ideas:

First, outlier exponents are handled at row granularity through verbatim rows, rather than through per-symbol escape codes. This eliminates branches from the decode and matmul hot paths.

Second, restricting the coded alphabet to a 16-value palette makes the Huffman decoder branchless and terminal-table based. This fits Hopper's throughput-oriented execution model better than an escape-oriented symbol stream.

Third, the decompression workload is scattered across pipeline stages: preprocess kernels handle the sequential Huffman parsing, and the reconstructive matmul completes reconstruction from fixed-length pieces in shared memory before WGMMA issue. Different pipelines shift different amounts of work to the matmul consumer.

Fourth, encoding format, execution pipeline, and scheduling are orthogonal choices connected by runtime transcoding. A Huffman-encoded distribution bundle can be transcoded to palette form on load; the same kernels serve both use cases.

Fifth, pipeline choice is inherently batch-, projection-, and orchestration-dependent. Unweight treats pipeline selection, matmul variant choice, and preprocess CTA count as autotuned runtime decisions rather than fixed architectural constants.

All of this work is ongoing. The kernels are being open-sourced (https://github.com/cloudflareresearch/unweight-kernels), and both the engineering and compression-research streams continue to iterate on dense, MoE, and distribution applications.

References

[1] S. Han, H. Mao, and W. J. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In *ICLR*, 2016. arXiv:1510.00149.

[2] J. Johnson. DietGPU: GPU-based lossless compression for numerical data. Meta Research, 2023. https://github.com/facebookresearch/dietgpu
[3] G. Hershcovitch, A. H. Cim, B. Ilani, and N. Peia. ZipNN: Lossless Compression for AI Models. arXiv:2404.15198, 2024.
[4] Y. Hao, S. Cao, H. Guo, and Z. Zhang. NeuZip: Memory-Efficient Training and Inference with Dynamic Compression of Neural Networks. arXiv:2410.20650, 2024.
[5] C. Zhang, J. Cheng, I. E. Kang, P. Guo, Y. Wang, A. Borzunov, and B. Yan. DFloat11: Decoding 11-bit Brain Floats with Lookup Tables. In *NeurIPS*, 2025. arXiv:2411.16784.
[6] T. Yubeaton, K. Cheng, H. Liu, W. Jiang, and J. Chen. Huff-LLM: Huffman Coding-Based Compression for Energy-Efficient LLM Accelerators. arXiv:2503.15134, 2025.
[7] S. Fan, S. F. Yeh, T. Zhao, G. Zhu, Y. Wang, J. Gu, and Y. Zhang. ZipServ: Exploiting BFloat16 Sparsity for Efficient LLM Serving. In *ASPLOS*, 2026. arXiv:2505.10554.
[8] B. Spector, S. Singla, and A. Gu. ThunderKittens: Simple, Fast, and Adorable AI Kernels. arXiv:2410.20399, 2024.
[9] A. Dubey et al. The Llama 3 Herd of Models. arXiv:2407.21783, 2024.
[10] NVIDIA. CUDA C++ Programming Guide. NVIDIA Corporation, 2024. https://docs.nvidia.com/cuda/cuda-c-programming-guide/
[11] NVIDIA. Parallel Thread Execution ISA Version 8.5. NVIDIA Corporation, 2024. https://docs.nvidia.com/cuda/parallel-thread-execution/
[12] Cloudflare. Cloudflare's Most Efficient AI Inference Engine. Cloudflare Blog, 2025. https://blog.cloudflare.com/cloudflares-most-efficient-ai-inference-engine/
[13] G. Hershcovitch, B. Ilani, A. H. Cim, N. Peia, and M. Bercovich. Lossless Compression of Neural Network Components: Weights, Checkpoints, and K/V Caches in Low-Precision Formats. arXiv:2508.19263, 2025.
[14] D. A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
[15] NVIDIA. NVIDIA H100 Tensor Core GPU Architecture. NVIDIA Whitepaper, 2022.
[16] N. Shazeer. GLU Variants Improve Transformer. arXiv:2002.05202, 2020.

A Explored Alternatives and Negative Results

Several design alternatives were evaluated and rejected during development. We document them here because they represent plausible approaches that did not work in practice.

A.1 Cascaded Multi-Level Decode Tables

DFloat11 [5] uses cascaded 8-bit sub-tables to keep the decode table small (~2.3 KB). We evaluated a similar approach: replace the flat 12-bit table with a chain of 8-bit lookups where misses redirect to child sub-tables. The problem is that multi-level misses create dependent shared-memory loads — each miss adds ~30 cycles of SMEM latency, and warp divergence on the slow path stalls all 32 threads. On Hopper with 228 KB of shared memory, the flat table (up to 16 KB) is only 7% of capacity, so the memory savings are worthless. The cascaded approach was not implemented beyond analysis.

A.2 Two-Level Fast LUT

A less aggressive variant: keep the flat 12-bit table as the authoritative decoder, but add a small 8-bit prefix table as a fast path. Symbols with code length ≤ 8 (which account for ~99.6% of lookups) would be decoded in one 256-entry lookup; the remaining ~0.4% would fall through to the full 12-bit table. Host-side replay confirmed the hit rate was high enough to be plausible, but the actual CUDA implementation regressed throughput by ~10%. The kernel is dominated by total work and HBM output traffic, not lookup latency — the extra prefix handling and per-symbol branching cost more than the lookup it saved.

A.3 Dense-Prefix Escape Layout for Palette Transcoder

Before verbatim rows, the palette transcoder handled non-palette exponents via per-element escapes: a packed palette-index plane plus a dense escape plane. We evaluated a split layout where

the first 16 columns of each row stored palette indices and the remaining columns stored escape exponents, using `__popc`-based ordinal advancement to index the escape region. This regressed transcode throughput from ~400 GiB/s to ~330–365 GiB/s because the compaction logic introduced serial dependency chains in the hot loop. The old sparse write path (predicated stores at fixed column offsets) generated better machine code. This failure motivated the move to verbatim rows, which eliminates per-element escape handling entirely.

A.4 Warp-Cooperative Huffman Decode

An alternative to per-thread work-stealing: assign one warp to one tile, with 32 threads cooperatively parsing the bitstream for 32 rows. The appeal is amortizing metadata loads across the warp. The problem is that coded-row lengths vary significantly (2-bit codes vs. 10-bit codes), and the warp must wait for the slowest thread to finish before moving to the next tile. In practice this creates severe load imbalance that eliminates the metadata-sharing benefit. Per-thread work-stealing with atomic row dispatch consistently outperforms it.

A.5 Register-Side BF16 Reconstruction

ZipServ [7] reconstructs BF16 values in registers and feeds `mma.sync` directly. This works for their fixed-length bitmap encoding because each thread's data is deterministic. For Huffman-decoded data, register-side reconstruction would require cross-warp shuffling to reform the WGMMA operand layout (which expects data in shared memory with a specific swizzle pattern). The shuffling overhead would negate the benefit. Shared-memory reconstruction is the natural fit for Hopper's WGMMA interface.