FServe: FP8-Native W4A8KV4 Quantization and Attention Kernel Optimization for Efficient LLM Serving

Daehyeon Baek*, **Jieun Choi***, **Jimyoung Son***, **Jae-Young Choi**, **Kyungmin Bin**, **Seungbeom Choi**, **Kihyo Moon**, **Minsung Jang**, **Hyojung Lee**†
Cloud Research Team, Samsung SDS

Abstract

As large language models (LLMs) scale, inference throughput is increasingly bottlenecked by memory bandwidth. While prior 4-bit quantization schemes promise memory savings, their dequantization overhead often negates these gains. We present FServe, a hardware-software co-designed framework featuring an FP8-native W4A8KV4 scheme that combines channel pre-scaling and group-wise scale relaxation for linear layers with post-RoPE key smoothing that aligns channel-wise means and ranges for attention layers. To translate these memory savings into FP8-native throughput, FServe further extends FlashAttention-3 kernels for prefill with packed-FP16 softmax instruction reordering and multi-warp asynchronous TMA fetching, and introduces a dynamic Stream-KV tile scheduler for decode phase workload imbalance. Our evaluation on Llama-3.1-8B and Qwen3-8B shows that FServe outperforms AWQ W4A16 and W4A8 baselines on accuracy despite additionally compressing the KV cache to INT4, achieves up to 1.41× TTFT speedup over BF16, and sustains effective decode bandwidth above the H100's theoretical peak.

1 Introduction

As large language models (LLMs) continue to scale in parameter count and input sequence lengths, inference throughput is increasingly bottlenecked by memory bandwidth (Lee et al., 2024; Gholami et al., 2024), motivating efficient post-training quantization (PTQ). While 8-bit formats like FP8 leverage hardware-native acceleration, the persistent memory wall in large-scale deployments demands sub-8-bit schemes such as INT4. For 4-bit quantization, however, the dequantization overhead often outweighs the memory bandwidth gains, creating a processing bottleneck that negates the throughput benefits of 4-bit representation (Kim et al., 2023; Frantar et al., 2025; Wilkinson, 2024).

To address these challenges, we present FServe, a hardware-software co-designed framework featuring a W4A8KV4 (INT4 weights, FP8 activations, INT4 KV cache) mixed-precision inference scheme. FServe combines an accuracy-preserving quantization strategy with FlashAttention-3 (FA3) based (Shah et al., 2024) attention kernels that sustain hardware-native throughput in prefill and balance variable-length workloads in decode. We adopt W4A8KV4 rather than W4A4 since sub-byte integer arithmetic cannot exploit Hopper's asynchronous FP8 execution path (Appendix A).

For linear layers, FServe applies a two-step W4A8 quantization strategy with FP8 group scales. BF16 pre-scaling along the input channel dimension reshapes the weight distribution to fit the dynamic range of FP8 group scales, preventing scale underflow, and a group-wise relaxation factor recovers the underused -8 bin of the asymmetric INT4 range. For the attention layer, FServe applies a post-RoPE calibration-based key smoothing that combines shrunk centering with channel-wise range alignment, preserving the softmax-normalized attention distribution while improving INT4 key fidelity.

In addition, FServe modifies FlashAttention-3 (FA3) kernels for both prefill and decode phases. For prefill, strategic instruction reordering accelerates packed-FP16 softmax by eliminating redundant integer permutation instructions, and multi-warp asynchronous tensor memory accelerator (TMA) fetching with proactive prefetching prevents memory starvation under sub-8-bit dequantization. For decode, FServe adapts the kernel to FServe-specific quantization and introduces a dynamic Stream-KV tile scheduler. This scheduler distributes variable-length workloads evenly across streaming multiprocessors (SMs) without an additional combine kernel and adaptively toggles between deterministic and non-deterministic reduction modes based on the ratio of KV heads to the

*The first three authors contribute equally and are listed alphabetically. †Corresponding author.

Findings of the Association for Computational Linguistics: EMNLP 2026, pages 33132–33153
October 24-29, 2026 ©2026 Association for Computational Linguistics

number of available SMs.

We evaluate FServe on Llama-3.1-8B-Instruct and Qwen3-8B integrated into vLLM. FServe outperforms the AWQ W4A16 and W4A8 baselines on both Wikitext-2 perplexity and average downstream benchmark score, despite additionally compressing the KV cache to INT4, staying close to the BF16 and FP8 reference quality. FServe's kernels achieve up to 1.13× the FP8 prefill attention throughput of FA3 (1,233 vs 1,088 TFLOPS) at head dimension 256, sustain effective decode bandwidth above the H100's theoretical memory bandwidth of 3.35 TB/s, and deliver up to 1.41× speedup over BF16 on long context cases in these models.

Our contributions are summarized as follows. (i) Unlike AWQ and QServe, which assume high-precision scales, FServe quantizes weights directly under FP8 group scales, where BF16 pre-scaling prevents scale underflow and a group-wise relaxation factor recovers the unused -8 bin. (ii) Unlike QServe's pre-RoPE pair-shared scaling and SageAttention's online full-mean removal, FServe applies offline-calibrated partial centering and per-channel range alignment after RoPE, preserving the attention distribution. (iii) Beyond FA3, FServe adds packed-FP16 softmax reordering, multi-warp TMA prefetching, and a Stream-KV decode scheduler that eliminates the separate combine kernel of split-KV methods. Appendix K expands on the novelty of each component relative to its closest prior work.

2 Related Work

2.1 Quantization Fidelity for Low-bit Serving

Existing low-bit PTQ methods preserve fidelity by reshaping the tensor distribution before quantization. SmoothQuant (Xiao et al., 2023), AWQ (Lin et al., 2024), QServe (Lin et al., 2025), and rotation-based methods such as QuaRot (Ashkboos et al., 2024) and SpinQuant (Liu et al., 2025) reduce outlier-induced quantization error, but do not directly target INT4 quantization with FP8 scaling factor, where FP8 scale underflow may cause accuracy degradation. FServe addresses this regime through a two-step scaling mechanism that exploits high-speed FP tensor cores while improving INT4 bin utilization, preserving accuracy.

For KV cache quantization, FServe builds on attention-specific key smoothing. Inspired by SageAttention (Zhang et al., 2025), FServe mitigates channel-wise key offsets through mean subtraction, but estimates channel-wise means during offline calibration to avoid heavy online reductions, and applies a calibrated fraction of the mean shift rather than full mean removal. FServe also differs from QServe (Lin et al., 2025) in its treatment of rotary positional embedding (RoPE), which is used on query-key pairs in modern LLMs. Since RoPE rotates key channels in pairs, pre-RoPE key scaling in QServe requires pair-wise shared scales, which restricts the granularity of smoothing. FServe instead applies smoothing after RoPE, which permits fine-grained outlier smoothing that improves INT4 key fidelity.

2.2 Workload Imbalance Problem for Variable Length Attention Decoding

With a larger LLM context length, attention decoding over the KV cache becomes a major latency bottleneck. When the number of KV heads is fewer than the number of available SMs, the resulting workload imbalance significantly reduces GPU utilization (Agrawal et al., 2024). While FlashDecoding (Dao et al., 2023) mitigates this by partitioning the KV cache along the sequence dimension and distributing computation across SMs, it introduces the overhead of an additional combine kernel to merge partial results. Moreover, in scenarios involving batched requests with heterogeneous sequence lengths, determining optimal split points to ensure balanced workload distribution remains a non-trivial challenge (Zhong et al., 2024).

Linear layers with a small number of tokens face similar workload imbalance challenges when using tile-based distribution(Chen et al., 2024). Typically, general matrix multiplication (GEMM) kernels on GPUs employ a data-parallel strategy, distributing the workload in output-tile units. However, when the number of output tiles is smaller than the available SMs, significant resource underutilization occurs as many SMs remain idle. While the Split-K technique mitigates this by partitioning the K-reduction dimension, it requires the K-dimension to be a multiple of the SM count for optimal efficiency, which is rarely the case in practice.

To address this, Stream-K (Osama et al., 2023) employs a work-centric decomposition that partitions output tiles across the K-reduction dimension, utilizing arrive-and-wait barriers to synchronize partial results. Stream-K relies on a fixed K-dimensional space to determine the number of partitions and the scheduling strategy for each SM

via simple integer arithmetic. However, this deterministic approach is not directly applicable to variable-length attention decoding. In such scenarios, the number of iterations in the KV dimension varies per output tile, rendering fixed-split scheduling ineffective. Furthermore, because sequence lengths are dynamic, it becomes fundamentally challenging to pre-determine the capacity of reduction buffers required to aggregate partial results, necessitating a more flexible and adaptive scheduling paradigm (Hong et al., 2024).

FServe addresses these challenges with a dynamic Stream-KV tile scheduler that improves decode phase performance, described in Section 4.2.

3 FServe's W4A8KV4 PTQ Method

3.1 W4A8 Quantization for Linear Layer

The weights of linear layers are quantized to INT4 with group-wise quantization scales stored in FP8. To reconcile its limited resolution, we decompose weight quantization into two sequential steps. The first step applies channel-wise BF16 pre-scaling along the input channel direction (d_{in}), thereby reshaping each input channel to a range representable by the subsequent FP8 group scales, mitigating underflow. The second step performs group-wise INT4 quantization along the input channel direction (d_{in}), combining the activation-aware exponent search of AWQ (Lin et al., 2024) with a newly introduced group-wise relaxation factor that recovers underutilized regions of the INT4 dynamic range.

Let $\mathbf{W} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ denote the full-precision weight matrix of a linear layer, and let $\mathbf{X}$ denote the corresponding calibration activations. Denote by $\mathbf{r} = (r_1, \ldots, r_{d_{\text{in}}})$ the per-input-channel activation magnitudes estimated from $\mathbf{X}$. Following AWQ, we introduce an activation-aware diagonal scaling for a candidate exponent $\alpha \in [0, 1]$,

$$\mathbf{W}_\alpha = \mathbf{W}\text{diag}(\mathbf{r}^\alpha),$$

where $\mathbf{r}^\alpha$ denotes element-wise exponentiation. This redistributes quantization difficulty from high-activation input channels toward the weights.

Step 1: BF16 pre-scaling. Let $\mathbf{c} = (c_1, \ldots, c_{d_{\text{in}}})$ be a per-input-channel BF16 scaling vector. We obtain the pre-scaled weight

$$\mathbf{W}'_\alpha = \mathbf{W}_\alpha \text{diag}(\mathbf{c}),$$

which is then passed to Step 2 and ultimately stored in INT4 with FP8 group scales. Because $\text{diag}(\mathbf{c})$ acts on entire input rows, the inverse scaling can be absorbed into the BF16 accumulator at the GEMM epilogue at negligible runtime cost, preserving the hardware-native W4A8 execution path. The per-channel factor c_k is selected so that each row of $\mathbf{W}'$ occupies a range representable by the subsequent FP8 group scales, mitigating underflow.

Step 2: Group-wise INT4 quantization with relaxed activation-aware scales. Group-wise INT4 quantization of $\mathbf{W}'_\alpha$ exposes a structural inefficiency. Let $\mathbf{w}_g$ collect the entries of $\mathbf{W}'_\alpha$ assigned to group g, and let s_g denote its baseline quantization scale. Although INT4 uses symmetric quantization, the integer range $[-8, 7]$ is itself asymmetric. When s_g is set from $\max(|\mathbf{w}_g|)$, the normalized values $\mathbf{w}_g/s_g$ fall within $[-7, 7]$ and the -8 bin remains unreached. To recover this underused region, we relax each group scale s_g by a *relaxation factor* $\gamma_g \in [0, 1]$.

How $\gamma_g^\star$ and $\alpha^\star$ derived? Each γ_g shrinks s_g to a smaller scale $\gamma_g s_g$. A smaller scale reduces the quantization step size and yields finer mid-range resolution. Since $\mathbf{w}_g$ and s_g implicitly depend on α through $\mathbf{W}'_\alpha$, we fix α during the following group-wise search and suppress this dependence in the notation. Let $\mathcal{G}$ denote the set of quantization-group indices. We define the quantized-and-dequantized reconstruction operator for a group vector $\mathbf{z}$ with scale s as

$$\Pi_s(\mathbf{z}) := s \cdot \text{Quant}\left(\frac{\mathbf{z}}{s}\right).$$

For group-wise quantization, $\Pi_{\{s_g\}_{g\in\mathcal{G}}}(\cdot)$ denotes applying Π_{s_g} independently to each group g.

Given γ_g, the reconstructed group is

$$\widehat{\mathbf{w}}_g(\gamma_g) = \Pi_{\gamma_g s_g}(\mathbf{w}_g).$$

For each candidate α, the relaxation factor is selected independently per group by minimizing

$$\gamma_g^\star = \arg\min_{\gamma_g} \left\|\mathbf{w}_g - \widehat{\mathbf{w}}_g(\gamma_g)\right\|_2^2.$$

This search depends only on the rescaled weight values for a fixed α, so it can be efficiently computed in parallel across the relaxation factor and group dimensions.

The selected relaxation factors yield the relaxed group-wise scales

$$S_\alpha^\star = \{\gamma_g^\star s_g\}_{g\in\mathcal{G}},$$

from which we define the quantized-and-dequantized reconstruction of $\mathbf{W}'_\alpha$ as

$$\widehat{\mathbf{W}}_\alpha = \Pi_{S_\alpha^\star}(\mathbf{W}'_\alpha).$$

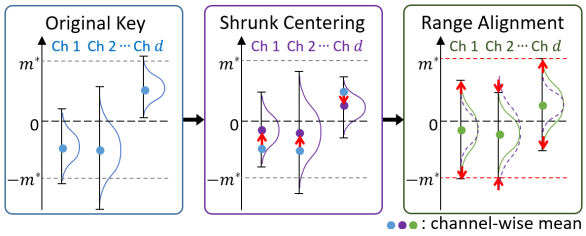


Figure 1: Calibration-based post-RoPE key smoothing for INT4 KV-cache quantization. Shrunk centering reduces mean offsets, and range alignment equalizes channel ranges.

The activation-aware exponent is then selected by minimizing the layer-local reconstruction error

$$\alpha^\star = \arg\min_{\alpha} \left\| \mathbf{X}\left(\mathbf{W}'^\top - \mathrm{diag}(\mathbf{r}^{-\alpha})\widehat{\mathbf{W}}_\alpha^\top\right)\right\|_F^2.$$

The per-group relaxation factors $\{\gamma_g^\star\}_{g\in\mathcal{G}}$ obtained under $\alpha^\star$ are adopted as the final calibration result.

3.2 KV4 Quantization for Attention Layer

FServe quantizes the attention layer's KV cache to INT4 (with FP8 queries) and applies calibration-based key smoothing to preserve attention fidelity.

KV4 quantization. KV cache quantization reduces memory traffic during long-context decoding. FServe stores both keys and values in INT4 with per-token group-wise scaling factors, whereas queries are quantized to FP8 on-the-fly. Values exhibit relatively uniform per-channel ranges and are quantized under this scheme without additional treatment. Keys, however, often exhibit inter-channel imbalance, causing a few channels with large means or wide ranges to dominate the group-wise quantization scale (Liu et al., 2024), leaving the remaining channels mapped to only a small subset of INT4 bins. FServe addresses this by applying calibration-based key smoothing before quantization.

Key outlier smoothing. The key smoothing is applied after RoPE, immediately before INT4 quantization, and combined *shrunk centering* with *range alignment* (Figure 1). Unlike pre-RoPE scaling, which must use a shared scale for each rotary pair, post-RoPE smoothing aligns key ranges independently for each channel.

For clarity, we omit layer and head indices. Let $\mathcal{C}$ denote the set of position indices of the calibration dataset, and for each $u \in \mathcal{C}$, let $\mathbf{k}_u \in \mathbb{R}^d$ be the corresponding calibration key. At inference time, $\mathbf{k}_i, \mathbf{q}_t \in \mathbb{R}^d$ denote the post-RoPE key at position i and query at position t, respectively. From $\mathcal{C}$, FServe precomputes the channel-wise key mean

$$\mu_j = \frac{1}{|\mathcal{C}|}\sum_{u\in\mathcal{C}} k_{u,j},$$

together with two scalar hyperparameters $\lambda^\star$ and $m^\star$ that control how aggressively the key distribution is centered (shrink centering) and rescaled (range alignment). The calibration procedure is described below. Given $\lambda^\star$ and $m^\star$, the per-channel range scale

$$\rho_j = \frac{1}{m^\star}\max_{u\in\mathcal{C}} |k_{u,j} - \lambda^\star \mu_j|$$

is set so that the shifted calibration distribution $\{k_{u,j} - \lambda^\star\mu_j\}_{u\in\mathcal{C}}$ aligns its range on $[-m^\star, m^\star]$.

How $\lambda^\star$ and $m^\star$ derived? The hyperparameters $\lambda^\star$ and $m^\star$ are calibrated to minimize the KL divergence (Wu et al., 2020) between attention distributions computed with and without FServe quantization. We sweep $\lambda \in [0, 1]$ in steps of 0.1, where $\lambda = 1$ corresponds to full mean removal that shifts each channel mean to zero (Figure 1, middle), and $m \in [7, 7.9]$ in step of 0.1, motivated by the INT4 range $[-8, 7]$.

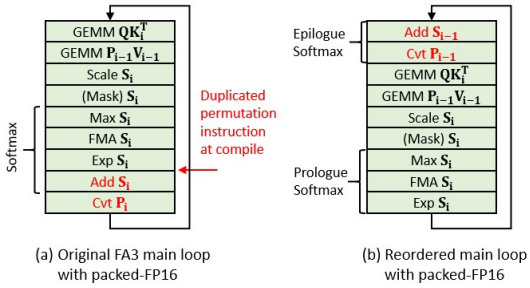
At inference time, each online key $\mathbf{k}_i$ is transformed component-wise as

$$\widetilde{k}_{i,j} = \frac{1}{\rho_j}(k_{i,j} - \lambda^\star\mu_j)$$

Choosing $\lambda^\star < 1$ removes only part of the calibration mean, avoiding over-centering when the offline calibration and online inference distributions diverge. FServe applies the inverse range scale to the post-RoPE query, $\widetilde{q}_{t,j} = q_{t,j}\rho_j$, thereby making the range-alignment dot product preserving. The centering-induced logit shift $\lambda^\star\mathbf{q}_t^\top$ is position-independent for a fixed query and is therefore canceled by the softmax normalization. Thus, FServe improves post-RoPE key quantization while preserving the pre-quantization attention distribution with only lightweight offline-calibrated affine transforms.

4 Attention Kernel Optimization

While the W4A8KV4 quantization scheme detailed in Section 3 reduces memory traffic, it introduces new micro-architectural bottlenecks in the compute pipeline (Wang et al., 2024). FServe implements a hardware-friendly optimizations across both prefill and decode phases, exploiting FP8 WGMMA execution paths to translate W4A8KV4 memory savings into actual throughput (Appendix A). The



(a) Original FA3 main loop with packed-FP16

(b) Reordered main loop with packed-FP16

Figure 2: Reordered main loop pipeline for optimization. The addition and FP16-to-FP8 conversion operation is moved to the beginning of the iteration loop to break the instruction dependency between the exponential operation.

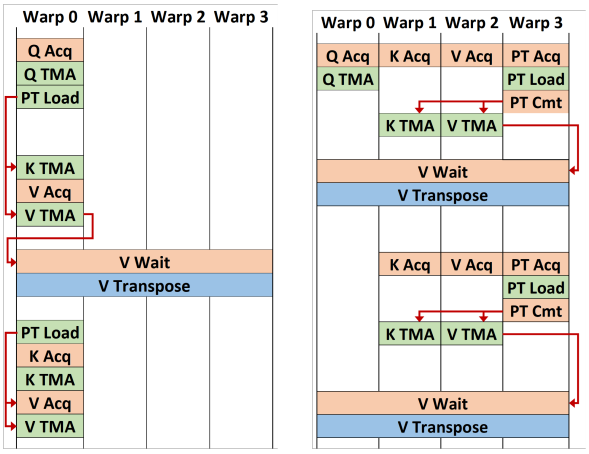
following sections detail attention kernel optimizations for prefill (Section 4.1) and a dynamic Stream-KV tile scheduler for decode (Section 4.2).

4.1 Prefill Phase Acceleration: Attention Kernel Optimization

We extend the original two-stage FlashAttention-3 (FA3) pipeline by introducing two critical architectural optimizations. We apply strategic instruction reordering for packed-FP16 softmax computations to mitigate integer pipeline stalls. We implement multi-warp asynchronous data fetching to prevent memory fetch stalls.

Reducing softmax compute precision and reordering instructions. To maximize floating-point compute throughput, we first reduce the precision of the accumulator output $\mathbf{S}_i = \mathbf{Q}\mathbf{K}_i^{\mathrm{T}}$ from FP32 to FP16. Because the accumulator $\mathbf{S}$ is eventually converted into the FP8 probability matrix $\mathbf{P}_i$, maintaining FP32 intermediate precision is fundamentally unnecessary. This targeted precision reduction safely enables the adoption of packed-FP16 instructions for the subsequent softmax operations. Utilizing these packed instructions significantly increases the overall utilization of the floating-point compute units. However, a naive packed-FP16 softmax implementation suffers from a severe integer ALU bottleneck. With careful profiling of the softmax pipeline, we decided to move the addition and FP16-to-FP8 conversion operations to the beginning of the KV iteration loop, reducing the number of integer operations, as shown in Figure 2. Detailed reasoning is described in Appendix B.

Multi-warp asynchronous TMA and prefetching. In the standard FlashAttention-3 architecture, the Tensor Memory Accelerator (TMA) is a component of the SM that asynchronously transfers data between global and shared memory. Because initiating TMA theoretically requires only a single thread, FA3 allocates an entire warpgroup (128 threads) strictly for data loading, while the remaining two warpgroups focus on computation. Within this loading warpgroup (Warps 0–3), FA3 delegates the entire responsibility of initiating TMA loads for the query, key, and value tensors to a single thread within Warp 0. Figure 3a shows the single-thread TMA initiation scheme of FA3. However, TMA relies on a producer-consumer paradigm requiring the initiating thread to acquire specific barrier locks before triggering any data transfer. The remaining 127 threads in the loading warpgroup are vastly underutilized, primarily performing minor tasks such as value-tile transposition to align with the hardware matrix-multiplication directions.



(a) Flash Attention 3

(b) FServe Framework

Figure 3: FlashAttention-3 uses only one thread for loading query, key, value, and page table entries. FServe framework distributes loading tasks to other warps, breaking the program order of barrier acquisition for parallelism. The actual implementation reorders barrier acquisition, TMA initiation, and value tile transposition to maximize throughput.

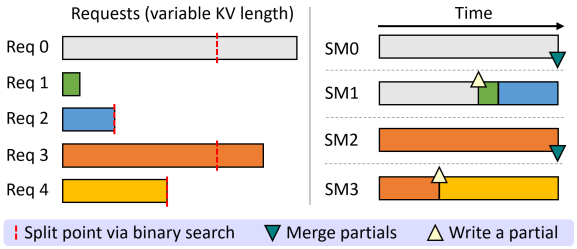
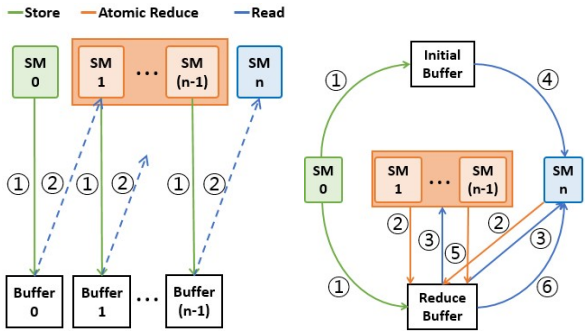


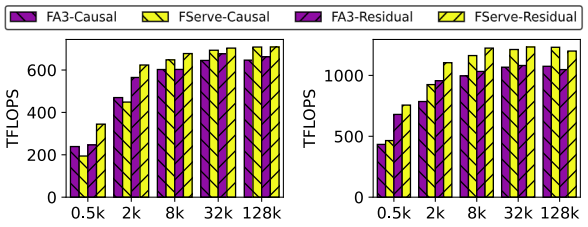
Figure 4: Stream-KV workload distribution on a variable-length decoding task. Each SM process has a similar number of KV sequences with a binary search and an asynchronous merging process.

For optimization, FServe distributes lock acquisition and TMA initiation across multiple warps, thereby resolving the program-order dependency. Figure 3b shows the data loading mechanism of FServe, which shows true parallel asynchronous data loading without potential starvation from sequential lock acquisition. In addition, FServe



(a) Deterministic Mode (b) Non-deterministic mode

Figure 5: Two output reduction strategies used in FServe. The initial SM stores its partial results and exits early. Intermediate SMs accumulate subsequent results using appropriate scaling factors through either **(a)** an ordered read-store sequence or **(b)** unordered atomic reductions. Finally, the terminal SM completes the accumulation to produce the final output. The numbered memory access sequences correspond to the detailed steps in Algorithms 1 and 2.



(a) Head Dim 64 (b) Head Dim 256

Figure 6: FP8 prefill phase attention throughput of FServe versus FA3 at head dimensions 64 and 256 under causal and residual masking. The x-axis represents sequence length, with the total token budget fixed at 131,072.

implements proactive TMA prefetching for one shared memory tile ahead, directly fetching tensor tiles into the L2 cache. In our system, sub-8-bit dequantization adds per-tile compute overhead in both linear and attention layers, extending the latency of consuming each input tile. By prefetching the next tensor tile before the current tile queue slot is entirely freed, we effectively overlap memory fetching with computation. This mechanism completely masks the extended consumption delay and guarantees instantaneous TMA loading the moment a queue slot becomes available.

4.2 Decode Phase Acceleration: Dynamic Stream-KV Tile Scheduling

Building on these optimizations, we further enhance the attention kernel to natively support FServe-specific quantization (KV4Q8) in the decode phase. As discussed in Section 2.2, variable-length requests cause workload imbalance across SMs. To resolve this inefficiency, we introduce dynamic Stream-KV tile scheduling inspired by Stream-K GEMM. As shown in Figure 4, this technique evenly distributes irregular workloads across SMs without requiring request sorting or an additional combine kernel. Specifically, the decoding kernel divides the workload evenly across the KV sequence dimension by performing a binary search on the cumulative lengths of the request sequences. To minimize the memory footprint, this initial partitioning is executed entirely on the warp-wide uniform datapath of a single warp within the thread block. Ultimately, all partial results are merged directly in the main kernel, effectively overlapping the reduction process with ongoing computations.

Once the partial results are computed, the decode attention kernel must merge them to generate the final output. This merging process relies on tracking two auxiliary vectors derived from the query $\mathbf{Q}$ and key $\mathbf{K}$ tensors. Specifically, we maintain the row-wise maximum $m = \text{rowmax}(\mathbf{Q}\mathbf{K}^{\mathrm{T}})$ and the sum of exponentials $l = \sum \exp(\mathbf{Q}\mathbf{K}^{\mathrm{T}} - m)$ to correctly scale the partial output $\mathbf{O}$ similar to the Split-KV merging process (Dao et al., 2023). When combining two intermediate results indexed by 1 and 2, the kernel rescales and accumulates their values according to the following equations.

$$m = \max(m_1, m_2),\ l = e^{m_1 - m} l_1 + e^{m_2 - m} l_2,$$
$$\mathbf{O} = e^{m_1 - m}\mathbf{O}_1 + e^{m_2 - m}\mathbf{O}_2. \tag{1}$$

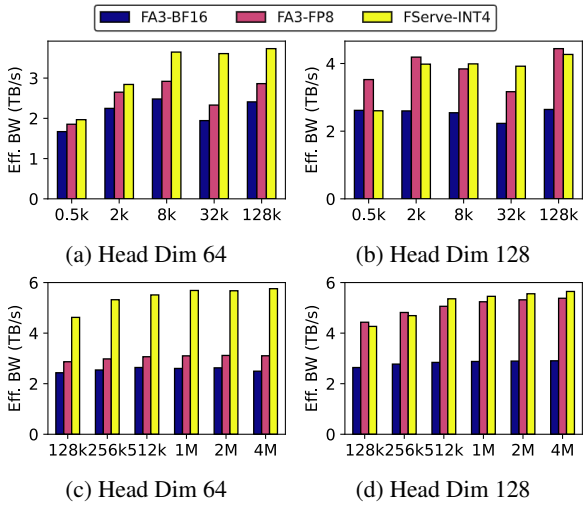
Finally, the accumulated sum vector l is used to normalize the merged output $\mathbf{O}$ by multiplying it by $1/l$ in a row-wise manner. However, existing implementations use a separate combine for merging results.

Output tile reduction. FServe selects between two reduction modes based on the ratio of output KV heads to SMs. The deterministic model (Figure 5a) passes partial results in lock-step when the number of KV heads exceed that of SMs, while the non-deterministic mode (Figure 5b) uses atomic reductions over a dual-buffer system that allows initial SMs to exit early when the number of KV heads are fewer. Detailed algorithms are provided in Appendix C.

5 Evaluation

5.1 Evaluation Setup

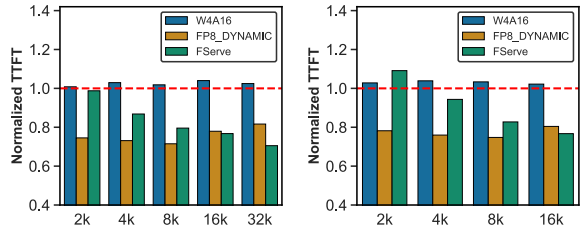
We evaluate FServe on a local cluster equipped with NVIDIA H100 80GB GPUs using Llama-3.1-8B-Instruct and Qwen3-8B (Yang et al., 2025). To evaluate the effectiveness of our algorithm-system



(a) Head Dim 64 (b) Head Dim 128

(c) Head Dim 64 (d) Head Dim 128

Figure 7: Multiple decode kernel performance comparisons. Figure 7a and 7b show the performance comparisons when the total number of tokens is fixed at 128k. Figure 7c and 7d show the long sequence performance comparisons, with the batch size fixed at 1 and varing number of tokens.



(a) Llama-3.1-8B-Instruct (b) Qwen3-8B

Figure 8: Normalized TTFTs compared to the BF16 baseline. FServe consistently outperforms BF16 and W4A16 as sequence length increases. FServe surpasses FP8-dynamic at 16k and beyond for both Llama-3.1-8B and Qwen3-8B, effectively mitigating the dynamic scaling bottleneck of FP8 during compute-bound, long-context prefill operations.

co-design, we integrated our kernels and additional layers into vLLM (Kwon et al., 2023; AI and vLLM Project, 2024) and benchmarked against the quantization schemes it supports. The baselines include the unquantized vLLM (BF16), the native 8-bit vLLM (FP8-dynamic), which applies dynamic activation scaling at runtime and is often constrained by early out-of-memory boundaries, and leading 4-bit schemes such as AWQ (W4A16, W4A8), which retain the KV cache in full precision. Since none of these baselines quantizes the KV cache, Appendix G additionally reports a matched bit-width W4A8KV4 comparison with QServe, where FServe reduces end-to-end latency by 1.5–5.1 times.

5.2 Prefill Phase Acceleration

We evaluate prefill phase performance at two levels, attention kernel throughput and end-to-end Time-to-First-Token (TTFT).

Prefill phase acceleration. Figure 6 compares the FP prefill-phase attention throughput of FServe against the FA3 baseline at head dimensions 64 and 256. We fix the total token budget at 131,072 and the query hidden dimension at 6,144 with 4:1 grouped-query attention ratio, varying sequence length and batch size to span representative workload shapes. Across both head dimensions and under both causal and residual masking, FServe outperforms FA3 in most configurations, achieving up to 1.13× FA3's throughput at head dimension 256 with 32,768 context length and residual masking (1,233 vs 1,088 TFLOPS).

Time-to-First-Token (TTFT) analysis. Figure 8 illustrates the normalized TTFT against the BF16 baseline, showing consistent performance trends across both Llama-3.1-8B-Instruct and Qwen3-8B. While FServe incurs a near BF16 TTFT at short contexts for both Llama-3.1-8B and Qwen3-8B, this initial overhead is rapidly offset as sequences lengthen, achieving up to 1.41× improvement over BF16 for Llama-3.1-8B and 1.30× for Qwen3-8B.

Notably, FServe crosses over hardware-native FP8-dynamic at 16k for both models. By 32k, FServe extends its performance lead for Llama-3.1-8B, achieving 1.16× speedup, 1.05× for Qwen3-8B at 16k. This confirms that at extended lengths, the dynamic scaling and casting overhead in the native FP8-dynamic creates a substantial compute bottleneck. Conversely, FServe effectively masks this overhead via micro-architectural kernel optimizations (Section 4.1). This crossover also holds for end-to-end serving under multi-GPU (TP=4) execution (Appendix H). Finally, FServe consistently outperforms W4A16 across entire regimes, mitigating the inherent dequantization bottleneck that limits the efficiency of standard 4-bit methods.

5.3 Decode Phase Acceleration

We evaluate decode phase performance through memory bandwidth efficiency and the effectiveness of Stream-KV tile reduction.

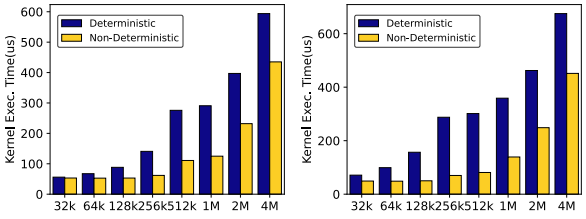
Effective memory bandwidth. We report BF16-equivalent effective memory bandwidth ($\mathrm{BW}_{\mathrm{effective}}$), converted from the measured memory traffic ($\mathrm{BW}_{\mathrm{measured}}$) for fair comparison across precision formats:

$$\mathrm{BW}_{\mathrm{effective}} = \mathrm{BW}_{\mathrm{measured}} \times \frac{\mathrm{bytes}_{\mathrm{BF16}}}{\mathrm{bytes}_{\mathrm{quant}}}.$$

Here, $\mathrm{bytes}_{\mathrm{quant}}$ accounts for the byte cost of the

Table 1: Wikitext-2 perplexity and downstream benchmark scores on Llama-3.1-8B-Instruct. Lower PPL is better, while higher task scores are better. HS, WG, and HE denote HellaSwag, WinoGrande, and HumanEval, respectively. Avg. is computed over the downstream task scores, excluding PPL.

Model	Precision	Algo.	PPL ↓	PIQA ↑	ARC-e ↑	ARC-c ↑	HS ↑	WG ↑	GSM8K ↑	MMLU ↑	HE ↑	Avg. ↑
Llama-3.1-8B	BF16	BF16	8.827	79.978	81.186	52.730	57.418	67.797	80.439	63.210	67.682	68.805
	FP8	FP8	8.893	80.032	80.892	52.474	57.179	68.271	79.548	62.334	64.634	68.171
	W4A16	AWQ	9.465	**79.597**	79.503	50.426	**56.452**	65.824	74.886	**62.127**	59.146	65.995
	W4A8	AWQ	9.539	79.162	78.956	**50.682**	56.253	65.272	75.189	60.874	60.365	65.844
	W4A8KV4	FServe	**9.328**	78.998	**81.313**	50.085	56.413	**67.561**	**75.265**	61.294	**60.975**	**66.488**



(a) Head Dim 64 (b) Head Dim 128

Figure 9: Performance comparisons between deterministic and non-deterministic reduction strategies. Batch size is fixed to 1, and hidden dimension is 6,144, with 4 query heads per key-value head.

quantized weights, activations, and KV cache.

Figure 7 reports $BW_{measured}$ under matched memory-load configurations, using the same hidden dimension configuration as Figure 6. Under this configuration, FServe achieves higher effective bandwidth than BF16 and remains competitive with, or even above, the FP8 format with FA3, while using the more compressed W4A8KV4 format. FServe's BF16-equivalent effective bandwidth surpasses the H100's theoretical memory bandwidth of 3.35 TB/s. It outperforms hardware-native FP8 despite the additional dequantization overhead, indicating compression-adjusted logical delivery rather than physical memory utilization beyond the hardware peak. Appendix D details the KV-wide consumer warpgroup placement in the decode kernel that underlies these results.

Performance of output tile reduction. We evaluate the performance of the non-deterministic reduction mode compared to the deterministic mode. Figure 9 shows the decode attention kernel execution time for both modes, with batch size fixed at 1 and a single head. This setup maximizes the reduction overhead since all partial results should be merged into a single final output. Across multiple head dimensions, non-deterministic reduction achieves significantly lower kernel execution time, saving up to 390 μs. However, this mode cannot estimate the number of split output heads in the variable-length attention decode, so FServe uses it only when the number of output tiles is smaller than the total number of SMs.

5.4 Quantization Fidelity

Table 1 reports perplexity and downstream benchmark scores to evaluate how closely each low-bit method preserves the reference model quality. We use BF16 and FP8 as high-fidelity references, where BF16 represents the uncompressed model quality, and FP8 represents the accuracy attainable with hardware-native 8-bit quantization. The low-bit methods below the separator line are then compared based on how closely they approach these references under stronger compression.

On Llama-3.1-8B, FServe is closer to the BF16 and FP8 references than the AWQ low-bit baselines despite using the more aggressively compressed W4A8KV4 format. FServe achieves 9.328 PPL, lower than W4A16 and W4A8 (9.465 and 9.539, respectively), and the highest average score (66.488) among low-bit methods, outperforming both AWQ baselines (65.995 and 65.844). FServe therefore outperforms the AWQ baselines on both axes, achieving higher accuracy under more aggressive KV compression.

This result shows that FServe improves the accuracy-compression trade-off through its quantization design rather than by avoiding aggressive compression. Channel pre-scaling and group-wise scale relaxation for linear layers, together with post-RoPE key smoothing for the KV cache, mitigate the accuracy loss that typically separates low-bit quantization from BF16 and FP8 reference quality.

Ablation study. Table 2 evaluates the cumulative effect of FServe's quantization components on ARC-e and HumanEval task scores. Linear-only quantization first exposes the loss caused by INT4 weights with FP8 group scales. Our relaxed two-step quantization reduces this loss by preventing FP8 scale underflow, and improves reconstruction by recovering the underused region of the asymmetric INT4 range, validating the linear layer design in Section 3.1.

Table 2: Ablation study on Llama-3.1-8B-Instruct.

Configuration	ARC-e	Δ (diff.)	HumanEval	Δ (diff.)
BF16 (baseline)	81.19		67.68	
+ linear quant.	79.17	0.0	62.20	0.0
+ **relaxed 2-step quant.**	80.68	+1.51	64.02	+1.82
+ KV quant.	80.68	+0.0	0.0	-64.02
+ **shrunk centering**	80.85	+0.17	62.80	+62.80
+ **range alignment**	81.31	+0.46	60.98	-1.82

We then apply KV quantization and progressively enable key outlier smoothing. Naive KV quantization extremely decreases HumanEval task score due to large channel-wise means and inter-channel range imbalance, while shrunk centering and range alignment recover fidelity by reducing mean magnitudes and equalizing channel ranges. The final row matches the full FServe configuration, demonstrating that FServe's fidelity comes from the combined calibration of both linear and attention layers, as designed in Section 3.2. Appendix I reports the same comparison on Llama-3.3-70B-Instruct, where FServe shows the same pattern at a substantially larger model scale.

5.5 System-Level Ablation

Prefill attention. We ablate the two prefill optimizations of Section 4.1. Instruction reordering alone yields a 8.96% geomean throughput gain over the baseline, and 19.4–43.5% at head dimension 64, where softmax accounts for a larger share of the kernel than at larger head dimensions. Adding multi-warp asynchronous TMA contributes a further 1.49% in geomean, concentrated at large head dimensions, bringing peak FP8 attention throughput to 1.3 PFLOPS. Appendix E reports the per-configuration results.

Stream-KV tile scheduling. We stress the scheduler with a worst-case variable-length decode workload of 132 requests, one per SM, with KV lengths drawn from a Gaussian distribution (mean 65,536, standard deviation 32,768). Under FA3's split-KV scheduling, per-SM active cycles show a relative min–max spread of 38.2%, causing completion to be gated by the most heavily loaded SMs while less-loaded SMs become idle earlier. Stream-KV reduces this spread to 3.4%. Despite the additional INT4 dequantization work, both kernels complete in 1.52 ms, showing that Stream-KV maintains comparable kernel time with a substantially more balanced workload distribution. The associated overhead remains small for both approaches. FA3 spends 24.2 μs across two auxiliary kernel launches per invocation, while Stream-KV spends 36.7 μs on in-kernel binary-search partitioning, already included in its kernel time. Appendix F details the workload construction, profiling setup, and per-SM cycle statistics (Figure 13).

6 Conclusion

We presented FServe, a hardware-software co-designed framework that translates W4A8KV4 memory savings into FP8-native inference throughput. FServe combines accuracy-preserving quantization for linear and attention layers with FA3-based kernels that hide dequantization overhead and balance variable-length decode workloads. On Llama-3.1-8B and Qwen3-8B, FServe outperforms AWQ baselines under stronger compression, achieves up to 1.41× TTFT speedup over BF16, and sustain effective decode bandwidth above the H100's theoretical peak. FServe's design principles apply to upcoming GPU generations that support native FP4 and microscaling formats.

Limitations

While FServe demonstrates improvements in FP8-native quantization and throughput optimization, several limitations remain. First, our evaluation focuses on the NVIDIA Hopper architecture. Porting FServe to other architectures, including those with native support for FP4 or microscaling formats, would require additional engineering. Appendix L discusses which components are likely to transfer and which would require redesign. Second, our evaluation is limited to dense LLMs, and extending FServe to mixture-of-experts architectures remains future work (Appendix L). Finally, the parameters used for both linear- and attention-layer quantization are calibrated offline. Appendix J shows that accuracy remains stable across calibration corpora (Pile, WikiText, and C4) and calibration sizes, but robustness to larger distribution shifts between calibration and inference data, such as in domain-specific serving, warrants further study.

References

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In *18th USENIX symposium on operating systems design and implementation (OSDI 24)*, pages 117–134.

Red Hat AI and vLLM Project. 2024. LLM Compressor.

Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L Croci, Bo Li, Pashmina Cameron, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. 2024. Quarot: Outlier-free 4-bit inference in rotated llms. *Advances in Neural Information Processing Systems*, 37:100213–100240.

Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. 2024. Punica: Multi-tenant lora serving. *Proceedings of Machine Learning and Systems*, 6:1–13.

Tri Dao, Daniel Haziza, Francisco Massa, and Grigory Sizov. 2023. Flash-decoding for long-context inference.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in neural information processing systems*, 35:30318–30332.

Advanced Micro Devices. 2025. Introducing amd cdna 4 architecture breakthrough ai and hpc acceleration with enhanced ai capabilities, advanced precisions, and high efficiency.

Advanced Micro Devices. 2026. Introducing amd cdna 5 architecture enabling the future of frontier ai with the amd helios rackscale solution.

Elias Frantar, Roberto L Castro, Jiale Chen, Torsten Hoefler, and Dan Alistarh. 2025. Marlin: Mixed-precision auto-regressive parallel inference on large language models. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 239–251.

Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W Mahoney, and Kurt Keutzer. 2024. Ai and memory wall. *IEEE Micro*, 44(3):33–39.

Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Kangdi Chen, Yuhan Dong, and Yu Wang. 2024. Flashdecoding++: Faster large language model inference with asynchronization, flat gemm optimization, and heuristics. In *Proceedings of Machine Learning and Systems*, volume 6, pages 148–161.

Sehoon Kim, Coleman Hooper, Thanakul Wattanawong, Minwoo Kang, Ruohan Yan, Hasan Genc, Grace Dinh, Qijing Huang, Kurt Keutzer, Michael W. Mahoney, Sophia Shao, and Amir Gholami. 2023. Full stack optimization of transformer inference. In *Architecture and System Support for Transformer Models (ASSYST @ISCA 2023)*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Hyungdeok Lee, Guhyun Kim, Dayeon Yun, Ilkon Kim, Yongkee Kwon, and Euicheol Lim. 2024. Cost-effective llm accelerator using processing in memory technology. In *2024 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*, pages 1–2. IEEE.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of machine learning and systems*, 6:87–100.

Yujun Lin, Haotian Tang, Shang Yang, Zhekai Zhang, Guangxuan Xiao, Chuang Gan, and Song Han. 2025. Qserve: W4a8kv4 quantization and system co-design for efficient llm serving. *Proceedings of Machine Learning and Systems*, 7.

Zechun Liu, Changsheng Zhao, Igor Fedorov, Bilge Soran, Dhruv Choudhary, Raghuraman Krishnamoorthi, Vikas Chandra, Yuandong Tian, and Tijmen Blankevoort. 2025. Spinquant: Llm quantization with learned rotations. In *International Conference on Learning Representations*, volume 2025, pages 92009–92032.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. 2024. KIVI: A tuning-free asymmetric 2bit quantization for KV cache. In *Forty-first International Conference on Machine Learning*.

NVIDIA. 2025. Nvidia blackwell architecture technical brief.

NVIDIA. 2026. Parallel thread execution isa.

Muhammad Osama, Duane Merrill, Cris Cecka, Michael Garland, and John D. Owens. 2023. Stream-k: Work-centric parallel decomposition for dense matrix-matrix multiplication on the gpu.

Bita Darvish Rouhani, Nitin Garegrat, Tom Savell, Ankit More, Kyung-Nam Han, Ritchie Zhao, Mathew Hall, Jasmine Klar, Eric Chung, Yuan Yu, Michael Schulte, Ralph Wittig, Ian Bratt, Nigel Stephens, Jelena Milanovic, John Brothers, Pradeep Dubey, Marius Cornea, Alexander Heinecke, Andres Rodriguez, Martin Langhammer, Summer Deng, Maxim Naumov, Paulius Micikevicius, Michael Siu, and Colin

Verrilli. 2023. Ocp microscaling formats (mx) specification.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *Advances in Neural Information Processing Systems*, 37:68658–68685.

Lei Wang, Lingxiao Ma, Shijie Cao, Quanlu Zhang, Jilong Xue, Yining Shi, Ningxin Zheng, Ziming Miao, Fan Yang, Ting Cao, et al. 2024. Ladder: Enabling efficient {Low-Precision} deep learning computing through hardware-aware tensor transformation. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 307–323.

Lucas Wilkinson. 2024. Introducing , a mixed-input gemm kernel optimized for nvidia hopper gpus. https://developers.redhat.com/articles/2024/10/14/introducing--mixed-input-gemm-kernel. Red Hat Developer Blog.

Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. 2020. Integer quantization for deep learning inference: Principles and empirical evaluation. *arXiv preprint arXiv:2004.09602*.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pages 38087–38099. PMLR.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Jintao Zhang, Pengle Zhang, Jun Zhu, Jianfei Chen, et al. 2025. Sageattention: Accurate 8-bit attention for plug-and-play inference acceleration. In *International Conference on Learning Representations*, volume 2025, pages 71566–71585.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210.

A Motivation: Hardware Landscape and Design Rationale

While W4A4 quantization offers higher theoretical arithmetic throughput, realizing this advantage on NVIDIA Hopper is substantially limited by the available instruction-level execution paths. Hopper achieves its highest low-precision matrix-multiplication throughput through warpgroup-level matrix multiply-accumulate (WGMMA) instructions, such as `wgmma.mma_async`. These instructions natively support FP8 and, importantly, can consume operands directly from shared memory. When coupled with the Tensor Memory Accelerator (TMA), this shared-memory operand path enables an asynchronous pipeline that overlaps data movement with tensor-core computation while avoiding explicit register staging of matrix operands.

Sub-byte integer matrix multiplication lacks an equivalent execution path. Although Hopper retains 1-bit and 4-bit integer MMA operations through PTX (NVIDIA, 2026), they are exposed through legacy synchronous warp-level instructions (`mma.sync`) rather than WGMMA. Consequently, INT4 operands must be staged and reformatted in registers before they can be consumed by the tensor cores. In particular, the mismatch between the packed 4-bit representation and the operand layout expected by the MMA instructions introduces additional unpacking and register-management overhead. A warp must therefore retain both packed data and intermediate operand representations, increasing register pressure and reducing achievable occupancy. The resulting reduction in resident warps weakens latency hiding and prevents the INT4 path from exploiting the same TMA–WGMMA asynchronous pipeline available to FP8. As a result, the nominal throughput advantage of INT4 arithmetic is difficult to realize in practice on Hopper.

These constraints are unlikely to be relaxed in subsequent generations, which deprecate sub-byte integer paths and instead extend low-precision support toward narrower floating-point formats with hardware-native microscaling (NVIDIA, 2025; Rouhani et al., 2023; Devices, 2025). INT4 is thus a diminishing target as an arithmetic format, even as 4-bit storage remains valuable wherever capacity and bandwidth dominate.

Accordingly, `FServe` adopts W4A8KV4 to decouple storage precision from compute precision.

```
00007fdd b37b6b30      MUFU.EX2.F16 R72, R80
00007fdd b37b6b40      HADD2 R64, R64, RZ.H0_H0
00007fdd b37b6b50      MUFU.EX2.F16 R33, R80.H1
00007fdd b37b6b60      PRMT R58, R58, 0x5410, R99
00007fdd b37b6b70      PRMT R99, R58, 0x5410, R99
00007fdd b37b6b80      MUFU.EX2.F16 R66, R74
00007fdd b37b6b90      MUFU.EX2.F16 R27, R74.H1
00007fdd b37b6ba0      PRMT R72, R72, 0x5410, R33
00007fdd b37b6bb0      PRMT R33, R72, 0x5410, R33
```

Figure 10: Duplicated integer permutation (PRMT) instructions generated in naive FP16 softmax implementation. The nvcc compiler inserts two duplicate PRMT instructions which writes same results into different registers (R72, R33) after two exponential instructions(MUFU.EX2.F16), causing integer ALU pipeline bottleneck.

```
00007fce 5b7b6a60      MUFU.EX2.F16 R47, R85
00007fce 5b7b6a70      MUFU.EX2.F16 R78, R85.H1
00007fce 5b7b6a80      PRMT R46, R46, 0x5410, R77
00007fce 5b7b6a90      MUFU.EX2.F16 R48, R86
00007fce 5b7b6aa0      MUFU.EX2.F16 R79, R86.H1
00007fce 5b7b6ab0      PRMT R47, R47, 0x5410, R78
00007fce 5b7b6ac0      MUFU.EX2.F16 R52, R90
00007fce 5b7b6ad0      MUFU.EX2.F16 R83, R90.H1
00007fce 5b7b6ae0      PRMT R48, R48, 0x5410, R79
```

Figure 11: The number of permutation instructions is reduced after the reordering the addition and FP8 conversion instructions into the beginning of the iteration loop.

Weights and KV caches, which dominate model capacity and memory traffic, are stored in 4-bit form and dequantized to FP8 in registers immediately before computation, so that matrix multiplications execute through the asynchronous WGMMA pipeline. Activations remain at 8 bits, which additionally avoids the accuracy degradation that 4-bit activations incur under the outlier-rich distributions characteristic of LLM activations (Dettmers et al., 2022; Xiao et al., 2023), and which existing W4A4 methods address only through additional transformations such as online rotations (Ashkboos et al., 2024; Liu et al., 2025). The result retains the memory benefits of 4-bit quantization without forfeiting the throughput of FP8 computation.

B Detailed Reasoning of Reordering Softmax Pipeline

While the Hopper architecture natively supports most packed-FP16 operations (e.g., `VHMNMX`, `HADD2`, `HMUL2`), it lacks a native packed-FP16 exponential instruction. To emulate this, the `nvcc` compiler generates two separate SASS FP16 exponential operations (`MUFU.EX2.F16`) that write to different registers. As illustrated in Figure 10, an integer permuta-

Algorithm 1 Deterministic Output Reduction

Require: row max m, exp sum l, output O
Require: start, end idx i_s, i_e, # of KV iters i_{KV}
Require: SM idx n, Reduction $R[]$, Barrier $B[]$
Initialize $B[]$ to 0
if $i_s > 0$ **then**
  Wait until $B[n-1] == i_s$
  $m_r :\leftarrow R[n-1].m$ ▷ Read results ②
  $m_t := \max(m, m_r)$
  $s_l := \exp(m - m_t), s_r := \exp(m_r - m_t)$
  $\{l_r, O_r\} :\leftarrow R[n-1]\{.l, .O\}$
  $l = s_l l + s_r l_r, O = s_l O + s_r O_r, m = m_t$
end if
if $i_s + i_{KV} < i_e$ **then**
  $R[n] \leftarrow \{m, l, O\}$ ▷ Write results ①
  Signal $B[n] = i_s + i_{KV}$
end if

tion (PRMT) is then required to repack these separate register values into a single output. Because these repacked results immediately feed into subsequent addition and FP8 conversion instructions, the compiler duplicates the results for generating separate input of addition and conversion. These redundant integer operations saturate the ALU pipeline, degrading computational throughput.

To explicitly eliminate these redundant instructions, we strategically reorder the operations inside the attention main loop (Figure 2). In the original layout, the exponential computation is immediately followed by the addition (Add $\mathbf{S}_i$) and conversion (Cvt $\mathbf{P}_i$) steps. We resolve this by shifting these final two operations to the very beginning of the loop. Specifically, we execute the addition and conversion for the previous iteration ($\mathbf{S}_{i-1}$ and $\mathbf{P}_{i-1}$) as an epilogue phase before initiating the matrix multiplications for the current iteration. By splitting the softmax operation in this manner, we successfully break the immediate dependency chain. This structural adjustment prevents the compiler from mistakenly identifying false dependencies between the end and the start of the loop. Consequently, we completely bypass the flawed automatic optimizations that trigger instruction duplication. As demonstrated in Figure 11, the resulting SASS code does not have such redundant permutation instructions.

C Two Algorithms of Output Tile Reduction

The system dynamically selects the reduction mode based on the ratio between the number of output KV heads and the total number of SMs. If the number of KV heads is greater than or equal to the number of SMs, we apply the deterministic reduction mode. As illustrated in Figure 5(a), this

Algorithm 2 Non-deterministic Output Reduction

Require: row max m, exp sum l, output O
Require: start, end idx i_s, i_e, # of KV iters i_{KV}
Require: Output idx n, Reduction $R[]$, Barrier $B[]$
Initialize $B[]$ to 0
if $i_s == 0$ and $i_{KV} < i_e$ **then**
  $R[2n] \leftarrow \{m, l, O\}$ ▷ Write & init ①
  $R[2n+1] \leftarrow \{m, 0, 0\}$
  Signal $B[2n] = i_s, B[2n+1] = i_s$
else if $i_s > 0$ **then**
  Wait until $B[2n] > 0$
  $R[2n+1].m \leftarrow_{\max} m$ ▷ Atomic Reduce max m ②
  Signal $B[2n] \leftarrow_{+} i_{KV}$
  Wait until $B[2n] == i_e$
  $m_t :\leftarrow R[2n+1].m$ ▷ Read true max m ③
  $s_l := \exp(m - m_t), l = s_l l, O = s_l O$
  $m = m_t$
  if $i_s + i_{KV} < i_e$ **then** ▷ Reduce add l, O ⑤
    $R[2n+1]\{.l, .O\} \leftarrow_{+} \{l, O\}$
    Signal $B[2n+1] \leftarrow_{+} i_{KV}$
  else ▷ Read init results ④
    $\{m_i, l_i, O_i\} :\leftarrow R[2n]\{.m, .l, .O\}$
    $s_i := \exp(m_i - m)$
    $l = l + s_i l_i, O = O + s_i O_i$
    Wait until $B[2n+1] == i_s$ ▷ Read results ⑥
    $\{l_r, O_r\} :\leftarrow R[2n+1]\{.l, .O\}$
    $l = l + l_r, O = O + O_r$
  end if
end if

approach passes the partial results to the next SM in a strict lock-step manner and maintains a reduction buffer size exactly equal to the number of SMs.

Conversely, if the number of KV heads is strictly smaller than the number of SMs, FServe choose the non-deterministic reduction mode. Under this scheme, multiple SMs simultaneously perform atomic reductions without enforcing a strict order. Figure 5(b) depicts this architecture, revealing a dual-buffer system designed to allow the initial SM to store its results and exit early without waiting.

① At first, write only SM writes its row-wise maximum, sum of exponentials, and partial outputs into initial and reduce buffer twice. ② Then, all the other SMs working on the same outputs perform atomic max operations on row-wise maximum at reduce buffer, computing shared true maximum across all SMs. ③ When the atomic maximum operations are finished, all SMs other than initial one shares the true maximum for computing scaling in Equation 1. ④ From that true row-wise maximum, the last SM reads the partial results of the initial SM from the initial buffer, and merge them with Equation 1. While the last SM reads the initial results, ⑤ other intermediate SMs rescale their summation of exponentials and partial outputs, and performs atomic addition on reduction buffers. ⑥, after the reduction finished, the last SM reads the accumulated results, and simply add them for final

output. Algorithms 1 and 2 detail this rescaling accumulation of partial results across SMs by supporting both deterministic and non-deterministic reduction paths.

D KV-wide Consumer Warpgroup Placement

For the decoding phase, we changed the mapping of the consumer warpgroups to along with the key-value dimension, since there are a few query tokens per key-value head. Such re-mapping warpgroups can enhance the effectiveness of softmax computation, reducing the number of softmax operations to 1/3 at head dimension 64 and 1/2 at other dimensions. As the number of executed exponential operations decreases, results of Figure 7a can be explained even though the utilization of special function unit decreases from 65% of the FA3 to 35%. Therefore, warpgroup-level partial output reduction is required between consumer warpgroups. Such in-CTA warpgroup-level tile reduction is performed at shared memory before global GPU-level tile reduction.

E Detailed Evaluation Results of Prefill Attention Optimization Techniques

Figure 12 shows the ablation of the two prefill attention techniques. Both techniques are also applied to decode attention. Instruction reordering could achieve 8.96% performance improvement over baseline in geomean. For head dimension 64, such reordering alone could achieve 19.4-43.5% advances, as the proportion of softmax computation relative to matrix-matrix multiplication is large compared to larger head dimensions. Building upon the instruction reordering, adding multi-warp asynchronous TMA technique can achieve additional 1.49% performance improvement in geomean, especially on larger head dimension cases. With such optimizations, FServe FP8 attention could reach 1.3 PFLOPS at maximum from the 1.17 PFLOPS by FlashAttention-3 (Shah et al., 2024) in the same scenario.

F Isolating the Effect of Stream-KV Tile Scheduling in Decode Attention

To evaluate the effectiveness of the Stream-KV tile scheduler, we generate 132 variable-length decode requests so that each SM handles at least one request, with each request containing a single query token. The KV lengths (Table 3) are drawn from a Gaussian distribution with mean 65,536 and standard deviation 32,768, truncated to [1, 131072]. We use one query, key, and value head with a head dimension of 256. Using NVIDIA Nsight Compute 2026.2.1, we profile the `FServe` decode attention kernel with an INT4 KV cache and compare it against the FlashAttention-3 (FA3) variable-length forward kernel with automatic KV splitting (`num_splits=0`) and an FP8 E4M3 KV cache.

Figure 13 shows that Stream-KV distributes the workload substantially more evenly across SMs. With FA3, per-SM active cycles range from 1.87M to 2.76M around an average of 2.34M, yielding a relative min–max spread of 38.2%. As a result, completion is determined by the most heavily loaded SMs while less loaded SMs become idle. In contrast, `FServe` narrows the range to 2.68M–2.77M around a 2.72M average, reducing the spread to only 3.4%. Despite the higher absolute cycle count caused by INT4 dequantization, both kernels complete in 1.52 ms at similar SM frequencies ($\approx$1.8 GHz). The absolute cycle counts therefore should not be interpreted as a direct performance comparison. Instead, the near-uniform distribution indicates that Stream-KV compensates for its additional per-SM work through substantially better load balance, allowing it to match FA3's kernel time. In addition, we also measured the execution tile of FA3, with all 132 requests have 64,001 context lengths, similar to the arithmetic mean of Table 3. In this uniform decoding, FlashAttention-3 uses static persistent tile scheduler, reporting 1.41 ms execution time, which is 9.93% faster than variable-length cases.

Beyond load imbalance within the main kernel, FA3 also incurs auxiliary-kernel overhead. Each decode attention invocation launches one kernel to determine the number of output tiles for KV splitting and another to combine the resulting partial outputs, adding a fixed cost of 17.63 and 6.60 μs per launch, making the total 1.55 ms, respectively. Stream-KV tile scheduling also has similar initial binary-search tile distribution overhead, measured as 36.7 μs, as shown in Figure 13b. Table 4 summarizes the profiled results of three decoding attention kernels.

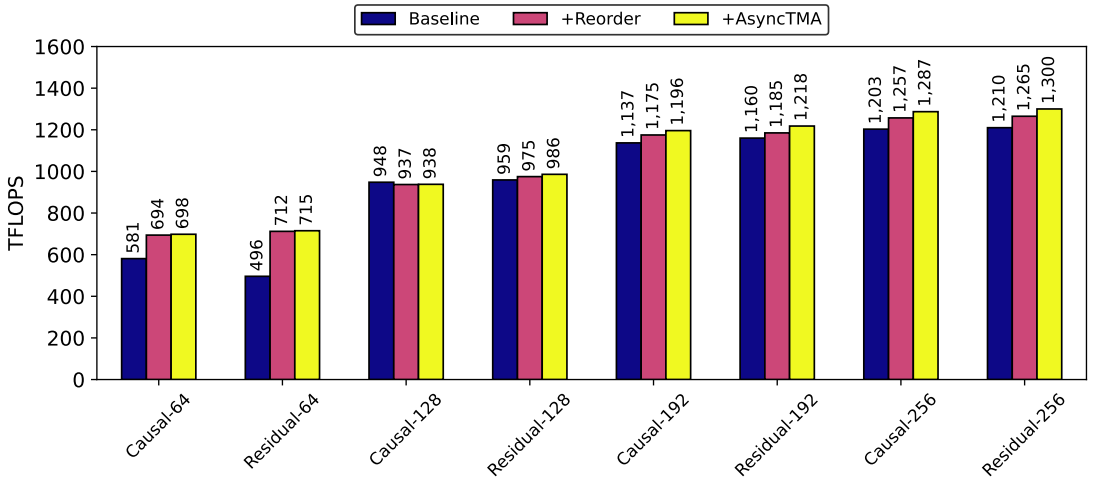


Figure 12: Effect of instruction reordering and multi-warp asynchronous TMA on FP8 prefill attention throughput, at head dimensions 64, 128, 192, and 256 under residual and causal masking. Baseline is the FA3 pipeline with only naive packed-FP16 softmax computation. Batch size 1, sequence length 16,384, query hidden dimension 6,144, key/value hidden dimension 1,536.

Table 3: The 132 randomly generated KV lengths, one per decode request. Lengths are drawn from a Gaussian distribution with mean 65,536 and standard deviation 32,768, truncated to [1, 131,072]; each request carries a single query token.

Req. 0–10	40,609	58,628	35,452	64,806	84,414	105,167	78,649	45,024	76,326	41,487	15,099
Req. 11–21	37,100	10,552	70,402	87,202	37,224	39,966	20,044	10,208	63,982	106,390	108,194
Req. 22–32	66,668	56,549	26,776	39,188	32,780	35,898	32,096	77,481	114,777	65,318	69,299
Req. 33–43	82,524	13,799	54,483	52,072	76,940	38,395	32,761	71,380	53,491	63,539	86,160
Req. 44–54	67,254	75,531	72,169	79,034	104,792	95,940	60,909	82,057	74,757	93,246	48,666
Req. 55–65	97,386	35,924	23,289	82,061	76,189	67,897	98,110	23,516	106,724	77,601	33,208
Req. 66–76	66,540	90,191	75,099	29,496	71,960	91,243	63,970	64,745	45,753	53,974	123,429
Req. 77–87	50,545	65,772	10,929	61,361	86,420	54,749	46,193	83,723	53,948	52,766	61,230
Req. 88–98	71,466	45,869	55,350	54,087	45,119	28,103	78,177	130,277	46,592	52,202	81,301
Req. 99–109	67,770	61,856	45,502	79,546	60,433	125,172	66,888	79,312	48,174	84,635	130,330
Req. 110–120	57,812	26,304	93,247	42,785	91,502	88,764	102,992	38,554	38,685	127,573	33,303
Req. 121–131	32,661	71,759	100,037	26,031	86,320	90,072	44,322	51,843	99,634	9,836	109,394

Table 4: Profiled results of decode attention kernels. Variable length cases were evaluated with context lengths of Table 3, and fixed length case was evaluated with 64,001 context length, similar to the arithmetic mean of Table 3.

Kernel	Main Exec. Time	Aux. Exec. Time	Total Exec. Time
FA3-VarLen	1.52 ms	18.23 μs	1.55 ms
FA3-FixLen	1.41 ms	0 s	1.41 ms
FServe-VarLen	1.52 ms	0 s	1.52 ms

G Matched Bit-Width Comparison with QServe

G.1 Baseline Scope and Integration

The baselines in Section 5.1 are the schemes vLLM supports natively, so that all serving measurements share one scheduling and execution regime. None of them quantizes the KV cache. We therefore add QServe (Lin et al., 2025) as a *matched bit-width* baseline: both systems use INT4 weights, 8-bit activations, and INT4 KV caches, but `FServe` uses FP8 activations whereas QServe uses INT8. They also differ in execution path, attention-kernel implementation, and KV-cache layout. A performance gap between them therefore reflects the combined effect of these differences rather than the contribution of any one of them.

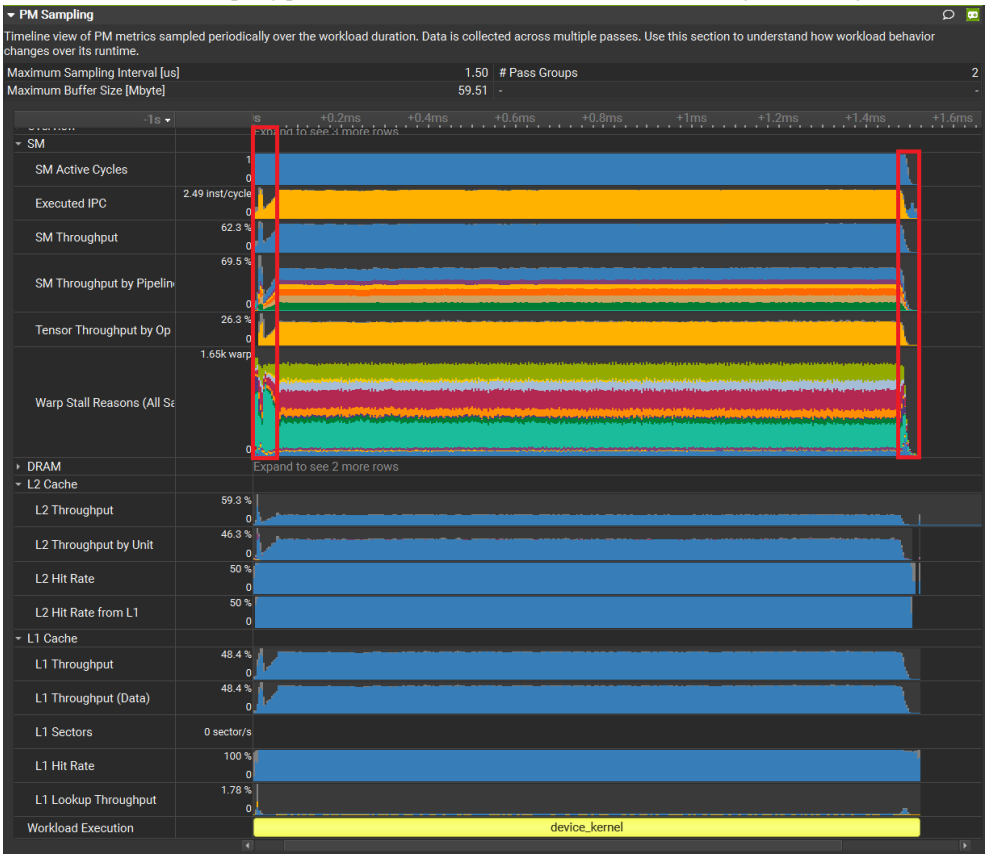
Accuracy is evaluated on Llama-2-13B and Llama-3-8B, the models for which QServe reports results; the serving comparison uses Llama-2-13B, so that QServe runs on a configuration it releases.

SpinQuant (Liu et al., 2025) is excluded because vLLM does not support its 4-bit KV-cache inference path. LServe is excluded because it performs block-sparse rather than full attention, so its latency is not comparable to systems that attend over the entire KV cache.

Accuracy. Accuracy is measured offline and does not involve the serving stack. Downstream task scores are obtained through lm-eval-harness, in-



(a) PM Sampling profiled results of FlashAttention-3 variable length decoding.



(b) PM Sampling profiled results of FServe variable length decoding.

Figure 13: PM Sampling profiled results of attention kernels. Red boxes show the head and tail latency of these kernels.

Table 5: Accuracy under a matched W4A8KV4 bit-width configuration. Both systems use INT4 weights, 8-bit activations, and INT4 KV caches; FServe uses FP8 activations whereas QServe uses INT8. Scores are acc_norm, whereas Section 5.4 uses acc, so values are not comparable between the two. Bold marks the better low-bit system within each model. Avg. is computed over the five tasks. HS: HellaSwag, WG: WinoGrande.

Model	Precision	Algo.	PIQA ↑	ARC-e ↑	ARC-c ↑	HS ↑	WG ↑	Avg. ↑
Llama-2-13B	FP16	FP16	80.36	76.60	48.81	79.63	72.45	71.57
	W4A8KV4	QServe	80.14	75.00	**48.89**	78.63	72.61	71.05
	W4A8KV4	FServe	**80.36**	**77.06**	46.76	**79.20**	**73.09**	**71.29**
Llama-3-8B	BF16	BF16	80.41	77.57	53.92	79.27	73.48	72.93
	W4A8KV4	QServe	79.76	**77.69**	**51.96**	77.57	**73.64**	**72.12**
	W4A8KV4	FServe	**80.14**	75.63	51.79	**78.31**	71.98	71.57

voked from DeepCompressor for QServe and from our own implementation for FServe, with the harness version and task configuration matched. Scores are reported as acc_norm following the QServe paper's setup. This differs from Section 5.4, which reports acc, so values should not be compared between the two.

Serving. QServe is evaluated by connecting the OmniServe engine to the current vLLM v1 scheduler, so request scheduling, admission, and the benchmarking harness are shared with FServe. The model execution engine itself is QServe's own, including the attention and linear-layer kernels and the paged KV-cache manager. All serving experiments run on NVIDIA H100 80GB GPUs, with one GPU at tensor-parallel degree TP=1 and four at TP=4, and both systems are run in the same software environment. Requests are submitted as a single burst of BS concurrent requests, each with input sequence length ISL and a fixed output sequence length OSL of 1024 tokens, and reported values are averaged over repeated runs.

G.2 Accuracy under a Matched Compression Budget

Table 5 compares FServe and QServe on Llama-2-13B and Llama-3-8B. All rows are measured by us rather than taken from published numbers, so the two systems are compared head-to-head under a matched harness version and task configuration, following the setup of the QServe paper (Appendix G.1). That setup differs from the one used for the main results in Section 5.4, so each set of numbers should be read on its own terms.

The two systems are close on both models. On Llama-2-13B FServe attains the higher average (71.29 vs. 71.05). On Llama-3-8B the ordering reverses (71.57 vs. 72.12). On that model QServe also exceeds the BF16 reference on ARC-Easy and WinoGrande, and on Llama-2-13B both low-bit systems exceed the FP16 reference on WinoGrande. Quantization noise therefore does not act in a single direction on these benchmarks, and differences of this size should not be read as a consistent quality ordering.

We therefore read these results as accuracy parity under a matched bit width rather than as an accuracy advantage for either system: neither wins consistently across tasks, and which system leads on average depends on the model. The difference between the two appears instead in end-to-end serving (Appendix G.3), where FServe reduces end-to-end latency by 1.5–$5.1\times$ relative to QServe.

G.3 End-to-End Serving Performance

Tables 6–9 report the full batch-size sweep at TP=1. FServe achieves lower end-to-end latency than QServe in every configuration, by 1.5–$5.1\times$, and reduces latency relative to W4A16 by up to $2.0\times$.

Token volume. FServe's advantage grows with total token volume. At BS=16 it trails W4A16 at 1k and 2k ($0.85\times$ and $0.94\times$) and overtakes it from 4k onward, reaching $1.78\times$ at 16k. At BS=32 it trails only at 1k ($0.93\times$), and at BS$\geq$64 it leads across the entire range. The one departure from this pattern, a drop from $1.98\times$ at 1k to $1.57\times$ at 2k when BS=128, follows from the queueing transition described below.

Throughput grows with input length only for FServe: at BS=16 it rises from 969 to 2757 tok/s between 1k and 16k, while W4A16 stays within 1140–1637 and QServe within 670–1022. From BS=64 onward FServe is already near its ceiling of roughly 2900 tok/s at every input length, whereas both baselines remain flat at approximately 1500 and 900 across the entire sweep.

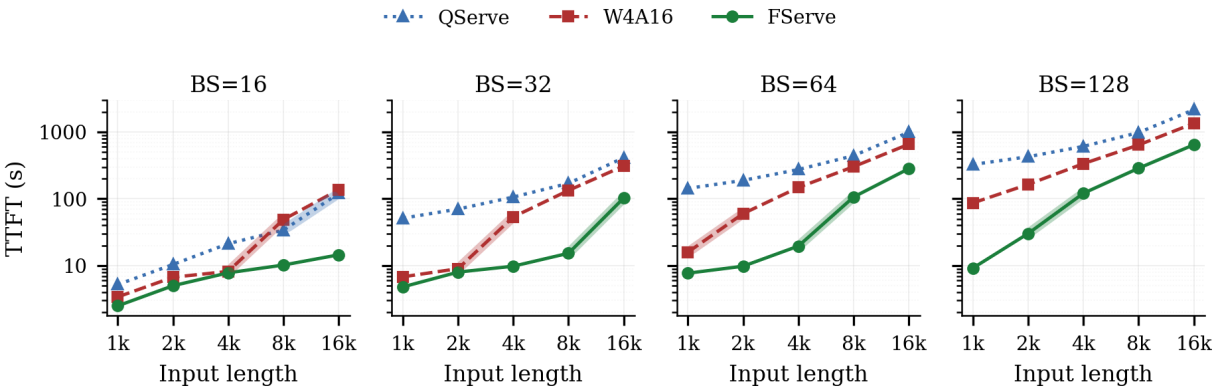


Figure 14: TTFT versus input length at each batch size on Llama-2-13B (TP=1, OSL=1024). Shading marks the transition where TTFT more than triples between adjacent input lengths, indicating that the resident batch can no longer hold all BS requests at once. `FServe` reaches this transition at roughly four times the input length of W4A16, matching the four-fold difference in KV-cache footprint; at BS=16 it does not reach the transition within the evaluated range, and W4A16 has already passed it at BS=128. QServe shows no transition, its TTFT being high already at the shortest input.

Table 6: End-to-end serving on Llama-2-13B at TP=1, BS=16, OSL=1024.

Scheme	ISL	Throughput (tok/s)	TTFT (s)	TPOT (ms)	E2E Latency (s)
W4A16	1k	1140.35	3.4	24.72	28.7
	2k	1446.53	6.7	26.66	34.0
	4k	1624.19	8.2	39.29	48.4
	8k	1636.92	49.0	37.71	87.5
	16k	1508.88	136.6	38.90	176.4
QServe	1k	669.91	5.2	42.75	48.9
	2k	774.67	10.4	51.85	63.4
	4k	913.46	21.3	66.87	89.7
	8k	1021.83	34.0	107.83	144.3
	16k	897.30	120.2	178.57	302.9
FServe	1k	969.42	2.5	30.62	33.8
	2k	1365.12	5.0	30.26	36.0
	4k	1844.83	7.8	35.77	44.4
	8k	2480.45	10.2	48.17	59.4
	16k	2756.85	14.5	82.47	98.9

Table 7: End-to-end serving on Llama-2-13B at TP=1, BS=32, OSL=1024.

Scheme	ISL	Throughput (tok/s)	TTFT (s)	TPOT (ms)	E2E Latency (s)
W4A16	1k	1871.9	6.8	27.5	35.0
	2k	1742.9	9.0	43.8	53.8
	4k	1623.2	53.6	42.5	97.1
	8k	1634.8	133.6	37.9	172.4
	16k	1507.1	310.0	39.0	350.0
QServe	1k	669.6	51.3	42.5	94.8
	2k	777.9	70.2	51.0	122.4
	4k	914.5	106.3	65.7	173.5
	8k	1021.7	170.3	106.8	279.5
	16k	897.6	413.1	178.4	595.6
FServe	1k	1736.4	4.8	32.1	37.7
	2k	2137.9	8.0	37.1	46.0
	4k	2614.5	9.8	51.7	62.6
	8k	2894.0	15.3	83.0	100.2
	16k	2865.9	102.9	81.4	186.2

Resident decode batch. `FServe` shows higher TPOT than W4A16 at long inputs (82.3 vs. 39.1 ms at BS=64, 16k) while reducing TTFT by 2.3× and end-to-end latency by 1.9× in the same configuration. Higher TPOT alongside lower TTFT and higher throughput is consistent with a larger resi-

Table 8: End-to-end serving on Llama-2-13B at TP=1, BS=64, OSL=1024.

Scheme	ISL	Throughput (tok/s)	TTFT (s)	TPOT (ms)	E2E Latency (s)
W4A16	1k	1717.8	15.9	55.7	72.8
	2k	1751.4	60.1	45.7	106.8
	4k	1620.8	147.8	42.8	191.6
	8k	1634.6	302.7	38.0	341.6
	16k	1506.6	656.7	39.1	696.7
QServe	1k	668.3	143.3	42.6	186.9
	2k	777.0	188.8	51.1	241.1
	4k	914.4	274.4	65.6	341.4
	8k	1019.2	442.2	106.9	551.5
	16k	897.6	995.1	178.5	1177.7
FServe	1k	2622.2	7.7	41.4	50.0
	2k	3002.4	9.8	54.4	65.4
	4k	2921.8	19.6	87.1	108.7
	8k	2995.6	106.4	81.3	189.6
	16k	2886.0	282.8	82.3	367.0

Table 9: End-to-end serving on Llama-2-13B at TP=1, BS=128, OSL=1024.

Scheme	ISL	Throughput (tok/s)	TTFT (s)	TPOT (ms)	E2E Latency (s)
W4A16	1k	1718.5	86.9	56.7	144.9
	2k	1768.3	164.6	45.9	211.5
	4k	1633.6	335.9	42.9	379.7
	8k	1634.7	640.9	38.0	679.8
	16k	1507.4	1349.2	39.1	1389.2
QServe	1k	669.1	326.7	42.5	370.2
	2k	774.4	427.4	51.3	479.9
	4k	914.6	610.1	65.5	677.1
	8k	1022.4	981.7	106.4	1090.6
	16k	897.2	2160.2	178.5	2342.9
FServe	1k	3415.9	9.2	62.5	73.1
	2k	2842.5	30.3	101.7	134.4
	4k	3006.4	121.1	86.6	209.7
	8k	3030.1	287.2	81.4	370.5
	16k	2884.3	645.2	83.4	730.5

dent decode batch: a compressed KV cache lets the scheduler keep more of the BS concurrent requests resident at once, so each decode step serves more requests while fewer wait in the queue. QServe's TPOT grows from 42.6 to 178.5 ms with input length at this batch size, whereas FServe stays within 41–87 ms.

The same mechanism produces a visible knee in TTFT, shown in Figure 14. Once the resident batch can no longer hold all BS requests at once, later requests wait for earlier ones to finish, and TTFT grows faster than linearly in input length. Doubling the input roughly halves the number of resident requests, so TTFT can more than double across this transition and then resumes a milder slope. The knee moves to shorter inputs as batch size grows: for FServe it falls between 8k and 16k at BS=32, between 4k and 8k at BS=64, and between 1k and 2k at BS=128, and it lies beyond 16k at BS=16. W4A16 crosses the same boundary at roughly a quarter of the input length at every batch size, which matches the four-fold difference in KV-cache footprint between an INT4 and an FP16 cache; by BS=128 it has already passed the knee at the shortest input evaluated.

QServe shows no such transition at any batch size. Its TTFT is already 326.7 s at BS=128 and 1k, and grows smoothly from there, indicating that it operates in the queueing regime across the entire sweep. This is consistent with the flat throughput discussed below: the limiting factor is not KV-cache capacity, which QServe compresses to the same bit width as FServe.

Execution path. We did not profile QServe's kernels and offer no causal decomposition of this gap. We note only that Hopper exposes FP8 through asynchronous WGMMA but provides no comparable path for narrow integer operations (Ap-

Table 10: End-to-end serving on Llama-2-13B at TP=4, BS=64, OSL=1024. QServe is omitted because it does not support tensor parallelism.

Scheme	ISL	Throughput (tok/s)	TTFT (s)	TPOT (ms)	E2E Latency (s)
BF16	1k	5041.5	3.2	22.3	26.0
	2k	6113.2	4.0	27.5	32.1
	4k	7334.3	5.2	38.6	44.7
	8k	6786.2	31.0	51.4	83.5
	16k	6736.7	106.6	51.2	158.9
FP8-dynamic	1k	6465.6	2.3	17.6	20.3
	2k	7784.2	2.9	21.9	25.2
	4k	9279.2	3.7	30.9	35.3
	8k	10119.9	4.7	51.0	56.8
	16k	9062.0	57.4	60.0	118.7
W4A16	1k	6061.7	3.2	18.0	21.6
	2k	7068.5	4.1	23.1	27.8
	4k	8130.5	6.3	33.3	40.3
	8k	8972.2	8.3	56.1	65.7
	16k	8143.3	54.0	78.3	134.1
FServe	1k	5316.6	2.8	21.3	24.6
	2k	6944.0	3.8	24.0	28.3
	4k	8538.2	4.7	32.9	38.3
	8k	10379.5	6.8	48.9	56.8
	16k	11704.1	9.3	83.9	95.1

pendix A), and that QServe's throughput is flat in batch size, which places the limiting factor in the execution path rather than in request-level parallelism.

H Multi-GPU Serving

Table 10 reports a four-way tensor-parallel (TP=4) configuration on Llama-2-13B, run on four NVIDIA H100 80GB GPUs under the same workload as Appendix G.3: a single burst of BS=64 concurrent requests, each with a fixed output sequence length of 1024 tokens, with reported values averaged over repeated runs. This configuration is reported separately because QServe does not support tensor parallelism, so no matched bit-width comparison is available at TP=4.

At 1k and 2k FServe trails FP8-dynamic and W4A16, by 17% and 12% respectively at 1k, since the fixed cost of dequantization is not yet amortized, although it is already faster than BF16. It overtakes W4A16 at 4k, matches FP8-dynamic at 8k, and leads all three at 16k, where it reaches $1.41\times$ lower end-to-end latency than W4A16, $1.25\times$ than FP8-dynamic, and $1.67\times$ than BF16, and attains the highest throughput (11,704 vs. 9,062 tok/s for FP8-dynamic). This is consistent with the TTFT crossover observed in Section 5.2: FServe trails at short inputs, where the fixed dequantization cost dominates, and overtakes as input length grows.

I Quantization Fidelity at 70B Scale

Table 11 compares FServe against BF16, FP8-dynamic, and W4A16 on Llama-3.3-70B-Instruct. The benchmarks here (GPQA-Diamond, MMLU-Pro, HumanEval, Math500, GSM8K-CoT) are harder than those of Table 1 and use different extraction protocols, so scores are not comparable across the two tables; each should be read within itself.

All low-bit schemes stay within a few points of the BF16 reference. FServe attains the highest GPQA-Diamond score (41.41) and the highest GSM8K-CoT flexible-extract score (94.69), both above the BF16 reference; W4A16 also exceeds BF16 on these two benchmarks. As in Appendix G.2, low-bit scores exceeding the full-precision reference show that quantization noise does not act in a single direction, so differences of this size should not be read as a consistent quality ordering. On the remaining benchmarks FServe trails BF16 by 1.0–3.7 points and W4A16 by at most 1.4 points, while additionally compressing activations to FP8 and the KV cache to INT4, which W4A16 leaves in full precision.

We read these results as the same pattern observed at 8B and 13B scale: FServe stays close to the reference under a more aggressive compression budget than any of the compared schemes, and this holds at a substantially larger dense model.

Table 11: Accuracy on Llama-3.3-70B-Instruct. GPQA-D: GPQA-Diamond (flexible-extract); Math500 uses math-verify; GSM8K-CoT reports strict-match / flexible-extract. Bold marks the best low-bit scheme per column; for GSM8K-CoT the two figures are marked independently.

Model	Precision	Algo.	GPQA-D ↑	MMLU-Pro ↑	HumanEval ↑	Math500 ↑	GSM8K-CoT ↑
Llama-3.3-70B-Instruct	BF16	BF16	38.89	69.51	87.80	74.2	90.60 / 94.47
	FP8	FP8	38.89	69.46	85.37	76.0	90.60 / 94.47
	W4A16	AWQ	40.40	**67.89**	**84.76**	**74.2**	**89.94** / 94.56
	W4A8KV4	FServe	**41.41**	67.24	84.15	72.8	89.61 / **94.69**

J Robustness to the Choice of Calibration Data

The calibrated parameters of Sections 3.1 and 3.2 are estimated from a fixed calibration set, which raises the question of whether FServe's fidelity depends on that choice. Table 12 evaluates FServe on Llama-3.1-8B-Instruct with calibration sets drawn from three corpora (Pile, WikiText, and C4) at three sizes (16, 32, and 64 sequences), reporting five zero-shot tasks.

The results are stable across both axes. The zero-shot average stays within 72.74–73.44 across all nine configurations, a spread of 0.7 points, and no corpus dominates across tasks, although Pile with 64 sequences yields the lowest average. Sensitivity to the calibration size is similarly small: within each corpus, the average varies by at most 0.62 points across the three sizes. All configurations remain within 1.0–1.7 points of the BF16 reference (74.38). FServe's calibration is therefore not overly sensitive to the choice of calibration data within the natural-text domain. Robustness under more severe distribution shift, such as calibrating on natural text and serving code-heavy or multilingual workloads, remains untested.

K Novelty

This appendix provides novelty of FServe beyond its closest prior work and expands on the contributions outlined in Section 1. The novelty of FServe lies in their co-design under a constraint that prior work does not face: every matrix multiplication runs on Hopper's asynchronous FP8 WGMMA path, requiring quantized tensors to be calibrated and represented using FP8 group scales (Appendix A).

Linear-layer quantization under FP8 group scales. Prior weight-quantization methods, including AWQ's per-channel scale search, are calibrated with high-precision scales. When directly applied to FP8 group scales, small scale values can underflow and cause entire groups to collapse to zero (Section 2). FServe is designed for this regime. Channel pre-scaling conditions the weight distribution before quantization, while relaxed two-step quantization introduces per-group relaxation factors to recover the underused portion of the asymmetric INT4 range (Section 3.1). Table 2 isolates the benefit of these techniques, improving ARC-e by 1.51 points and HumanEval by 1.82 points over naive linear quantization under the same scale representation.

Post-RoPE key smoothing with offline shrunk centering and range alignment. Prior KV-cache smoothing methods address key outliers along two axes. For channel means, SageAttention removes the full channel mean online during inference. For channel ranges, QServe rescales keys before RoPE with scales shared across rotary pairs, coupling two channels whose distributions diverge after the rotation. FServe addresses both axes after RoPE and independently per channel. Shrunk centering removes only a calibrated fraction of the channel mean, and range alignment rescales each channel to a common range. Their hyperparameters are selected to minimize the KL divergence between attention distributions before and after quantization (Section 3.2). Inverse query scaling preserves the corresponding dot products, while the logit shift introduced by centering cancels under softmax. Because all smoothing parameters are calibrated offline, no online statistics are required during inference. Table 2 shows that shrunk centering recovers HumanEval from 0.0 to 62.80 after KV quantization; range alignment then raises ARC-e from 80.85 to 81.31 while lowering HumanEval by 1.82 points, reflecting a trade-off between the two smoothing axes.

Kernel-level contributions beyond FA3. At the kernel level, FServe extends the FA3 pipeline in three ways. First, instruction reordering removes compiler-induced duplicate permutation

Table 12: Calibration-data dependency of FServe on Llama-3.1-8B-Instruct. All FServe rows use the W4A8KV4 format; BF16 is the uncalibrated reference. Seqs is the number of calibration sequences. Avg. is the mean of the five zero-shot tasks. HS: HellaSwag, WG: WinoGrande.

Calib. set	Seqs	PIQA	ARC-e	ARC-c	HS	WG	Avg.
BF16 (ref.)	—	81.23	82.24	55.46	79.57	73.40	74.38
Pile	64	80.20	80.98	53.24	78.71	70.56	72.74
	32	80.69	81.48	53.07	78.79	72.77	73.36
	16	81.07	80.56	52.56	78.80	72.38	73.07
WikiText	64	80.30	81.31	54.78	78.54	71.98	73.38
	32	80.41	81.02	54.18	78.63	72.77	73.40
	16	80.36	81.36	54.95	78.62	71.90	73.44
C4	64	81.01	81.36	53.50	78.55	72.77	73.44
	32	81.12	80.60	52.65	78.89	73.01	73.25
	16	81.23	81.31	53.75	78.74	72.06	73.42

instructions in the packed-FP16 softmax (Appendix B). Second, multi-warp asynchronous TMA with proactive prefetching improves data movement during prefill (Section 4.1). Third, Stream-KV tile scheduling partitions variable-length decode work within the main kernel and merges partial results in place, eliminating FA3's separate prepare and combine kernels (Section 4.2).

End-to-end co-design. These components are tightly coupled by design. The linear-layer quantization scheme addresses the constraints imposed by FP8 computation. The KV-cache scheme enables INT4 keys to interact with FP8 queries while preserving the attention distribution. The kernel optimizations account for the dequantization and data-movement patterns introduced by W4A8KV4, which differ from the homogeneous data path assumed by FA3. Together, these techniques enable the central system-level goal of FServe: translating the memory savings of W4A8KV4 into FP8-level throughput on Hopper. Sections 5.2–5.4 evaluate this design end to end.

L Discussion: Portability and Extension to MoE

Portability to other architectures. Our evaluation focuses on the NVIDIA Hopper architecture, and we do not have access to Blackwell or AMD accelerators for a full implementation. We therefore discuss portability only at the design level rather than making experimentally verified claims. The Stream-KV scheduler itself does not rely on Hopper-specific scheduling assumptions, although its output-tile layout would need to be adapted to the tensor-core format of each target architecture. Multi-warp TMA prefetching relies on the TMA engine, which Blackwell retains, and is expected to remain applicable, while other kernel components may require architecture-specific modifications. In addition, we expect that multi-warp TMA fetching could be applied to the upcoming AMD CDNA 5 GPUs (Devices, 2026) too, which will include a Tensor Data Mover (TDM) unit similar to the TMA of NVIDIA for asynchronous tensor fetching to Local Data Share. The packed-FP16 softmax reordering in Section 4.1 is more closely tied to Hopper's register dependencies and would require architecture-specific tuning. On AMD accelerators, an alternative implementation would likely be needed. As discussed in Appendix A, the broader design principle of pairing 4-bit storage with a hardware-native floating-point execution path may also extend to architectures that provide native FP4 or microscaling support.

Extension to mixture-of-experts models. Our evaluation is limited to dense LLMs. At the design level, the quantization components may also be applicable to mixture-of-experts architectures because the linear- and attention-layer components are modular, and expert FFNs consist primarily of linear layers to which the two-step quantization of Section 3.1 could be applied. However, MoE models introduce additional considerations on quantization, such as expert routing and expert-specific execution behavior, which we do not evaluate in this work. We therefore leave a full evaluation on MoE models to future work.