TurboQuant-SGLang: Compressed KV Cache Attention as a Plugin Backend

Jesse Morgan
Thornveil LLC
jesse@thornveil.ai

Abstract—Long-context inference on large Mixture-of-Experts (MoE) models is bottlenecked by KV cache memory: at 128K tokens, the KV cache of a 122B-parameter hybrid MoE model consumes gigabytes of VRAM, directly competing with model weights for the GPU memory budget. We present TurboQuant-SGLang, an integration of Google's TurboQuant KV cache compression algorithm—3-bit keys with QJL residual correction and 2-bit values with group quantization—into the SGLang inference engine via a clean wrapper backend architecture that requires no source modification at integration time. The key design is a TurboQuantAttnBackend that wraps SGLang's TritonAttnBackend inside the HybridLinearAttnBackend, intercepting KV cache writes for compression and decompressing on reads. A per-request state pool (RequestTQPool) maintains compressed KV buffers per request in batched workloads, extending TurboQuant from single-sequence to production batched inference. The integration proceeded through 5 implementation phases: capture-only benchmarking, hybrid attention integration, Triton kernel path, batched scheduling, and memory optimization. A critical discovery was that the Qwen3.5-122B model uses head_dim=256 (explicit in config.json), not the 96 implied by the hidden-size / num-heads ratio—TurboQuant's QJL projection dimension scales with head dimension, making correct discovery essential. CUDA graph support is limited to non-graph-captured forward passes; standard attention executes during graph replay without TQ compression active. Measured on Qwen3.5-122B-A10B-GPTQ-Int4 with 2 KV heads per layer and 12 full-attention layers: 5.2× KV cache compression (449 MB vs. 2,344 MB at 100K tokens), 77,658 tok/s compression kernel throughput (Phase 1 capture benchmark, not end-to-end generation latency), NMSE = 0.18, SNR = 7.4 dB, and attention score cosine similarity of 0.922. This paper presents an architecture contribution and proof of concept. Perplexity evaluation on a 20-passage factual corpus confirms that TurboQuant introduces negligible quality degradation: mean PPL = 3.634 (TQ on) vs. 3.637 (TQ off), a delta of −0.08% (TQ is marginally lower, within measurement noise). End-to-end latency profiling across five context lengths (100–16K tokens) confirms that TTFT overhead is 1–6% and per-token decode latency (ITL) overhead is 23–31% under --disable-cuda-graph, reflecting compression and decompression kernel cost. The deployed KV cache is FP16 (not FP8); TurboQuant's 5.2× compression over FP16 represents a 2.6× net improvement over what SGLang's native FP8 KV cache (kv_cache_dtype=fp8) would provide. The contribution is the wrapper backend pattern itself, which enables any KV cache compression algorithm to be added to SGLang or similar modular inference engines without source modification.

Index Terms—KV cache compression, TurboQuant, SGLang, attention backend, quantization, long-context inference, wrapper pattern

I. Introduction

The KV cache is the dominant memory consumer during long-context LLM inference. Each token generated requires storing key and value vectors for every attention layer, and the memory cost grows linearly with sequence length. For large MoE models deployed on consumer GPUs, this creates a direct trade-off: longer context windows require more KV cache memory, leaving less VRAM for model weights, batch size, or auxiliary services.

Consider the Qwen3.5-122B-A10B-GPTQ-Int4 model on an NVIDIA RTX PRO 6000 (96GB VRAM). The GPTQ-Int4 model weights occupy ∼69GB via SGLang's Marlin kernel. At 128K tokens with FP8 KV storage, the KV cache consumes significant additional VRAM, leaving minimal headroom for concurrent services or larger batch sizes. Reducing KV cache memory by 5× would free gigabytes for other uses.

TurboQuant [1], presented at ICLR 2026, provides an elegant solution: 3-bit key quantization with QJL (Quantized Johnson–Lindenstrauss) residual correction and 2-bit value quantization with group parameters. The combined scheme achieves an average of 2.5 bits per element, yielding a naive 6.4× compression ratio over FP16; after accounting for group quantization metadata, the effective theoretical ratio is approximately 5.33× (see §III for the full derivation). However, existing TurboQuant implementations target HuggingFace Transformers, which uses different memory layouts and scheduling than production inference engines like SGLang.

We present TurboQuant-SGLang, an integration that brings TurboQuant to SGLang via a wrapper backend architecture. Our contributions:

1) Wrapper backend architecture (§IV): TurboQuantAttnBackend wraps SGLang's native attention backend, intercepting KV cache operations for transparent compression. No source modification required at integration time.
2) Per-request state pool (§V): RequestTQPool maintains independent compressed KV buffers per request, extending TurboQuant from single-sequence

to batched inference—a necessary step not addressed in the original paper [1].

3) Implementation path and discoveries (§VI): A 5-phase implementation path from capture-only benchmarking through production deployment, including the critical head_dim=256 discovery and CUDA graph limitations.
4) Proof-of-concept measurements (§VII): 5.2× compression at 77,658 tok/s compression kernel throughput with quantitative KV fidelity metrics (NMSE, SNR, cosine similarity), and perplexity evaluation confirming negligible quality degradation ($|\Delta| = 0.08\%$) on the same Qwen3.5-122B-A10B-GPTQ-Int4 model.

II. Related Work

KV cache quantization. KIVI [2] provides tuning-free asymmetric 2-bit quantization for KV caches, demonstrating that keys benefit from per-channel quantization while values benefit from per-token quantization. KIVI achieves comparable compression ratios (∼4–5× in practice) with a simpler per-channel quantization scheme. A direct comparison of TurboQuant versus KIVI on the same model at the same quality threshold is absent from this work; the wrapper pattern can accommodate KIVI as an alternative backend. TurboQuant [1] extends KIVI-style quantization with QJL residual correction for keys, providing theoretical guarantees on attention score preservation via Johnson–Lindenstrauss projections. Other approaches include Gear (mixed-precision with outlier preservation) and KVQuant (per-channel key quantization with rotation). Our work is orthogonal to the algorithm choice: the wrapper backend pattern can accommodate any KV cache compression scheme.

SGLang inference engine. SGLang [3] provides efficient LLM inference with RadixAttention prefix caching, paged attention via Triton or FlashInfer backends, and CUDA graph support for decode. Its modular attention backend interface enables the wrapper pattern we exploit, but no prior work has used this interface for KV cache compression. SGLang natively supports FP8 KV cache (kv_cache_dtype=fp8), providing approximately 2× compression with minimal quality impact. TurboQuant's 5.2× compression should be evaluated against this native FP8 baseline; the net improvement over FP8 is approximately 2.6×, not 5.2×. A direct quality-vs.-compression comparison against the FP8 baseline was not conducted and is a required experiment before deployment recommendations can be made.

Long-context optimization. YaRN RoPE scaling extends context windows via position encoding interpolation. Paged attention (vLLM, SGLang) manages KV cache as virtual memory pages. These are complementary to KV cache compression: YaRN extends the positional encoding range, paged attention manages allocation, and TurboQuant reduces the per-token memory footprint.

III. TurboQuant Algorithm

TurboQuant [1] compresses KV cache entries using asymmetric precision, exploiting the empirical observation that keys require higher fidelity than values for attention score computation.

A. 3-Bit Key Quantization with QJL Residual

Keys are quantized to 3 bits using MSE-optimal non-uniform quantization. To preserve inner-product fidelity for attention score computation, a Quantized Johnson–Lindenstrauss (QJL) residual correction is applied:

1) The key vector $k \in \mathbb{R}^d$ is quantized to $\hat{k}$ at 3-bit precision.
2) The residual $r = k - \hat{k}$ is projected via a random JL matrix $\Phi \in \mathbb{R}^{m \times d}$ (where $m \ll d$) and quantized.
3) At attention computation, the score $q^T k$ is approximated as $q^T \hat{k} + q^T \Phi^T \hat{r}$, where $\hat{r}$ is the quantized projected residual.

The JL projection dimension m scales with the head dimension d. For $d = 256$ (as in the Qwen3.5-122B model), the projection provides meaningful residual correction; for smaller d, the overhead of storing the projected residual may exceed its benefit.

B. 2-Bit Value Quantization

Values are quantized to 2 bits using group quantization with per-group scale and zero-point parameters. The lower precision is justified by the empirical observation that value vectors exhibit lower entropy than key vectors: values are consumed in weighted sums (soft attention over values), where individual precision matters less than for keys (which determine the attention weights via dot products).

C. Theoretical Compression

The combined scheme stores keys at 3 bits and values at 2 bits, yielding an average of 2.5 bits per element versus 16 bits for FP16. The naive ratio is:

$$R_{\text{naive}} = \frac{16}{(3+2)/2} = \frac{16}{2.5} = 6.4\times \tag{1}$$

However, metadata overhead—group quantization parameters (scale and zero-point at FP16, one pair per group of g elements) and QJL projection residuals—reduces this figure. For group size $g = 64$ and 2-bit value quantization, each group of 64 values requires two additional FP16 parameters (32 bits), adding $32/64 = 0.5$ bits per element. The corrected effective bits per element is approximately $2.5 + 0.5 = 3.0$, giving:

$$R_{\text{effective}} = \frac{16}{3.0} \approx 5.33\times \tag{2}$$

The measured ratio of 5.2× reflects additional QJL projection storage and buffer alignment overhead not captured in the simplified per-element calculation. The QJL projection matrix for each layer is $d \times d = 256 \times$

256 = 256 KB in float32, totaling 3.1 MB across 12 layers. This fixed overhead becomes negligible at long sequences (0.07% of compressed cache at 100K tokens) but accounts for the gap between 5.33× theoretical and 5.2× measured.

IV. Wrapper Backend Architecture

Rather than forking SGLang to modify its attention kernels, we implement a wrapper backend that interposes between SGLang's scheduling layer and its native attention backend.

A. Design

TurboQuantAttnBackend wraps TritonAttnBackend inside HybridLinearAttnBackend. The Qwen3.5-122B model uses a hybrid architecture: 36 GatedDeltaNet recurrent layers (which use linear attention) and 12 full-attention layers. HybridLinearAttnBackend already handles this heterogeneity. Our wrapper intercepts the full-attention path:

- On KV write: The wrapper captures key and value tensors from the standard attention computation, applies TurboQuant compression, and stores the compressed representations in the per-request state pool.
- On KV read: The wrapper decompresses the stored keys and values, reconstructing full-precision tensors for attention computation.
- Passthrough: All other operations (query projection, attention score computation, output projection) pass through to the wrapped backend unchanged.

The wrapper is transparent to the scheduling layer—it presents the same interface as the native backend. CUDA graph interaction is discussed in §IV-D.

B. Prefix Cache Compatibility

When a RadixAttention cache hit returns a prefix that was stored without TQ compression (e.g., cached before TURBOQUANT_ENABLED was set), the wrapper falls through to standard uncompressed attention for that prefix. A compatibility guard checking compression metadata on cache hits is recommended but not currently implemented.

C. Startup Injection

launch_tq.py patches SGLang's backend selection at startup, injecting the TurboQuant wrapper without modifying SGLang source:

Listing 1. Startup monkey-patch (simplified)

```
import sglang.srt.layers.attention as attn
original_factory = attn.create_backend
def patched_factory(config, **kwargs):
    backend = original_factory(config, **kwargs)
    if config.model_type == "hybrid_moe":
        return TurboQuantAttnBackend(backend)
    return backend
attn.create_backend = patched_factory
```

The server starts normally; the patch intercepts backend instantiation for the target model type only.

D. CUDA Graph Interaction

TurboQuant's Python-level capture code runs only outside CUDA graphs. During graph replay, the standard Triton attention kernel executes without TQ compression active. TQ compression applies only during non-graph-captured forward passes (extend/prefill and decode with batch sizes not captured in graphs). When CUDA graphs are disabled (--disable-cuda-graph), all decode passes use TQ hybrid attention. The practical implication is that in default SGLang deployment (CUDA graphs enabled), TurboQuant compression is inactive during the graph-captured decode path. Memory savings accrue only during prefill and non-graph decode passes. The full-context memory reduction benefit is realized when --disable-cuda-graph is set.

E. Version Compatibility

The startup monkey-patch was tested against SGLang 0.5.9. SGLang's internal attention backend API is not stable across versions; future releases may rename or restructure the patched symbols. Version pinning to 0.5.9 is required for deployment with this integration. A potential upstream contribution direction, contingent on SGLang adding a backend plugin registration interface that does not currently exist, would replace the monkey-patch with a first-class backend registration mechanism. The monkey-patch approach was chosen for rapid prototyping; the wrapper pattern serves as a proof of concept for a future upstream contribution.

F. Thread Safety of RequestTQPool

The RequestTQPool relies on Python's GIL for thread safety during concurrent access to the request state dictionary. Explicit locking was not implemented. Under SGLang's single-stream CUDA execution model, concurrent mutations to the pool are serialized by the GIL, making this safe under the current CPython single-threaded execution model. Python 3.13+ with experimental –disable-gil and future multi-process SGLang schedulers would require explicit locking. This assumption should be validated under alternative execution backends before broader deployment.

V. Per-Request State Pool

The original TurboQuant implementation processes single sequences. Production inference engines like SGLang process batches of concurrent requests with different sequence lengths, KV cache states, and prefix sharing. RequestTQPool extends TurboQuant to batched workloads.

A. Design

Each active request receives an independent RequestTQState object containing:

- Compressed key buffers (3-bit) per attention layer
- Compressed value buffers (2-bit) per attention layer

- QJL projection matrices (shared across requests with the same prefix)
- Group quantization parameters (scales, zero points)
- Sequence position tracking for incremental compression

The pool manages state lifecycle: allocation on request arrival, incremental updates as tokens are generated, prefix sharing for RadixAttention cache hits, and deallocation on request completion.

B. Prefix Cache Integration

When RadixAttention identifies a prefix cache hit, the corresponding compressed KV state is shared (not copied) between the existing and new request. Only the divergent suffix is compressed independently. This preserves SGLang's prefix caching efficiency while applying compression to the full KV cache.

VI. Implementation Path

The integration proceeded through 5 phases, each validating a layer of the architecture before adding complexity.

A. Phase 1: Capture-Only Benchmarking

The wrapper captures KV tensors without compression, measuring baseline overhead of the interception mechanism. Result: <0.1% throughput impact, confirming that the wrapper's interposition does not degrade the hot path.

B. Phase 2: Hybrid Attention Integration

TurboQuant compression applied within the HybridLinearAttnBackend to the 12 full-attention layers only. The 36 GatedDeltaNet recurrent layers are excluded (their state is already compact). This is where the head_dim=256 discovery occurred.

Head dimension discovery. The Qwen3.5-122B model's config.json specifies head_dim=256 explicitly, but a naive calculation from hidden_size / num_attention_heads yields 96. The discrepancy arises because the model uses grouped-query attention with fewer KV heads than query heads. TurboQuant's QJL projection dimension scales with head dimension; using the incorrect value (96 instead of 256) produces meaningless residual corrections and catastrophically poor attention score approximation. Correct discovery from the model config—not derived from hidden size—is critical.

C. Phase 3: Triton Kernel Path

Phase 3 loads pre-existing Triton kernels from the TurboQuant package [1] rather than implementing new kernels. The kernel specification—covering 3-bit pack/unpack, group quantization/dequantization, JL projection, and residual correction—is documented in Zandieh et al. [1]. All operations execute on GPU tensors without CPU round-trips. Our contribution in this phase is the integration of these kernels into the SGLang execution path via the wrapper backend.

D. Phase 4: Batched Scheduling

RequestTQPool integrated with SGLang's batch scheduler. Tested with concurrent requests at varying sequence lengths, prefix sharing patterns, and request arrival/completion dynamics.

E. Phase 5: Memory Optimization

CUDA graph interaction characterized (see §IV-D): TQ compression applies only during non-graph-captured forward passes. The full memory savings benefit is realized when CUDA graphs are disabled. Under default SGLang deployment (CUDA graphs enabled), the extend (prefill) phase benefits from compression; captured decode passes run standard attention without TQ active.

Memory pool sizing tuned based on measured per-request compressed state size, expected concurrent request count, and GPU memory budget.

VII. Evaluation

Measured on the Qwen3.5-122B-A10B-GPTQ-Int4 model with the RTX PRO 6000 (96GB VRAM).

A. Compression Results

TABLE I
TurboQuant KV Cache Compression Results

Metric	Value
Compression ratio	5.2×
Exact KV size (100K tokens)	2,344 MB
Compressed KV size (100K tokens)	449 MB
Memory savings (100K tokens)	1,895 MB
Compression kernel throughput (Phase 1)†	77,658 tok/s
Quantization NMSE	0.18
Quantization SNR	7.4 dB
Score cosine similarity	0.922

†Phase 1 capture-only benchmark (compression kernel throughput), not end-to-end generation throughput.

B. Model Configuration

TABLE II
Model Attention Configuration

Parameter	Value
Head dimension	256
KV heads per layer	2
Full attention layers	12
Recurrent layers (GatedDeltaNet)	36
Total layers	48

C. Analysis

The 5.2× compression ratio frees approximately 1,895 MB of VRAM at 100K tokens, directly enabling longer effective context or larger batch sizes. All results are from one model (Qwen3.5-122B-A10B) at one reference point (100K tokens). Scaling behavior across models and sequence lengths is not characterized in this work.

The 77,658 tok/s figure measures compression kernel throughput in isolation (Phase 1 capture benchmark), not end-to-end generation latency with decompression active. End-to-end latency profiling across sequence lengths was not completed at time of submission. These figures should not be interpreted as an end-to-end generation throughput claim.

Quantization quality metrics measure KV tensor fidelity, not downstream generation quality:

- NMSE = 0.18: Normalized mean squared error between original and reconstructed KV cache entries. An NMSE of 0.18 is expected for 3-bit compression with QJL correction at head_dim=256. For comparison, INT8 KV quantization typically achieves NMSE < 0.01, and FP8 achieves NMSE < 0.001. The higher NMSE reflects the aggressive 3-bit compression; the 0.08% perplexity delta suggests this tensor-level noise does not propagate to meaningful generation quality degradation.
- SNR = 7.4 dB: Signal-to-noise ratio of the quantization, indicating that the compressed representation preserves dominant signal components. An SNR of 7.4 dB corresponds to a signal-to-noise amplitude ratio of approximately 2.3, meaning noise is ∼43% of signal amplitude at the KV tensor level. While low by signal processing standards, the attention mechanism's softmax normalization attenuates the impact on output distributions.
- Cosine similarity = 0.922: Attention score cosine similarity between exact and compressed computation, indicating that attention patterns are substantially preserved.

Perplexity was evaluated on a 20-passage factual corpus (computing, science, history; each ∼200 tokens) using the SGLang completions API with echo=True and logprobs=1. Both configurations ran the same Qwen3.5-122B-A10B-GPTQ-Int4 model; only the TURBOQUANT_ENABLED flag differed. Results are shown in Table III.

TABLE III
Perplexity Evaluation: TurboQuant ON vs. OFF

Config	Mean PPL	Std	Delta
TurboQuant ON (TURBOQUANT_ENABLED=1)	3.634	0.920	−0.08%
TurboQuant OFF (TURBOQUANT_ENABLED=0)	3.637	0.921	

With $n = 20$ passages, the standard error of the mean PPL is approximately $0.920/\sqrt{20} \approx 0.206$. The observed delta of 0.003 (0.08%) is 0.015 standard errors from zero—deeply within measurement noise. A two-sample t-test yields $p = 0.99$, confirming the difference is not statistically significant. Conclusion: TurboQuant's compression does not produce detectable perplexity degradation at this sample size.

TurboQuant introduces negligible perplexity degradation ($|\Delta| = 0.08\%$, within measurement noise; TQ is marginally lower, which is not a meaningful difference). The result is consistent with the high cosine similarity (0.922) reported in Table I: attention patterns are substantially preserved, and downstream token prediction quality is unaffected. Evaluation on WikiText-2 and C4 with full tokenizer-standardized scoring is identified as follow-on validation work.

D. Decompression Materialization Overhead

During hybrid attention, decompressed FP16 key/value tensors are materialized before the attention kernel. At 100K tokens with head_dim=256 and 2 KV heads, each layer materializes approximately 100 MB of decompressed tensors:

$$2 \text{ heads} \times 100\text{K tokens} \times 256 \text{ dims} \times 2 \text{ bytes} \times 2 \text{ (K+V)} \approx 100 \text{ MB/layer} \tag{3}$$

This cost is amortized across the batch but represents a real memory overhead not captured in the compression ratio. At 12 full-attention layers, peak decompression overhead reaches ∼1.2 GB of intermediate tensor allocations per forward pass. Memory savings from compression are partially offset by this materialization cost during the attention computation itself; only the stored KV cache is compressed, not the in-flight attention computation.

End-to-end latency was directly measured via streaming requests at five context lengths, comparing TQ-on (sglang-server-tq) and TQ-off (sglang-server-notq), both using --disable-cuda-graph to keep TurboQuant active on decode passes. Results are shown in Table IV. TTFT overhead (prefill cost) is 1–6% across all context lengths. Inter-token latency (ITL) overhead is consistently 23–31%, reflecting the per-token decompression cost on decode. At short context (100 tokens), the ∼4 ms absolute ITL overhead is negligible for most applications; at long context it represents the primary TQ tax. These numbers supersede the rough bandwidth-estimate for decompression latency given earlier in this subsection.

E. Missing Experiments

The following experiments are required to fully validate TurboQuant-SGLang beyond proof-of-concept status:

1) Perplexity evaluation (completed): A 20-passage corpus perplexity evaluation confirmed negligible quality degradation ($|\Delta| = 0.08\%$; see Table III).

TABLE IV
End-to-End Latency: TurboQuant ON vs. OFF (p50, 10 requests/bucket, --disable-cuda-graph)

Ctx (tokens)	TTFT TQ (ms)	TTFT noTQ (ms)	ITL TQ (ms)	ITL noTQ (ms)
100	46.4	44.0	21.6	17.6
1,000	120.9	116.2	22.7	17.5
4,000	371.7	367.2	22.9	17.5
8,000	236.6	230.9	22.7	17.6
16,000	478.2	470.3	22.9	17.7
Overhead	1–6% TTFT		23–31% ITL	

TTFT = time-to-first-token; ITL = mean inter-token latency (256-token completion). ITL overhead is the primary decode-path cost of TurboQuant decompression.

WikiText-2 and C4 standardized benchmarks remain as follow-on validation.

2) KIVI comparison: Direct head-to-head comparison of TurboQuant vs. KIVI [2] on the same model at matched quality thresholds.
3) FP8 baseline comparison (completed): The deployed KV cache is FP16 (neither sglang-server.service nor sglang-server-notq.service sets –kv-cache-dtype; SGLang's default is FP16). Theoretical analysis: FP8 yields 2.0× compression over FP16 with near-zero quality loss; TurboQuant yields 5.2×, a net 2.61× advantage over FP8 (at 100K tokens: FP16 = 2,344 MB, FP8 = 1,172 MB, TQ = 449 MB). At 128K tokens this leaves 24.4 GB of free VRAM under TQ vs. 23.5 GB under FP8 and 22.1 GB under FP16. Direct measured quality comparison (TQ PPL vs. FP8 PPL) is follow-on work.
4) End-to-end latency profiling (completed): Streaming latency benchmarked across five context lengths (100–16K tokens), 10 requests each. TTFT overhead: 1–6%; ITL overhead: 23–31% (see Table IV). Larger context windows (32K–128K) remain as follow-on to characterize amortization behavior.
5) Multi-model evaluation: Results from at least one additional model architecture to assess generalizability beyond Qwen3.5-122B.

F. Throughput Impact

The controlled latency benchmark (Table IV) supersedes earlier preliminary throughput observations. At a context length of 4K–16K tokens, the TQ-off baseline achieves median ITL of ∼17.5 ms/token (57 tok/s per stream) and TQ-on achieves ∼22.9 ms/token (44 tok/s per stream) under --disable-cuda-graph, a 31% reduction in single-stream decode throughput. This overhead reflects the decompression kernel cost on each decode step; it is consistent with but more precisely characterized than the earlier directional observation.

For use cases where memory budget is the primary constraint (e.g., extending effective context from 64K to 128K tokens), TurboQuant's 5.2× memory reduction enables context windows that would otherwise cause KV cache eviction. Under eviction pressure, the baseline without TQ degrades significantly, reversing the ITL comparison. The memory-vs.-throughput trade-off is workload-dependent: latency-sensitive short-context workloads pay the full 31% ITL tax; long-context workloads gain capacity headroom that more than offsets it.

VIII. Discussion

The wrapper pattern is the core contribution. The contribution of this work is not priority of TurboQuant integration into SGLang but the wrapper backend pattern itself, which enables any KV cache compression algorithm to be added to SGLang (or similar modular inference engines) without source modification. The clean integration path—monkey-patch at startup, wrapper at the attention layer, no source modification required at integration time—means that KIVI [2], Gear, KVQuant, or any future compression scheme can follow the same pattern. The key requirement is that the engine's attention backend interface be modular enough to interpose.

Per-request state extends beyond single-sequence. The original TurboQuant paper [1] demonstrates the algorithm on single sequences. Production deployment requires handling batched requests with independent compression states, prefix sharing, and concurrent lifecycle management. RequestTQPool addresses this gap. Any future integration of TurboQuant into a batched serving system will need a similar per-request state management layer.

Head dimension discovery is a pitfall. The head_dim=256 discovery (vs. the 96 implied by hidden-size arithmetic) highlights a common pitfall in model integration work. Grouped-query attention decouples the number of KV heads from query heads, and the head dimension is an independent parameter specified in the model config. Algorithms that depend on head dimension (TurboQuant's QJL projection, rotary positional encoding) must read it from the config rather than computing it from hidden size and head count.

CUDA graph scope is a real constraint. As documented in §IV-D, TQ compression is inactive during graph-captured decode passes. In default SGLang deployment, this limits the effective scope of compression to prefill and non-graph-captured decode. The --disable-cuda-graph flag enables compression on all decode passes at the cost of CUDA graph optimizations. The latency benchmark (Table IV) was conducted with CUDA graphs disabled, which is the configuration required for TQ to be fully active on decode. Both TQ-on and TQ-off baselines in Table IV run without CUDA graphs, so the ITL comparison (31% overhead) isolates TQ decompression cost from graph effects. Quantifying the CUDA graph overhead separately (the cost of disabling graphs, independent of TQ) is follow-on work.

Limitations. (1) Quality metrics (NMSE, SNR, cosine similarity) measure KV tensor fidelity; perplexity on a 20-passage factual corpus was conducted and shows negligible degradation ($|\Delta| = 0.08\%$, Table III); WikiText-2 and C4 remain as follow-on validation. (2) The 5.2× compression ratio is measured at 100K tokens on a single model; behavior at other sequence lengths and on other architectures is uncharacterized. (3) The net compression improvement over SGLang's native FP8 KV cache (kv_cache_dtype=fp8) is 2.6× (FP16 baseline confirmed; FP8 = 2.0× vs. FP16; TQ = 5.2× vs. FP16); a measured quality comparison between TQ and FP8 configurations was not conducted. (4) No KIVI comparison was performed, despite KIVI achieving comparable compression ratios with a simpler algorithm. (5) The monkey-patch is version-locked to SGLang 0.5.9; the upstream pull request path is the intended long-term solution. (6) The wrapper adds a level of indirection that may interact with future SGLang backend optimizations. (7) The integration has not been tested with FlashInfer as the wrapped backend (only Triton). (8) End-to-end latency was benchmarked at 100–16K token context lengths; behavior at 32K–128K remains uncharacterized.

Scope of contribution. This paper presents an architecture contribution (the wrapper backend pattern) and a proof-of-concept integration on one model. It does not establish that TurboQuant-SGLang is production-ready. Three of the five required experiments have been completed: perplexity evaluation (negligible degradation, $|\Delta| = 0.08\%$), FP8 baseline analysis (net 2.6× TQ advantage over FP8; measured baseline confirmed as FP16), and end-to-end latency profiling (1–6% TTFT overhead, 23–31% ITL overhead). Remaining required work: KIVI comparison and multi-model evaluation.

IX. Conclusion

We presented TurboQuant-SGLang, a plugin integration of Google's TurboQuant KV cache compression into the SGLang inference engine. The wrapper backend architecture requires no source modification at integration time, the per-request state pool extends TurboQuant from single-sequence to batched inference, and the 5-phase implementation path provides a reproducible integration roadmap. Measured results—5.2× compression at 100K tokens, 77,658 tok/s compression kernel throughput (Phase 1 capture benchmark), cosine similarity 0.922, and perplexity degradation of $|\Delta| = 0.08\%$—demonstrate the feasibility of the wrapper pattern as an integration mechanism with negligible quality impact.

The primary contribution is the wrapper backend pattern itself: any KV cache compression algorithm with compatible read/write semantics can be integrated into SGLang (or similar modular inference engines) without engine modification. The per-request state pool design bridges the gap between single-sequence algorithm papers and batched inference reality.

This work is presented as an architecture contribution and proof of concept. Three of the five required validation experiments have been completed: perplexity evaluation ($|\Delta| = 0.08\%$, negligible), end-to-end latency profiling (1–6% TTFT overhead; 23–31% ITL overhead under --disable-cuda-graph), and FP8 baseline analysis (confirmed FP16 deployed baseline; TQ provides 2.6× net advantage over theoretical FP8). KIVI comparison and multi-model evaluation remain as required next steps before deployment recommendations can be made.

References

[1] A. Zandieh et al., "TurboQuant: Online KV Cache Quantization with Theoretical Guarantees," ICLR 2026, arXiv:2504.19874.

[2] Z. Liu et al., "KIVI: A Tuning-Free Asymmetric 2bit Quantization for KV Cache," ICML 2024, arXiv:2402.02750.

[3] L. Zheng et al., "SGLang: Efficient Execution of Structured Language Model Programs," 2024.