DISAGGREGATED QUANTIZATION: SPECIALIZING LLM PREFILL AND DECODE

Anonymous authors
Paper under double-blind review

ABSTRACT

The prefill and decode phases of LLM inference reward different approaches to quantization: low-precision arithmetic accelerates prompt processing, while compact weights reduce memory traffic during generation. We propose "disaggregated quantization" (DQ), which specializes computational formats, weights and storage placement to both of these phases. Starting with format disaggregation, removing activation quantization specifically on decode improves accuracy on decode-heavy tasks without increasing inference cost. Moving to full disaggregation, training separate compute-native prefill weights accelerates prompt processing relative to weight-only inference while exceeding weight-only accuracy on both decode-heavy and prefill-heavy tasks for 2–3-bit quantization. To accommodate the extra prefill checkpoint on a single device, offloaded disaggregated prefill (ODP) streams its weights from SSD, amortizing loading over prompt length. Using released Qwen3.8-27B GGUF decoders, training an auxiliary NVFP4 prefill model in a fully-disaggregated fashion improves 1-bit accuracy by 32.5 points on MMLU-Pro and 35.3 on MMMU-Pro, without modifying the decode checkpoint. Combined with ODP, it delivers a $1.78\times$ prefill speedup over the weight-only baseline at 8K prompt length in `llama.cpp` at no device memory overhead. Additional PTQ validation includes models with up to 2.8T parameters.

1 INTRODUCTION

Historically, transformer models were first proposed for sequence-to-sequence tasks (Vaswani et al., 2023). In encoder–decoder transformers, the architecture for encoding and decoding is different. The encoder processes the input, while the decoder generates an output conditioned on its representations (Raffel et al., 2023). Cross-attention acts as a bridge between the two, allowing the two sub-networks to be trained toward the same output objective.

Modern decoder-only language models (Radford et al., 2019), however, treat every token in a sequence equally as both a target conditioned on preceding tokens and context for future tokens. Input processing and output generation consequently share the same parameters and architecture.

In instruction-following use-cases, the logical distinction reappears. A user supplies data or instructions, and the model produces an answer conditioned on them. Post-training reinforces these roles through structured interactions (Wei et al., 2022; Rafailov et al., 2024) or rewards for generated answers (Guo et al., 2025). This splits up inference into two different phases where prefill constructs the prompt key–value (KV) cache and decode consumes it while generating the response.

Prefill and decode workloads are different enough that large-scale deployment systems may run them on separate accelerators. Similarly, we show that input processing and generation can use distinct quantized representations while remaining jointly optimized for the response. We explore this through *disaggregated quantization* (DQ). DQ retains the pretrained attention architecture while specializing the linear computations and, optionally, their weights and their placement, to the two phases of LLM inference.

1.1 HARDWARE COST

At every linear layer, inference combines (1) loading model weights and activations from device memory (DRAM) into the compute units and (2) general matrix multiplication (GEMM) inside

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

them. For an $H \times H$ weight and $T \times H$ activations, it transfers $\mathcal{O}(H^2 + TH)$ elements and performs $\mathcal{O}(TH^2)$ arithmetic. Increasing the token count T therefore amortizes weight transfer over more computation. At batch one ($T = 1$), each weight contributes only one multiply-add, so weight loading dominates. In sufficiently long prefill workloads ($T \gg 1$), reuse across tokens makes the linear layers generally compute-bound.

These load profiles motivate (1) compressing weights to reduce memory traffic, admitting complex weight-only encodings (Frantar et al., 2023; Egiazarian et al., 2024; Tseng et al., 2024; 2025), and (2) quantizing weights and activations to hardware-supported compute formats such as INT4 (Ashkboos et al., 2023; 2024; Liu et al., 2025a) or NVFP4 (Egiazarian et al., 2026a; Chen et al., 2025b). A compact weight encoding, however, need not be a hardware-native compute format. A weight-only decode kernel can reconstruct values as it loads them for a matrix-vector product, still reducing weight traffic. It need not quantize activations either. Using arbitrary encoding with hardware-native quantized GEMM, however, requires both weight re-quantization and activation quantization into supported formats.

A common setup in efficient LLM inference is *disaggregated serving*, where separate prefill and decode model instances reside on different accelerators and exchange a prompt KV cache. Both instances, however, normally hold the same weights and execute the same computational schemes, with the majority of ongoing research instead focusing on communication (Qin et al., 2025) and scheduling (Hu et al., 2024).

This leaves a gap between how quantization interacts with hardware and how it is applied in existing serving systems. In this paper, we aim to generalize quantization schemes to multi-phase serving, expand disaggregation via format and weight specialization and, for the first time, extend disaggregated serving to local and edge use-cases.

1.2 Contributions

We introduce *disaggregated quantization* (DQ), a broad concept in which we treat quantization for prefill and decode separately. Splitting prefill and decode computation and weight formats yields a ladder of different schemes, each targeting a different axis of inference cost. We introduce the different schemes, and the tools to optimize networks for each. Our contributions are as follows:

1. **Quantization-aware distillation with disaggregation (QADD)** trains phase-specific pathways toward a common response objective in one forward-backward pass, supporting shared or separate master weights and prefill-only adaptation to a frozen decoder.

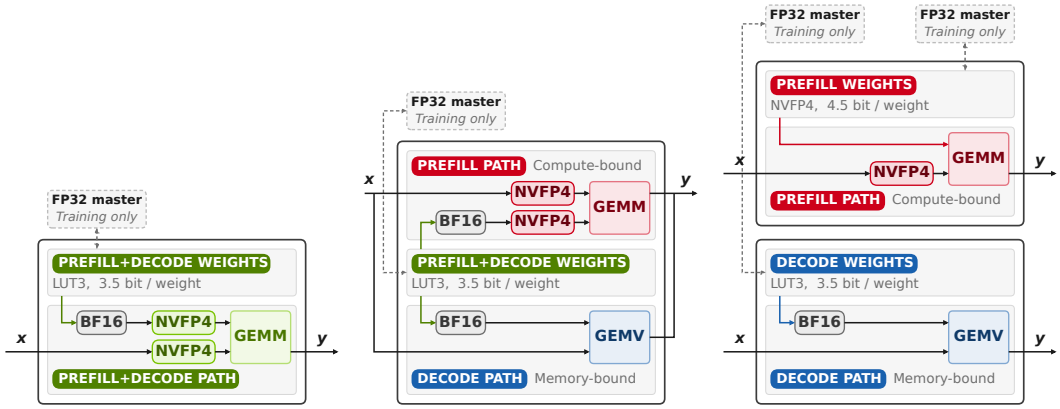
2. **Disaggregated quantization** is presented by three compounding schemes we introduce:

   (a) **Format disaggregation** combines quantized compute prefill with low-bitwidth weight-only decode for a common model checkpoint. Compared to phase-agnostic computations, this scheme improves accuracy primarily on decode-heavy tasks without increasing storage or inference costs.

   (b) **Full disaggregation** trains separate compute-native prefill weights in conjunction with low-bitwidth decode weights. The additional representational capacity of prefill weights is a net benefit for quality on both prefill-heavy and decode-heavy workloads, improving over both format disaggregation and weight-only serving.

   (c) **Offloaded disaggregated prefill (ODP)** streams prefill weights from SSD through buffers that reuse device memory and overlap loading with compute to avoid additional device memory occupation and, at longer context, hide loading overhead. On Qwen3.8-27B, our `llama.cpp` implementation delivers a $1.78\times$ time-to-first-token speedup over a weight-only baseline at 8K context while retaining all quality benefits of full disaggregation.

3. **Prefillers:** training fully-disaggregated NVFP4 prefill checkpoints to augment arbitrary frozen weight-only checkpoints via QADD. For 1-bit GGUF compression of Qwen3.8-27B, we show it more than doubling accuracy over weight-only inference, while simultaneously making prefill faster and keeping memory in check via ODP.



(a) **Non-disaggregated** quantized linear. Common model weights, same formats for the two phases.

(b) **Format-disaggregated** quantized linear. Common model weights, phase-specific formats.

(c) **Fully-disaggregated** quantized linear. Phase-specific model weights and formats.

Figure 1: Storage and computation schemes for various degrees of disaggregated quantization, using the LUT3 format as an example. This application of disaggregated quantization retains the prefill speed of NVFP4 and the decode speed of LUT3 while achieving higher accuracy (see Figure 4). Offloaded disaggregated prefill replaces full device residency of the extra prefill checkpoint with block buffers (see Section 2.5).

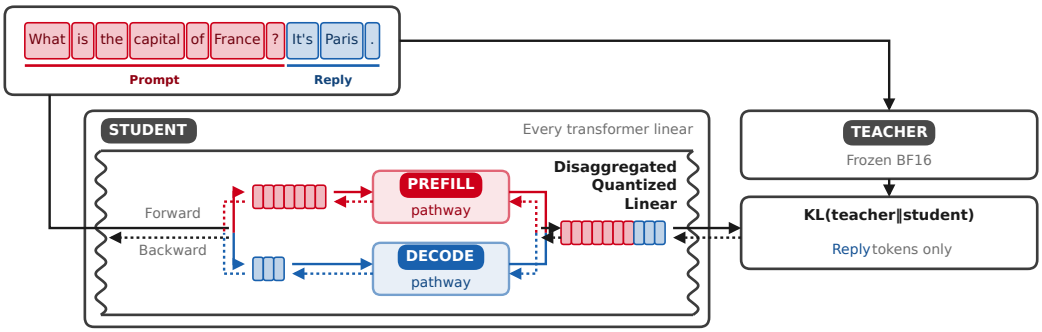


Figure 2: Quantization-aware distillation with disaggregation (QADD). The SFT assistant token mask, normally used only for loss, is also used to select the linear layer computational pathway.

2 Disaggregated Quantization

We first introduce QADD (Section 2.1), then subsequently disaggregate formats, weights and storage (Sections 2.2–2.5). The measured effect of these schemes on accuracy and inference cost is presented in Section 3.

2.1 Quantization-aware distillation with disaggregation

Quantization-aware distillation with disaggregation (QADD) builds on top of quantization-aware distillation (QAD) (Polino et al., 2018; Lee et al., 2025; Xin et al., 2026). It uses the SFT label mask to propagate the prefill/decode separation from post-training data into the quantized model layers. The mask, normally used for loss masking, now also selects the computational pathway: prompt (user turn) uses prefill, while response (assistant turn) uses decode. Although the distillation loss supervises only response targets, its gradients reach the prefill weights through the prompt keys and values consumed by decode. Both pathways are therefore trained toward the same response objective in one forward-backward pass (Figure 2). Implementation details are provided in Appendix A.

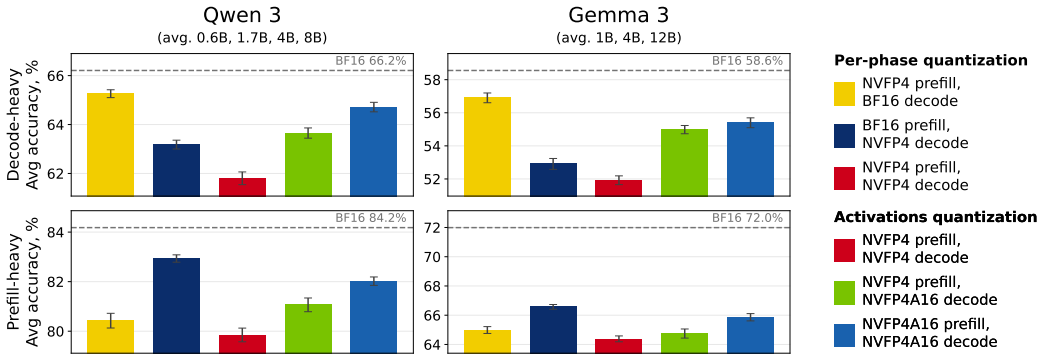


Figure 3: Family-mean accuracy of NVFP4-based formats after quantization-aware distillation on decode-heavy (top) and prefill-heavy (bottom) workloads. Decode-only NVFP4 quantization is more damaging than prefill-only quantization on decode-heavy tasks; the ordering reverses on prefill-heavy tasks. Format-disaggregated NVFP4 (green) combines NVFP4 prefill with NVFP4A16 decode and improves accuracy over uniform NVFP4 on both workloads without increasing weight storage or prefill cost. On decode-heavy tasks, it approaches the accuracy of NVFP4A16, which uses slower weight-only prefill.

2.2 Quantization sensitivity depends on the workload

Before turning to DQ formats, we first establish that different benchmarks interact with quantization of either phase differently, allowing us to monitor phase-specific accuracy effects. We do that by quantizing each phase to NVFP4 in isolation and gauging the effect on two distinct sets of benchmarks: decode-heavy reasoning benchmarks and prefill-heavy benchmarks with long prompts and short answers.

We find that on decode-heavy benchmarks, quantizing decode alone incurs 2–4$\times$ the accuracy loss of quantizing prefill alone on most models, and up to 7$\times$ on Gemma3-1B. On prefill-heavy tasks, prefill-only quantization incurs 1.1–4.1$\times$ the accuracy loss of decode-only quantization on seven of eight models (Figure 3, Figure 12).

With disaggregated quantization, we aim to improve the quality of both stages. Tracking quality on both prefill-heavy and decode-heavy evaluations, verified above, allows us to separate and quantify improvements per stage.

2.3 Format-disaggregated quantization reduces decode-phase error

NVFP4 quantizes weights and activations alike (Egiazarian et al., 2026a; Chen et al., 2025b), enabling fast prefill; its weight-only variant NVFP4A16 leaves activations unquantized, achieving higher quality but forfeiting the faster prefill computations.

We propose disabling activation quantization only on decode, yielding format-disaggregated NVFP4. It retains original NVFP4's storage and prefill costs while slightly accelerating memory-bound decode by skipping activation quantization (Table 1, Appendix C.2).

The same approach extends to arbitrary weight encodings and computational formats. Since 2–3-bit quantization has been shown to be Pareto-optimal in size-to-accuracy (Egiazarian et al., 2024; Liu et al., 2025b; Panferov et al., 2025), we also evaluate 2- and 3-bit scalar look-up table (LUT) weight encodings, that we refer to as "LUT2A16" and "LUT3A16" (Appendix A.3). To enable native low-precision computations on top of low-bitwidth weights, on-the-fly "autocast" re-quantizes them, along with activations, to NVFP4. We refer to these accelerated-compute formats as "LUT3" and "LUT2". The non-disaggregated scheme applies this autocast in both phases indiscriminately (Figure 1a); format disaggregation restricts it to prefill, combining native NVFP4 prefill computation with compact weight-only LUT decode (Figure 1b).

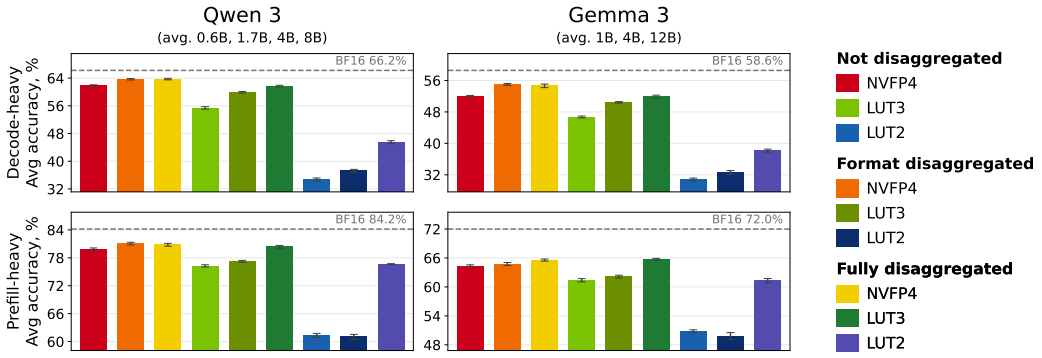


Figure 4: Family-mean accuracy of non-disaggregated, format-disaggregated and fully-disaggregated schemes with 2–4-bit decode weights and NVFP4 prefill compute after quantization-aware distillation. Format disaggregation primarily improves accuracy on decode-heavy tasks (top). Full disaggregation further improves low-bit accuracy on both workload types, with the largest gains over format disaggregation at 2-bit decode on prefill-heavy tasks.

Prefill still uses a re-quantized view of the low-bit decode weights, so its representation remains constrained by their compact encoding, which does not accelerate the NVFP4 computations used by prefill. This motivates giving prefill its own weights.

2.4 Fully-disaggregated quantization increases prefill-phase capacity

We propose training separate prefill weights, in a scheme we refer to as "fully-disaggregated quantization". Both prefill and decode weights start from the same unquantized model and are optimized together by QADD under their respective formats towards a common response objective, yielding a native NVFP4 prefill checkpoint and a separate weight-only decode checkpoint (Figure 1c). Each decode format is trained with its own prefill checkpoint, rather than reusing one across bitwidths.

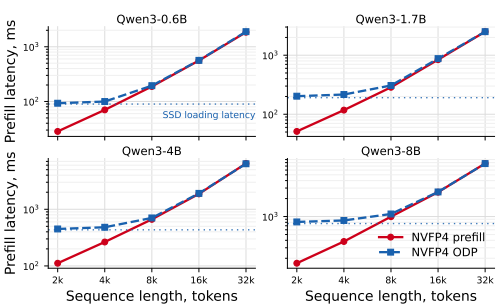
At inference, the prefill checkpoint produces the prompt keys and values at each layer. The decode checkpoint then attends to these representations. The cache retains the original model's layer and head dimensions, so separating the weights does not require any attention modifications. Speed-wise, full disaggregation enjoys the benefits of both NVFP4 prefill computations and weight-only low-bitwidth decode, cleanly combining the best of both worlds.

Full disaggregation mandates storing an additional prefill checkpoint, increasing total storage while preserving the decode weight footprint and speed. Datacenter disaggregation already stores a model instance per phase, but holding both on a single device can be prohibitive. Their residency requirements differ, however. Decode-only weights need high-bandwidth access during generation but are unused during prefill, while prefill weights are needed only during prompt processing and sit idle during decode. We exploit this duality next.

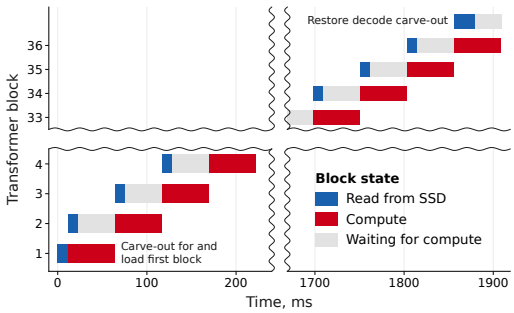
2.5 Offloaded disaggregated prefill

Once a prefill transformer block has produced its outputs, its weights are no longer needed for the rest of the assistant turn, even though its cached keys and values remain in use. *Offloaded disaggregated prefill* (ODP) therefore loads prefill weights from SSD block by block and reuses their device buffers as the context propagates through the network.

Prefill compute grows with context length, while loading the prefill checkpoint from SSD has a fixed cost, so the relative loading overhead decreases as prompts grow. On DGX Spark, compute overtakes loading around 8K context for all Qwen 3 models (Figure 5a). Two device block buffers suffice to overlap compute with loading: while one block processes the prompt, the next is loaded into the other buffer. Longer prompts leave more time for this transfer before the next block is needed, as shown in Figure 5b; short prompts can instead stall on loading. We obtain buffer space by carving-out an equally sized portion of decode weights, unused on prefill, and restoring it before generation. This makes ODP occupy no additional device weight memory at the cost of extra reads.



(a) NVFP4 prefill latency with and without ODP. Compute overtakes SSD loading at around 8K context and offloading overhead stays under 5% across Qwen 3 above 16K context.



(b) ODP pipeline schematic for Qwen 3 4B at 16K context length in NVFP4, scaled to measured loading and total latency. After a cold start on the first block, subsequent loads overlap with compute.

Figure 5: Latency effect (a) and pipelining scheme (b) of offloaded disaggregated prefill (ODP).

The same streaming principle, in theory, applies to separate input-processing networks, including encoders in encoder–decoder LLMs and prefill models connected to a decoder through a learned KV-cache adapter (Heo et al., 2026), replacing full device weight residency with streamed buffers. The same idea, however, does not seamlessly transfer to mixture-of-experts models, as the ratio of compute cost to loading cost grows with the fraction of active parameters, making loading considerably more expensive than compute up to extremely high context lengths. We, therefore, present ODP as mainly a tool for local deployment of dense LLMs.

2.6 Prefillers for arbitrary weight-only checkpoints

So far, we have jointly optimized the prefill and decode weights. In practice, however, a suitable quantized decoder may already exist, produced via advanced algorithms over complex encodings (Egiazarian et al., 2024; Tseng et al., 2024; van der Ouderaa et al., 2026) or released pre-quantized with closed-source or opaque data and algorithms (Gemma Team, 2026). As the result, it is possible that existing pre-quantized checkpoints either can't or don't need to be trained during QADD.

We extend full disaggregation to such checkpoints by training only their *prefillers*: prefill models specialized to particular pre-quantized frozen decoders. During training, the decoder uses its dequantized weights without updating them, while gradients propagate through its computations to the prefill pathway. The NVFP4 prefill weights thus adapt to the representations needed by the existing decoder. The decode quantization pipeline can remain a black box: we require its resulting checkpoint, not its training data or optimization algorithm. The resulting model retains the released checkpoint's compressed decode weights and execution pathway while enabling hardware-native NVFP4 prefill. ODP streams the prefiller without increasing device weight residency.

3 Experimental setup and large-scale validation

3.1 Experimental setup

Core QADD experiments. We minimize $\mathrm{KL}(p_{\text{teacher}} \| p_{\text{student}})$ against the frozen unquantized teacher on 100M tokens from the Tülu 3 (Lambert et al., 2025) SFT corpus. Uniform and disaggregated configurations use the same corpus, token budget and optimization schedule (Appendix A.1).

Instruction-tuned models. DQ requires a logical input/output separation, so we use models that have undergone post-training. Core experiments are performed on Qwen 3 (Yang et al., 2025) at 0.6B, 1.7B, 4B and 8B parameters, and Gemma 3 (Gemma Team, 2025) at 1B, 4B and 12B parameters.

Core QADD benchmarks. For decode-heavy evaluation, we use the generative versions of GSM8K (Cobbe et al., 2021), MATH-500 (Hendrycks et al., 2021) and MMLU-Pro (Wang et al.,

Table 1: Cost and accuracy for Qwen 3 and Gemma 3. Speedups and device allocations are for Qwen3-8B and Gemma3-12B. Accuracy (%) is averaged over 0.6B/1.7B/4B/8B for Qwen 3 and 1B/4B/12B for Gemma 3. DH denotes decode-heavy accuracy over three benchmarks (two reasoning modes for Qwen 3); PH denotes prefill-heavy RULER accuracy over 13 tasks at 4K/8K/16K/32K context. Both use the last five QADD checkpoints (Section 3.1). Prefill is measured at 16K context length on DGX Spark. Decode is measured end-to-end in vLLM as per-token latency at batch one.

Format	Qwen 3					Gemma 3				
	Prefill speedup	Decode speedup	Device GB	Acc. DH	Acc. PH	Prefill speedup	Decode speedup	Device GB	Acc. DH	Acc. PH
BF16	1.00x	1.00x	16.38	66.2	84.2	1.00x	1.00x	23.53	58.6	72.0
NVFP4A16	1.00x	2.93x	6.40	64.7	82.0	1.00x	3.27x	8.07	55.4	65.9
NVFP4	1.49x	2.86x	6.40	61.8	79.8	1.67x	3.18x	8.07	51.9	64.4
+Format disagg.	1.49x	2.93x	6.40	63.7	81.1	1.67x	3.27x	8.07	55.0	64.7
3-bit weight-only	1.00x	3.33x	5.53	61.5	79.8	1.00x	3.44x	6.72	51.6	64.1
LUT3	1.49x	3.23x	5.53	55.4	76.3	1.67x	3.37x	6.72	46.7	61.4
+Format disagg.	1.49x	3.33x	5.53	59.9	77.2	1.67x	3.44x	6.72	50.4	62.1
+Full disagg.	1.49x	3.33x	9.44	61.7	80.4	1.67x	3.44x	12.77	51.9	65.7
+ODP	1.47x	3.33x	5.53	61.7	80.4	1.58x	3.44x	6.72	51.9	65.7
2-bit weight-only	1.00x	3.82x	4.66	38.4	64.1	1.00x	4.15x	5.38	33.7	52.4
LUT2	1.49x	3.73x	4.66	34.8	61.3	1.67x	4.03x	5.38	30.8	50.8
+Format disagg.	1.49x	3.82x	4.66	37.2	61.0	1.67x	4.15x	5.38	32.6	49.8
+Full disagg.	1.49x	3.82x	8.57	45.5	76.6	1.67x	4.15x	11.43	38.2	61.3
+ODP	1.47x	3.82x	4.66	45.5	76.6	1.58x	4.15x	5.38	38.2	61.3

2024), with optional Qwen 3 reasoning both enabled and disabled. For prefill-heavy evaluation, we use RULER (Hsieh et al., 2024), whose long prompts and short answers complement these reasoning workloads. We evaluate its 13 tasks at 4K, 8K, 16K and 32K context lengths, with Qwen 3 reasoning disabled. The reported accuracy summaries use *real* disaggregated serving via vLLM (Kwon et al., 2023) and NIXL. Decode-heavy accuracy is averaged over the three benchmarks (times two reasoning modes for Qwen 3). Prefill-heavy scores are averaged over tasks and then context lengths. Per-model accuracy further averages the last five checkpoints of each QADD run, where performance plateaus. A number quoted for a family, such as "on Qwen 3", is the unweighted mean over that family's model sizes. Error bars describe two standard deviations over temporal averaging within one training run, not uncertainty across independent runs (Appendix A.5,D).

Latency. We measure end-to-end batch-one per-output-token latency through vLLM and prefill transformer-stack latency with a custom stack built from vLLM kernels. LUT2 and LUT3 decode use custom weight-only kernels. All measurements are performed on DGX Spark (Appendix C).

3.2 Main QADD-based results

Format-disaggregated quantization. Accuracy-wise, on decode-heavy tasks, format disaggregation boosts mean accuracy over the non-disaggregated scheme by 1.9 and 3.1 points for NVFP4, 4.5 and 3.7 points for LUT3, 2.5 and 1.8 points for LUT2 on Qwen 3 and Gemma 3, respectively (respective improvement for Qwen 3 and Gemma 3 is implied throughout this subsection). On prefill-heavy tasks, however, this decoding-phase optimization has an effect of less than 1.3 points for all considered model families and formats (Figure 4, Table 1).

Speed-wise, on prefill, format disaggregation uses the same accelerated NVFP4 computations as the non-disaggregated scheme, with up to $1.49\times$ and $1.67\times$ speedup over BF16 on Qwen3-8B and Gemma3-12B. On decode, format disaggregation is 2–3% faster than the non-disaggregated scheme by virtue of skipping activations quantization (Table 1, Table 10, Figure 9).

That justifies format disaggregation as a plug-in replacement for non-disaggregated inference that boosts accuracy on decode-heavy tasks while retaining or improving all speed and storage costs for single-user serving. It is most useful when device memory is scarce and interactivity of the original model needs to be fully preserved.

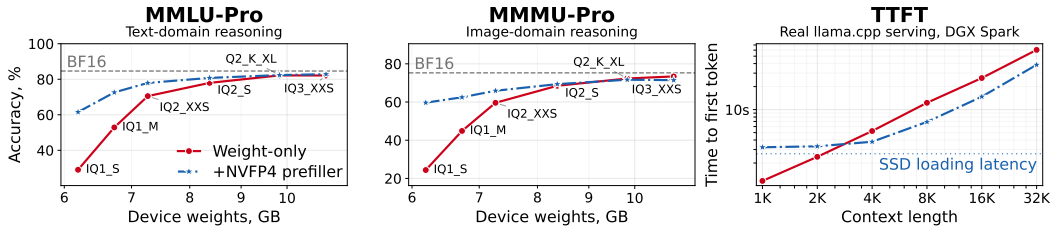


Figure 6: NVFP4 prefillers for off-the-shelf Qwen3.8-27B GGUF decoders. Left and middle: training the prefiller more than doubles IQ1_S accuracy on both benchmarks while leaving the decode checkpoint unchanged. Right: measured time to first token in `llama.cpp`, comparing ODP with weight-only IQ1_S. ODP is faster from 4K context, reaching $1.78\times$ speedup at 8K.

Fully-disaggregated quantization and ODP. Accuracy-wise, full disaggregation improves both decode-heavy and prefill-heavy performance over non-disaggregated formats. For the former, it yields 6.3 and 5.2 points for LUT3, 10.7 and 7.4 points for LUT2. For the latter, it gains 4.1 and 4.3 points for LUT3, 15.3 and 10.5 points for LUT2. For 2–3-bit models, the improvement is noticeable over not only non-disaggregated and format-disaggregated serving, but also weight-only serving: fully-disaggregated LUT2 substantially outperform LUT2A16, by 4.5–12.5 points, while also delivering faster prefill computations (Figure 4, Table 1).

Cost-wise, ODP negates the device memory overhead of prefill weights while changing the prefill latency landscape: flat SSD-bandwidth-bound time-to-first-token (TTFT) for short sequences is followed by a diminishing overhead for compute-constrained longer sequences. At context length above 16K, offloading increases resident NVFP4 prefill latency by less than 5% on Qwen 3 and 8% on Gemma 3, coming from a cold start on the first block and synchronization logic. At 16K, the streamed transformer stack remains $1.47\times$ faster than BF16 on Qwen3-8B and $1.58\times$ on Gemma3-12B. In this regime, ODP retains most of the resident NVFP4 prefill speedup without additional device weight memory. Decode speedup remains unchanged relative to format disaggregation as both use native weight encoding there (Table 1, Table 10).

Thus, fully-disaggregated quantization is preferable for both low-concurrency disaggregated serving, when two checkpoints are resident on accelerators anyway and decode is still memory-bound, and for local single-user serving, via ODP. The latter, however, is not as useful for MoE models, and when TTFT on short sequences is critical.

Prefillers for pre-quantized Qwen3.8-27B. For the prefillers experiments, we scale our QADD setup to Qwen3.8-27B (Qwen Team, 2026). We use eight openly-available pre-quantized GGUF (Gerganov & contributors, 2023) checkpoints released by Unsloth (Daniel Han & team, 2023), including vector-quantized formats such as IQ2_XXS. We evaluate MMLU-Pro (Wang et al., 2024) for text reasoning and MMMU-Pro (Yue et al., 2025) for visual reasoning. We report one complete evaluation per format at the end of the 95M-token QADD training on text-only reasoning traces from the unquantized model (Appendix A.2).

Accuracy-wise, training an NVFP4 prefiller improves 1-bit decoder accuracy by 32.5 points on MMLU-Pro and 35.3 on MMMU-Pro, more than doubling its weight-only accuracy on both benchmarks. The gains at 2-bit decode are around 7.4 and 6.3 points and diminish at higher bitwidths. At 3-bit decode, an NVFP4 prefiller leads to slight performance degradation instead. Figure 6 summarizes the gains across formats and benchmarks. The QADD corpus contains no multi-modal examples, yet the low-bit accuracy gains transfer strongly to visual reasoning on MMMU-Pro. For the lowest-bit decoders, learned prefillers also substantially outperform RTN prefill at the same NVFP4 precision, showing that training matters beyond the choice of format (Appendix B.4). Appendix B.5 reports generation-length and truncation measurements.

Speed-wise, adding an NVFP4 prefiller through our custom `llama.cpp` (Gerganov & contributors, 2023) extension reduces time to first token from 12.27 to 6.90 seconds at 8K context, a $1.78\times$ speedup over the weight-only baseline. Across the measured 4K–32K contexts, the speedup ranges from $1.38\times$ to $1.78\times$, while SSD loading makes ODP slower at shorter prompts (Figure 6). The deployment trade-offs are otherwise identical to those of the underlying full disaggregation.

Table 2: Accuracy over text (MMLU-Pro) and image (MMMU-Pro) reasoning benchmarks for SOTA LLMs, as well as gains from 4-bit format disaggregation. Gains highlighted in **bold** are statistically significant (per-comparison $p < 0.05$). †: FP8 reference instead of BF16. ‡: MXFP4 instead of NVFP4, with no BF16 checkpoint available.

Model	MMLU-Pro					MMMU-Pro				
	BF16	W4A16	W4A4 disaggregation			BF16	W4A16	W4A4 disaggregation		
			None	Format	Δ			None	Format	Δ
Qwen3.8-27B	84.58	82.33	81.47	81.83	**+0.35**	74.78	71.16	69.84	70.46	+0.62
Gemma-4-31B	84.92	84.51	84.07	84.17	+0.10	66.84	65.36	64.81	65.94	**+1.13**
Muse-Glimmer-30B	75.43	76.80	75.12	76.04	**+0.92**	72.92	71.81	70.38	71.34	**+0.97**
Gemma-4-26B	82.40	81.01	79.74	80.52	**+0.78**	63.66	60.95	59.05	59.68	+0.64
Nemotron-3-120B	83.01	82.78	82.66	82.68	+0.02	-	-	-	-	-
Nemotron-3-550B	86.71	86.47	86.51	86.37	-0.14	-	-	-	-	-
Qwen3.8-2.4T†	88.99	84.73	84.28	84.68	**+0.40**	-	-	-	-	-
Kimi-K3-2.8T‡	-	85.94	85.78	85.57	-0.21	-	79.93	79.18	79.71	+0.53

3.3 Large-scale validation with PTQ

Scaling further up, we apply format disaggregation to eight text and multi-modal models up to 2.8T parameters. We use NVFP4 PTQ without retraining, testing the most straightforward intervention that is disabling activation quantization only on decode. We evaluate on MMLU-Pro and MMMU-Pro, using paired per-item tests with four evaluation repeats (Appendix A.5). We evaluate Qwen 3.8 (Qwen Team, 2026), Gemma 4 (Gemma Team, 2026), Muse Glimmer (Meta, 2026), Nemotron 3 (NVIDIA, 2025) and Kimi-K3 (Kimi Team, 2026). Format disaggregation at 4-bit improves point estimates in 11 of 13 model–benchmark combinations, with 6 significant gains at $\alpha = 0.05$ and no significant degradations (Table 2).

4 Related work

Existing work on interaction between inference phases and quantization balances prefill quality against memory-bound decode through phase-specific weight precision (Chen et al., 2025a), accelerates prefill with low-precision compute while retaining BF16 decode (Lu et al., 2026; Wei et al., 2026), or fine-tunes task-specific prefill modules around a frozen quantized decoder (Woo et al., 2026). Forys et al. (2026) allow prefill and decode to be quantized independently in a disaggregated serving simulator. These works establish the value of phase-specific precision and prefill adaptation. Disaggregated quantization takes this idea further to combine hardware-native prefill with compressed weight-only decode in a unifying approach that can share master weights or maintain separate weights specialized to each phase, supporting both joint optimization and prefill-only adaptation to a frozen decoder. Offloading-wise, FlexGen (Sheng et al., 2023) amortizes weight transfers through large batches for both phases. ODP, in contrast, streams only the additional prefill checkpoint, amortizing loading over prompt length while decode weights remain resident.

5 Conclusion and Limitations

DQ introduces a phase-aware approach to co-designing quantization formats and model serving. Prefill and decode representations can be chosen for their distinct hardware costs and trained to work together toward a common response objective. This separation also changes local serving: weights needed only during prompt processing can reside on SSD between requests, making room for a specialized prefill model without increasing device weight residency.

Our evaluations cover decode-heavy reasoning and single-turn prefill-heavy tasks, with batch-one decode and prefill timings on DGX Spark. We do not evaluate highly batched performance or multi-turn and agentic behavior. In multi-turn use, cached assistant tokens retain decode-produced KV entries, while rebuilding their cache through prefill can produce different representations for the same token history. Robustness to this cache-policy dependence remains untested.

AI USE STATEMENT

In this work, we used generative AI tools to implement methods, including PyTorch-based training frameworks, CUDA kernels, evaluation pipelines and benchmarking code. We also used them to provide feedback on research methodology and experiments, refine hypotheses and interpret results. Additional uses included literature search and comparison, drafting and editing manuscript text and creating scientific figures.

We have reviewed the AI-assisted work. Generated code was tested through evaluations on real inference engines, and generated CUDA kernels were tested against hand-crafted baselines. Numerical claims were checked against saved evaluation outputs and measurements, as well as official performance figures, and literature comparisons were checked against the cited papers. We reviewed and revised all AI-assisted text and code. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

All model hyperparameters and training and evaluation configurations are described in Appendix A. Hardware setups and benchmarking protocols are specified in Appendix C. The newly produced training, evaluation and benchmarking suites are attached as supplementary materials. All models and datasets used in this work are publicly available.

REFERENCES

Saleh Ashkboos, Ilia Markov, Elias Frantar, Tingxuan Zhong, Xincheng Wang, Jie Ren, Torsten Hoefler, and Dan Alistarh. Quik: Towards end-to-end 4-bit inference on generative large language models, 2023. URL https://arxiv.org/abs/2310.09259.

Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L. Croci, Bo Li, Pashmina Cameron, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. Quarot: Outlier-free 4-bit inference in rotated llms, 2024. URL https://arxiv.org/abs/2404.00456.

Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation, 2013. URL https://arxiv.org/abs/1308.3432.

Patrick Blumenberg, Thomas Graave, and Tim Fingscheidt. Improving block-wise llm quantization by 4-bit block-wise optimal float (bof4): Analysis and variations, 2025. URL https://arxiv.org/abs/2505.06653.

Hao Mark Chen, Fuwen Tan, Alexandros Kouris, Royson Lee, Hongxiang Fan, and Stylianos I. Venieris. Progressive mixed-precision decoding for efficient llm inference, 2025a. URL https://arxiv.org/abs/2410.13461.

Mengzhao Chen, Meng Wu, Hui Jin, Zhihang Yuan, Jing Liu, Chaoyi Zhang, Yunshui Li, Jie Huang, Jin Ma, Zeyue Xue, Zhiheng Liu, Xingyan Bin, and Ping Luo. Int v.s. fp: A comprehensive study of fine-grained low-bit quantization formats, 2025b. URL https://arxiv.org/abs/2510.25602.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021. URL https://arxiv.org/abs/2110.14168.

Jack Cook, Hyemin S. Lee, Kathryn Le, Junxian Guo, Giovanni Traverso, Anantha P. Chandrakasan, and Song Han. Adaptive block-scaled data types, 2026. URL https://arxiv.org/abs/2603.28765.

Michael Han Daniel Han and Unsloth team. Unsloth, 2023. URL https://github.com/unslothai/unsloth.

Vage Egiazarian, Andrei Panferov, Denis Kuznedelev, Elias Frantar, Artem Babenko, and Dan Alistarh. Extreme compression of large language models via additive quantization, 2024. URL https://arxiv.org/abs/2401.06118.

Vage Egiazarian, Roberto L. Castro, Denis Kuznedelev, Andrei Panferov, Eldar Kurtic, Shubhra Pandit, Alexandre Marques, Mark Kurtz, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Bridging the gap between promise and performance for microscaling fp4 quantization, 2026a. URL https://arxiv.org/abs/2509.23202.

Vage Egiazarian, Erik Schultheis, Andrei Panferov, Earl Killian, Torsten Hoefler, and Dan Alistarh. Grid games: The power of multiple grids for quantizing large language models, 2026b. URL https://arxiv.org/abs/2605.12327.

Przemyslaw Forys, Haoran Wu, Can Xiao, Jiayi Nie, Tony Liu, Rika Antonova, Timothy Jones, Robert Mullins, Wayne Luk, Aaron Zhao, and George A. Constantinides. When does disaggregation pay? simulating prefill–decode–attention–ffn specialization for agentic llm inference, 2026. URL https://arxiv.org/abs/2608.03741.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers, 2023. URL https://arxiv.org/abs/2210.17323.

Elias Frantar, Roberto L. Castro, Jiale Chen, Torsten Hoefler, and Dan Alistarh. Marlin: Mixed-precision auto-regressive parallel inference on large language models, 2024. URL https://arxiv.org/abs/2408.11743.

Gemma Team. Gemma 3 technical report, 2025. URL https://arxiv.org/abs/2503.19786.

Gemma Team. Gemma 4 technical report, 2026. URL https://arxiv.org/abs/2607.02770.

Georgi Gerganov and contributors. llama.cpp: Llm inference in c/c++, 2023. URL https://github.com/ggml-org/llama.cpp.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Honghui Ding, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jingchang Chen, Jingyang Yuan, Jinhao Tu, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaichao You, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen

Zhang. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638, Sept 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-09422-z. URL http://dx.doi.org/10.1038/s41586-025-09422-z.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset, 2021. URL https://arxiv.org/abs/2103.03874.

Taekyung Heo, Rasoul Shafipour, Ritchie Zhao, Maximilian Golub, Mohammad Mahdi Kamani, Ritika Borkar, Makesh Tarun Chandran, Pantea Zardoshti, and Bita Darvish Rouhani. Cross-model kv cache transfer in llm families: A closed-form linear mapping for prefill reuse, 2026. URL https://arxiv.org/abs/2608.03893.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. Ruler: What's the real context size of your long-context language models?, 2024. URL https://arxiv.org/abs/2404.06654.

Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. Inference without interference: Disaggregate llm inference for mixed downstream workloads, 2024. URL https://arxiv.org/abs/2401.11181.

Kimi Team. Kimi k3: Open frontier intelligence, 2026. URL https://arxiv.org/abs/2607.24653.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention, 2023. URL https://arxiv.org/abs/2309.06180.

Jonas Kübler, Kailash Budhathoki, Matthäus Kleindessner, Xiong Zhou, Junming Yin, Ashish Khetan, and George Karypis. When llms get significantly worse: A statistical approach to detect model degradations, 2026. URL https://arxiv.org/abs/2602.10144.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V. Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, Yuling Gu, Saumya Malik, Victoria Graf, Jena D. Hwang, Jiangjiang Yang, Ronan Le Bras, Oyvind Tafjord, Chris Wilhelm, Luca Soldaini, Noah A. Smith, Yizhong Wang, Pradeep Dasigi, and Hannaneh Hajishirzi. Tulu 3: Pushing frontiers in open language model post-training, 2025. URL https://arxiv.org/abs/2411.15124.

Jung Hyun Lee, Seungjae Shin, Vinnam Kim, Jaeseong You, and An Chen. Unifying block-wise ptq and distillation-based qat for progressive quantization toward 2-bit instruction-tuned llms, 2025. URL https://arxiv.org/abs/2506.09104.

Zechun Liu, Changsheng Zhao, Igor Fedorov, Bilge Soran, Dhruv Choudhary, Raghuraman Krishnamoorthi, Vikas Chandra, Yuandong Tian, and Tijmen Blankevoort. Spinquant: Llm quantization with learned rotations, 2025a. URL https://arxiv.org/abs/2405.16406.

Zechun Liu, Changsheng Zhao, Hanxian Huang, Sijia Chen, Jing Zhang, Jiawei Zhao, Scott Roy, Lisa Jin, Yunyang Xiong, Yangyang Shi, Lin Xiao, Yuandong Tian, Bilge Soran, Raghuraman Krishnamoorthi, Tijmen Blankevoort, and Vikas Chandra. Paretoq: Improving scaling laws in extremely low-bit llm quantization, 2025b. URL https://arxiv.org/abs/2502.02631.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019. URL https://arxiv.org/abs/1711.05101.

Haiquan Lu, Zigeng Chen, Gongfan Fang, Xinyin Ma, and Xinchao Wang. Mix-quant: Quantized prefilling, precise decoding for agentic llms, 2026. URL https://arxiv.org/abs/2605.20315.

Meta. Muse Glimmer, July 2026. URL https://huggingface.co/meta-models/Muse-Glimmer-30B.

NVIDIA. Nvidia nemotron 3: Efficient and open intelligence, 2025. URL https://arxiv.org/abs/2512.20856.

Andrei Panferov, Jiale Chen, Soroush Tabesh, Roberto L. Castro, Mahdi Nikdan, and Dan Alistarh. Quest: Stable training of llms with 1-bit weights and activations, 2025. URL https://arxiv.org/abs/2502.05003.

Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model compression via distillation and quantization, 2018. URL https://arxiv.org/abs/1802.05668.

Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: A kvcache-centric disaggregated architecture for llm serving, 2025. URL https://arxiv.org/abs/2407.00079.

Qwen Team. Qwen3.8-Max: A new bar for coding and cowork, August 2026. URL https://qwen.ai/blog?id=qwen3.8.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019. URL https://api.semanticscholar.org/CorpusID:160025533.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model, 2024. URL https://arxiv.org/abs/2305.18290.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2023. URL https://arxiv.org/abs/1910.10683.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu, 2023. URL https://arxiv.org/abs/2303.06865.

Albert Tseng, Jerry Chee, Qingyao Sun, Volodymyr Kuleshov, and Christopher De Sa. Quip#: Even better llm quantization with hadamard incoherence and lattice codebooks, 2024. URL https://arxiv.org/abs/2402.04396.

Albert Tseng, Qingyao Sun, David Hou, and Christopher De Sa. Qtip: Quantization with trellises and incoherence processing, 2025. URL https://arxiv.org/abs/2406.11235.

Tycho F. A. van der Ouderaa, Mart van Baalen, Paul Whatmough, and Markus Nagel. Leech lattice vector quantization for efficient llm compression, 2026. URL https://arxiv.org/abs/2603.11021.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023. URL https://arxiv.org/abs/1706.03762.

Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, Tianle Li, Max Ku, Kai Wang, Alex Zhuang, Rongqi Fan, Xiang Yue, and Wenhu Chen. Mmlu-pro: A more robust and challenging multi-task language understanding benchmark, 2024. URL https://arxiv.org/abs/2406.01574.

Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. Finetuned language models are zero-shot learners, 2022. URL https://arxiv.org/abs/2109.01652.

Zhixiang Wei, Yun Wang, James Yen, Mingyuan Xia, and Zhengwei Qi. Hbm is not all you need: Efficient disaggregated llm serving across memory-heterogeneous accelerators, 2026. URL https://arxiv.org/abs/2606.29986.

Sunghyeon Woo, Ahreum Seo, Jaegwang Lee, Jaeeun Kil, Hanbae Seo, Joonghoon Kim, Baeseong Park, Se Jung Kwon, and Dongsoo Lee. Sun: Shared use of next-token prediction for efficient multi-llm disaggregated serving, 2026. URL https://arxiv.org/abs/2603.02599.

Meng Xin, Sweta Priyadarshi, Jingyu Xin, Bilal Kartal, Aditya Vavre, Asma Kuriparambil Thekkumpate, Zijia Chen, Ameya Sunil Mahabaleshwarkar, Ido Shahaf, Akhiad Bercovich, Kinjal Patel, Suguna Varshini Velury, Chenjie Luo, Zhiyu Cheng, Jenny Chen, Chen-Han Yu, Wei Ping, Oleg Rybakov, Nima Tajbakhsh, Oluwatobi Olabiyi, Dusan Stosic, Di Wu, Song Han, Eric Chung, Sharath Turuvekere Sreenivas, Bryan Catanzaro, Yoshi Suhara, Tijmen Blankevoort, and Huizi Mao. Quantization-aware distillation for nvfp4 inference accuracy recovery, 2026. URL https://arxiv.org/abs/2601.20088.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Xiang Yue, Tianyu Zheng, Yuansheng Ni, Yubo Wang, Kai Zhang, Shengbang Tong, Yuxuan Sun, Botao Yu, Ge Zhang, Huan Sun, Yu Su, Wenhu Chen, and Graham Neubig. Mmmu-pro: A more robust multi-discipline multimodal understanding benchmark, 2025. URL https://arxiv.org/abs/2409.02813.

A Training and model hyper-parameters

A.1 QADD setup

Weights and activations are fake-quantized under straight-through estimation (Bengio et al., 2013), with FP32 master weights updated by AdamW (Loshchilov & Hutter, 2019).

Hyper-parameters. Table 3 lists common hyper-parameters for the core Qwen 3 and Gemma 3 QADD runs. The Qwen3.8-27B setup is described in Appendix A.2. Table 4 lists per-model parallelization hyper-parameters.

Table 3: QADD optimization hyperparameters, identical across all formats and models in the core Qwen 3 and Gemma 3 experiments.

Objective	$\mathrm{KL}(p_{\text{teacher}} \| p_{\text{student}})$, teacher frozen in BF16
Corpus	Tülu 3 SFT, 100M non-padding tokens, mean sequence length 638
Sequence length	2048
Global batch size	64 sequences
Steps	≈2450
Optimizer	AdamW, $\beta = (0.9, 0.95)$, $\epsilon = 10^{-8}$
Learning rate	3×10^{-6}, constant
Warmup	100 steps
Weight decay	0.1
Gradient clipping	1.0, per-DDP-rank
Master weights	FP32; straight-through estimator to the quantized weight
Compute precision	BF16 autocast; FP32 residual stream
Parallelism	DDP, ZeRO-2, pipeline parallelism

Phase masking. The prefill/decode assignment of each input position is read from the SFT labels: positions with an ignored label use prefill, while assistant positions use decode. A per-position mask selects the computational pathway in each quantized linear layer. The loss uses the causal shift,

Table 4: Data parallelism (DP), pipeline parallelism (PP) and gradient accumulation setup per model and format class.

Model / format	B300 GPUs	PP	DP	Micro-batch	Accum.
Qwen 3 0.6B / 1.7B / 4B, all formats	8	1	8	8	1
Qwen3-8B, single-master formats	8	1	8	8	1
Qwen3-8B, fully-disaggregated formats	16	2	8	8	1
Gemma 3 270m / 1b / 4b, all formats	8	1	8	8	1
Gemma-3-12b, single-master formats	16	1	16	4	1
Gemma-3-12b, fully-disaggregated formats	32	2	16	4	1
Qwen3.8-27B, frozen-decode	64	2	32	1	1

Table 5: Qwen3.8-27B accuracy (%) with released weight-only decoders and their NVFP4 prefiller trained by QADD. All eight formats are included, including IQ3_S and Q3_K_XL omitted from Figure 6. Fully-disaggregated results use step 980; each score is one complete benchmark evaluation, without checkpoint or repeat averaging. Δ is the percentage-point change from weight-only, computed before rounding. BF16 is the unquantized reference.

Decode format	MMLU-Pro			MMMU-Pro		
	Weight-only	Full disagg.	Δ	Weight-only	Full disagg.	Δ
BF16	84.62	–	–	75.26	–	–
IQ1_S	29.04	61.54	+32.50	24.39	59.65	+35.26
IQ1_M	52.88	72.59	+19.71	44.86	62.49	+17.63
IQ2_XXS	70.50	77.93	+7.42	59.60	65.90	+6.30
IQ2_S	77.96	80.76	+2.80	68.50	69.36	+0.87
Q2_K_XL	82.16	82.35	+0.18	72.31	71.68	-0.64
IQ3_XXS	82.11	82.84	+0.72	73.41	71.45	-1.97
IQ3_S	83.65	83.02	-0.63	74.74	71.97	-2.77
Q3_K_XL	84.25	83.64	-0.62	74.28	72.31	-1.97

scoring the hidden state at t against the response target at $t+1$, so the final prompt position uses the prefill pathway while predicting the first response token. Prefill weights receive gradients through this boundary prediction and through the prompt keys and values attended to by later response positions. Thus, restricting supervision to response targets still trains both pathways.

Training resources. Separate-weight configurations maintain optimizer state for both sets of master weights. Table 4 reports hardware allocations. The training corpus, token budget and optimization schedule are not changed between experiments.

A.2 QADD WITH A FROZEN QUANTIZED DECODER AT 27B

The experiments in Section 2.6 use QADD to train an NVFP4 prefiller for each externally quantized decoder. Unlike jointly trained full disaggregation, this setup requires trainable master weights, parameter gradients and optimizer state only for the prefill pathway. The decoder remains in memory during training, using its dequantized BF16 weights, but its parameters are neither updated nor included in the exported prefill checkpoint.

Model. Qwen3.8-27B has 64 transformer blocks: 48 use gated linear attention, with full attention in every fourth block. The input embeddings and output projection are untied, and the model includes a vision tower and a multi-modal adapter. Table 6 lists its dimensions.

Decode checkpoints. We use eight publicly released Unsloth GGUF checkpoints, spanning nominal 1–3-bit formats: `IQ1_S`, `IQ1_M`, `IQ2_XXS`, `IQ2_S`, `Q2_K_XL`, `IQ3_XXS`, `IQ3_S` and `Q3_K_XL`. Each checkpoint is dequantized to BF16 and frozen, while a separate NVFP4 W4A4 prefill model is trained for each decoder.

Trainable and shared parameters. We train the prefill copies of the quantized linear weights and text-stack normalization scales, leaving their decode counterparts fixed. Embeddings and the

`lm_head` are shared between phases and frozen. The narrow linear-attention gate projections `in_proj_a` and `in_proj_b` are also shared and frozen in BF16: they control the recurrent update, and quantizing them destabilized training. The recurrence parameters $A_{\log}$ and time-step biases likewise remain shared and frozen, preserving the dynamics used by the decode engine. The multi-token-prediction head, conceptually unnecessary for prefill, is not used in this setup and is omitted from export. Exports contain only the prefill checkpoint and the multi-modal adapter.

Corpus. Tülu 3 lacks explicit reasoning traces, so the chat template used here inserts an empty thinking block before each answer. For this reasoning-heavy model, we instead distill on `faunix/Qwen3.8-27B-Distillation-40K`, a public corpus of 40K traces generated by the same BF16 model as the teacher. Examples requiring tool calls or lacking a nonempty reasoning trace or final answer are discarded (559 of $40{,}000$). No image or otherwise multi-modal inputs are present in the mix.

Sequence length and training budget. The median tokenized sequence and prompt lengths are 3172 and 111 tokens, respectively. We therefore increase the sequence-length limit to 8192 and reduce the global batch size to 32. Examples are dropped rather than truncated, since truncation can remove important milestones such as closure of the reasoning stage. The filtered corpus contains $32{,}608$ sequences and approximately 95M tokens, which corresponds to 980 training steps.

Benchmark overlap. The training prompts originate from twelve public corpora, so we check their overlap with the evaluation benchmarks. Among the $12{,}032$ MMLU-Pro items, we find one exact prompt match, two matches after normalization, and three with token-shingle Jaccard similarity of at least 0.3. These matches concern mathematics problems also present in the corpus's math sources. We find no MMMU-Pro prompt matches under these checks. This does not establish that the corpus is generally benchmark-free: its sources include BIG-Bench Hard and OlympiadBench, while its math sources contain MATH items. We therefore do not use those three benchmarks to evaluate this setup.

A.3 Quantization formats

Following best practices in grid design and micro-scaling quantization (Blumenberg et al., 2025; Cook et al., 2026; Egiazarian et al., 2026b), we tune our scalar LUT quantization scheme to a Gaussian prior and utilize two-level scaling.

NVFP4-esque two-level scaling. Tensors are split into groups of 16 elements along the contraction dimension. Each tensor carries one FP32 global scale $s = \max|W|/(6 \cdot 448)$, and each block an FP8-E4M3 scale, so a block's extreme element normalizes to approximately ± 6 and the block scales stay inside E4M3's range. Layers that an inference engine fuses ($q/k/v$ into `qkv_proj`, gate/up into `gate_up_proj`) share one global scale, derived from the group-wise maximum.

NVFP4. E2M1 elements on the magnitude grid $\{0, 0.5, 1, 1.5, 2, 3, 4, 6\}$ with sign, blocks of 16, FP8-E4M3 block scales and one FP32 per-tensor global scale. W4A4 quantizes activations with a static per-tensor input scale from a running-maximum observer, matching what the serving engine applies at inference.

LUT formats. The weight-only formats use a scalar look-up-table encoding with the same two-level scaling. The grids are asymmetric and contain zero, and the per-block scale now, as opposed to native NVFP4, absorbs the sign of the block's maximum-magnitude element, so after normalization that element is always $+6$ and the value distribution is asymmetric. The grids are

$$\text{LUT3A16 (8 levels):} \quad \{-4.7038,\ -2.8698,\ -1.3696,\ 0,\ 1.2204,\ 2.5285,\ 4.0473,\ 6\}$$
$$\text{LUT2A16 (4 levels):} \quad \{-3.6517,\ 0,\ 2.5227,\ 6\}$$

These grids were optimized to minimize the expected quadratic error over $\mathcal{N}(0; 1)$ samples scaled with the aforementioned 2-level scaling, constrained to contain 0 and $+6$.

Storage cost. A LUTb weight costs b bits per element plus one FP8 scale per 16 elements, i.e. $b+0.5$ bits per element amortized; NVFP4 costs 4.5 bits on the same accounting. The fully-disaggregated

formats store two checkpoints and therefore pay both. ODP keeps the additional checkpoint on SSD without increasing device weight residency (Section 2.5).

The `lm_head` is left unquantized in every format.

A.4 MODELS

Table 6: Inner shapes of the instruction-tuned models used in this work.

Model	Layers	d_{model}	d_{ffn}	Vocabulary	Tied embeddings
Qwen3-0.6B	28	1024	3072	151936	yes
Qwen3-1.7B	28	2048	6144	151936	yes
Qwen3-4B	36	2560	9728	151936	yes
Qwen3-8B	36	4096	12288	151936	no
Gemma-3-270m	18	640	2048	262208	yes
Gemma-3-1b	26	1152	6912	262208	yes
Gemma-3-4b	34	2560	10240	262208	yes
Gemma-3-12b	48	3840	15360	262208	yes
Qwen3.8-27B	64	5120	17408	248320	no

Per-model hyper-parameters are shown in Table 6.

A.5 EVALUATION PROTOCOL

Decode-heavy QADD results. We run evaluations with the following parameters: GSM8K (5-shot), MATH-500 (4-shot) and MMLU-Pro (5-shot). For Qwen 3, each is evaluated with reasoning enabled and disabled. Gemma provides no lever to control reasoning and is only evaluated in basic CoT mode. Reported accuracy is the mean over the three benchmarks, and both reasoning modes when applicable, and the last five exported checkpoints of each run, where performance has plateaued; error bars are two standard deviations bootstrapped over that temporal averaging (i.e., steps 1250, 1500, 1750, 2000, 2250). These intervals capture late-training checkpoint variation within a single run and do not account for seed or data-order variance. GSM8K is scored with flexible answer extraction, MATH-500 with symbolic verification, and MMLU-Pro with its standard extraction.

Prefill-heavy QADD results. We evaluate RULER's 13 tasks covering retrieval, multi-hop tracing, aggregation and question answering through the lm-evaluation-harness implementation. Each task uses 500 examples per context length at 4K, 8K, 16K and 32K, with zero-shot prompts, the model's chat template, greedy decoding and task-specific generation budgets of 30–128 tokens (default). Qwen 3 reasoning is disabled to retain the short-answer workload. We evaluate the same QADD checkpoints without benchmark-specific training, using the same disaggregated vLLM serving path as above. For each checkpoint, we average task scores within each context length, then average the four lengths. Reported means use the same five late checkpoints and bootstrap procedure as the decode-heavy results.

Qwen3.8-27B QADD results. For Section 2.6, we evaluate MMLU-Pro (12,032 items) with five category-specific CoT examples formatted as alternating user and assistant turns, and MMMU-Pro (1,730 items) with its zero-shot `vision` mode. Reasoning is enabled in both benchmarks, with temperature 1.0, top-$p = 0.95$, top-$k = 20$, a 32,768-token generation budget and a 65,536-token context limit. MMLU-Pro uses the lm-evaluation-harness answer pattern, counting extraction failures as incorrect; MMMU-Pro uses its official multiple-choice parser, with a fixed seed for its random fallback. Accuracy is computed over all benchmark items, without excluding truncated or unparsed responses.

All arms use vLLM on GB300. The GGUF decode weights are dequantized to BF16 for evaluation; fully-disaggregated arms pair these fixed weights with the trained NVFP4 prefill checkpoint and transfer both attention caches and recurrent state between engines. Figure 6 reports encoded GGUF backbone sizes rather than the allocations of this dequantized evaluation backend. We report one

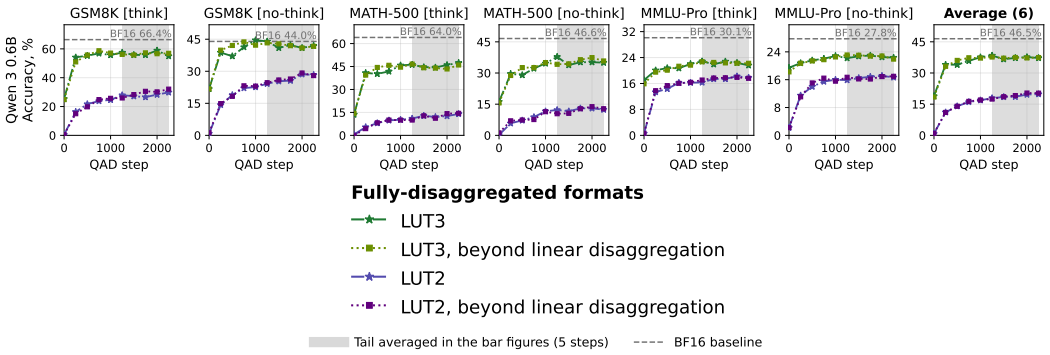


Figure 7: Fully-disaggregated quantization on Qwen3-0.6B with and without separation of the remaining unquantized parameters. This additional separation has no discernible effect on decode-heavy accuracy.

complete evaluation per format and benchmark, without repeat or checkpoint averaging. Fully-disaggregated arms use the final checkpoint at step 980.

Large and multi-modal PTQ models. For the models from Section 3.3, we report MMLU-Pro (12,032 items) and, for the multi-modal models, MMMU-Pro in the `vision` setting (1,730 items), scored with each benchmark's standard extraction. All arms are served with vLLM on GB300 nodes; disaggregated arms place the prefill and decode engines on disjoint nodes and transfer the KV cache over NVLink.

Four arms are reported per model. *BF16* is the released checkpoint and *W4A16* is our NVFP4A16 quantization of it. The two *W4A4* arms additionally quantize activations, and differ only in whether that is applied uniformly (*None*) or only to the prefill engine, with decode left weight-only (*Format*). Two models deviate from this scheme: Qwen3.8-2.4T's BF16 is actually FP8 in which it was natively trained and Kimi-K3 is released natively in MXFP4 with no BF16 checkpoint, so it has no BF16 column and its arms are MXFP4 rather than NVFP4. We use the recommended sampling parameters for each model. The model-specific reasoning parser is enabled where one exists. Kimi-K3 is evaluated with thinking enabled at its intermediate effort setting.

Flips analysis for large models. We generally follow the per-sample binary answer flips setup of Kübler et al. (2026). We conduct per-item paired tests using four evaluation passes for both MMMU-Pro and MMLU-Pro. With $s^A_{i,r} \in \{0,1\}$ the score of method A on item i in pass r, we test $\Delta = \frac{1}{n}\sum_i d_i$ where $d_i = \frac{1}{k}\sum_r s^A_{i,r} - \frac{1}{k}\sum_r s^B_{i,r}$. Under the null that the methods are exchangeable on every item, each d_i is equally likely to flip sign, giving $p = \Pr(|\sum_i \varepsilon_i d_i| \geq |\sum_i d_i|)$ with $\varepsilon_i \sim \mathrm{Unif}\{\pm 1\}$. Since $kd_i \in \mathbb{Z}$, the null distribution is a convolution of binomials and is computed exactly rather than sampled. We report per-comparison significance at $\alpha = 0.05$, without multiple-testing correction.

B Additional ablations

B.1 Disaggregation beyond quantized linear layers

Full disaggregation in Section 2.4 gives the quantized linear layers phase-specific weights while leaving the remaining unquantized parameters shared. We test whether separating these remaining parameters, including embeddings, the output head and layer normalizations, provides further benefit on Qwen3-0.6B, where they account for 26% of all model parameters. This "beyond linear disaggregation" has no discernible additional effect on decode-heavy accuracy (Figure 7). Both configurations already specialize the quantized linear weights, so this ablation does not distinguish their specialization benefit from that of higher prefill precision.

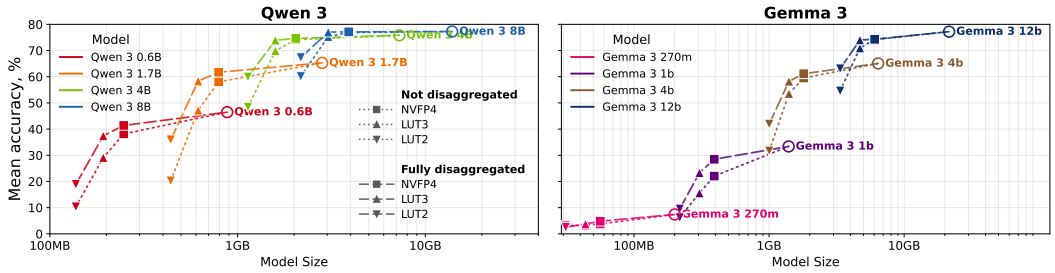


Figure 8: Decode-heavy accuracy of non-disaggregated and fully-disaggregated 2-4-bit quantized models with NVFP4 prefill versus encoded size of the quantized decode weights.

B.2 Weight-only quantization results

In Table 1, the reported weight-only quantization schemes are those used in "LUT3" and "LUT2" formats and described in Appendix A.3, except naturally without the NVFP4 autocast. Figures 14 and 15 break down their decode-heavy and prefill-heavy performance, respectively.

B.3 Pareto analysis

Table 1 also reports the cost of decode activation quantization. NVFP4 uses the native W4A4 pathway, while format-disaggregated NVFP4 uses NVFP4A16. For LUT2 and LUT3, the non-disaggregated decode timings add an FP4 activation-quantization call before each weight-only linear operation, discarding its output. These timings estimate the overhead of the extra operation, not a complete LUT-to-NVFP4 autocast implementation; they do not measure decode weight requantization.

For prefill, the table uses BF16 as the weight-only reference and reuses NVFP4 timings for LUT autocast, excluding weight-conversion overhead.

Figure 8 compares decode-heavy accuracy against the encoded size of the quantized decode weights for the Qwen 3 and Gemma 3 families. Full disaggregation improves low-bit accuracy without enlarging these weights, shifting the accuracy–decode-weight-size trade-off. The extra prefill checkpoint and its residency cost are accounted for separately in Table 1. Comparisons across model sizes also exclude different amounts of unquantized embedding and head weights.

B.4 Interoperability of prefillers

We test how much of the benefit of QADD transfers across decoders and how much depends on the decoder used during training. We compare three frozen Qwen3.8-27B decoders, IQ1_S, IQ1_M and IQ2_XXS, with each of the three step-980 prefiller and a common NVFP4 PTQ prefill checkpoint obtained by round-to-nearest (RTN) quantization. All four prefill checkpoints use NVFP4; we identify each QADD-trained prefiller by the decoder used during training. We exchange the prefillers without further training and keep the evaluation protocol of Appendix A.5 unchanged.

Table 7 separates the benefit of training prefill from the benefit of its pairing with a particular decoder. Compared with the common RTN prefill, training-matched prefillers improve MMLU-Pro accuracy by 13.1, 4.2 and 1.9 points for IQ1_S, IQ1_M and IQ2_XXS, respectively. On MMMU-Pro, the corresponding gains are 29.4 and 9.7 points for the two lowest-bit formats, while IQ2_XXS changes little (-0.3 points). Thus, at the lowest bitwidths, much of the benefit requires adapting prefill rather than merely replacing its weight-only computations with a separate NVFP4 checkpoint.

The learned checkpoints nevertheless remain partially interoperable. On MMMU-Pro, the IQ1_S decoder performs best with its own prefiller: accuracy falls from 59.65% to 52.60% with the IQ1_M prefiller and 47.05% with the IQ2_XXS prefiller. This ordering is consistent with stronger transfer between closer decode bitwidths in this comparison. It is not universal: the IQ1_M decoder reaches 62.49% with either its own or the IQ1_S prefiller, and on MMLU-Pro the IQ1_S decoder improves from 61.54% to 65.02% when paired with the IQ1_M prefiller.

Table 7: Qwen3.8-27B prefill–decode interoperability, accuracy (%). Rows fix the decode checkpoint; columns change only the prefill checkpoint. Bold marks the prefiller trained with that decoder, not necessarily the highest score. $\star$ denotes a difference from the bold cell in the same row under a two-sided exact McNemar test ($p < 0.05$, without multiple-comparison correction).

MMLU-Pro ($n = 12032$)				
Frozen decode	Prefill checkpoint (all NVFP4)			
	PTQ (RTN)	QADD IQ1_S	QADD IQ1_M	QADD IQ2_XXS
IQ1_S	48.39*	**61.54**	65.02*	62.59*
IQ1_M	68.37*	69.62*	**72.59**	72.85
IQ2_XXS	76.06*	74.27*	77.44	**77.93**
MMMU-Pro ($n = 1730$)				
Frozen decode	Prefill checkpoint (all NVFP4)			
	PTQ (RTN)	QADD IQ1_S	QADD IQ1_M	QADD IQ2_XXS
IQ1_S	30.29*	**59.65**	52.60*	47.05*
IQ1_M	52.83*	62.49	**62.49**	61.50
IQ2_XXS	66.18	67.11	66.30	**65.90**

Table 8: Generation length and truncation for Qwen3.8-27B. WO uses the released weight-only checkpoint in both phases; Pref. pairs the same decoder with its trained NVFP4 prefiller at step 980. Token counts include reasoning and final-answer generation over all benchmark items, including truncated responses. Length statistics are computed from per-item counts and rounded to the nearest token; truncation rates count length-limit terminations. Each arm uses one complete evaluation, with the same quantized checkpoints as Table 5.

MMLU-Pro								
Decode format	Mean tokens		Median tokens		95th percentile		Truncated (%)	
	WO	Pref.	WO	Pref.	WO	Pref.	WO	Pref.
IQ1_S	3964	4260	928	778	18952	26677	2.46	3.34
IQ1_M	3312	3612	883	699	16910	23550	2.42	3.39
IQ2_XXS	2394	3897	746	782	11191	23999	1.25	3.39
IQ2_S	1706	2766	774	671	6352	14429	0.27	1.53
Q2_K_XL	2647	2787	799	671	13092	14721	0.95	1.36
IQ3_XXS	2514	2872	804	653	12380	15722	1.15	1.54
IQ3_S	2799	3000	757	676	14839	16496	1.14	1.49
Q3_K_XL	2240	2701	679	658	11211	14071	0.66	0.97
MMMU-Pro								
Decode format	Mean tokens		Median tokens		95th percentile		Truncated (%)	
	WO	Pref.	WO	Pref.	WO	Pref.	WO	Pref.
IQ1_S	14579	6582	10966	2393	32768	29953	24.57	4.16
IQ1_M	8013	6875	3702	2459	32768	32768	6.24	6.47
IQ2_XXS	5294	7377	3236	3337	16669	32768	0.52	5.55
IQ2_S	4695	7118	2456	2930	15780	30445	0.40	4.34
Q2_K_XL	5094	6079	2823	2688	16736	25067	0.58	2.77
IQ3_XXS	5400	6850	2844	2852	19363	29873	1.27	4.10
IQ3_S	5860	6911	3212	3172	20578	28562	1.45	3.82
Q3_K_XL	5389	7156	2676	3305	19728	29416	1.39	3.53

B.5 Generation length with prefillers

An unchanged decode checkpoint does not imply unchanged generation cost: the prefiller changes the representations that condition generation and can therefore change response length. Table 8 reports generation lengths and truncation rates for the same Qwen3.8-27B evaluations as Table 5, using the final prefill checkpoint at step 980 and a common 32,768-token generation budget.

On MMMU-Pro, the largest low-bit accuracy gains coincide with shorter mean responses. IQ1_S's mean response length falls from 14,579 to 6,582 tokens, a 54.8% reduction, while accuracy rises from 24.4% to 59.7%. Its median falls from 10,966 to 2,393 tokens and its truncation rate from

Table 9: Attention backend selection for Gemma3-4B: $H = 8$, $H_{kv} = 4$, $d = 256$, window 1024. Latency is in milliseconds per call; bold marks the selected backends.

Backend	S=8192	S=32768	Used for
SDPA, `is_causal`	**3.14**	**50.6**	global layers
FA2 varlen, full causal	3.75	53.0	-
FA4 (`flash_attn.cute`)	3.06	49.8	-
FlexAttention, causal mask	-	123.2	-
FlexAttention, sliding block mask	2.03	8.7	-
FA4, native `window_size`	3.02	49.0	-
FA2 varlen, native window	**1.47**	**6.0**	local layers

24.57% to 4.16%. IQ1_M likewise uses 14.2% fewer tokens on average while gaining 17.6 accuracy points. These improvements are therefore not bought with more generated tokens.

The effect depends on the workload and format. On MMLU-Pro, prefillers shorten median generations for seven of eight formats by 3–21%, but increase mean length by 5–63%: longer upper tails outweigh the shorter typical responses. On MMMU-Pro, the remaining six formats also increase mean length. Prefill specialization can therefore reduce the total number of decode tokens as well as improve accuracy, but the unchanged decode weights and per-token pathway do not by themselves guarantee lower generation cost.

C Speed measurements

C.1 Prefill measurements

We benchmark on a single NVIDIA GB10 (DGX Spark, `sm_121`) with PyTorch 2.13 / CUDA 13.2. These transformer-stack measurements are separate from the `llama.cpp` TTFT measurements in Figure 6. For the latter, we integrate the same offloaded NVFP4 prefill pathway into `llama.cpp`'s prompt processing, without additional kernel optimizations. The baseline uses Unsloth's IQ1_S checkpoint through native `llama.cpp` processing. TTFT measurements use three repetitions after one warmup, with prompt caching disabled.

Compute and timing scope. BF16 uses cuBLAS through `torch.nn.Linear`; NVFP4 uses vLLM activation quantization with static global scales and shape-tuned CUTLASS or FlashInfer GEMMs. We fuse compatible projections and activation processing, and use CUDA graphs. Attention backends are selected per layer type (Table 9). The timed region spans the first transformer layer's input through the last layer's output, including attention but excluding embedding lookup, RoPE-table construction, final normalization and the output head.

Offloading protocol. For the core Qwen 3 and Gemma 3 models, ODP streams weights from SSD through pinned host buffers into two device slots. Slots borrow decode-weight memory. The benchmark represents the evicted weights by a serialized byte payload equal to the slot allocation and restores it into those buffers. Timing includes the first block's cold load and ends only after restoration completes. Total SSD traffic is the prefill checkpoint plus the carve-out: $3.64 + 0.20$ GB for Qwen3-8B and $5.64 + 0.23$ GB for Gemma3-12B.

Before each repetition, we request page-cache eviction for both prefill files and the restoration payload. Loading-only references are measured separately for each model and format, including carve-out transfer for ODP.

Speedup decomposition. Table 10 reports full-stack results, Table 11 explains where time is spent within a layer. The 3.0–$3.4\times$ projection speedups are diluted by attention, normalization, activation processing and quantization. Attention accounts for 37% of NVFP4 device time on Qwen3-8B, which uses global attention throughout, versus 13% on Gemma3-12B, which alternates five sliding-window layers with one global layer. Fused timings do not isolate activation-quantization overhead. Layer wall-clock and profiler timings come from separate runs, so their residual includes measurement variation and is not a direct launch-cost estimate.

Table 10: Prefill transformer-stack speedup over BF16 for the core Qwen 3 and Gemma 3 models on DGX Spark. NVFP4 keeps weights resident; +ODP streams them under the cold-load and carve-out protocol above. BF16 columns give latency in milliseconds. All three arms are measured in the same benchmark session.

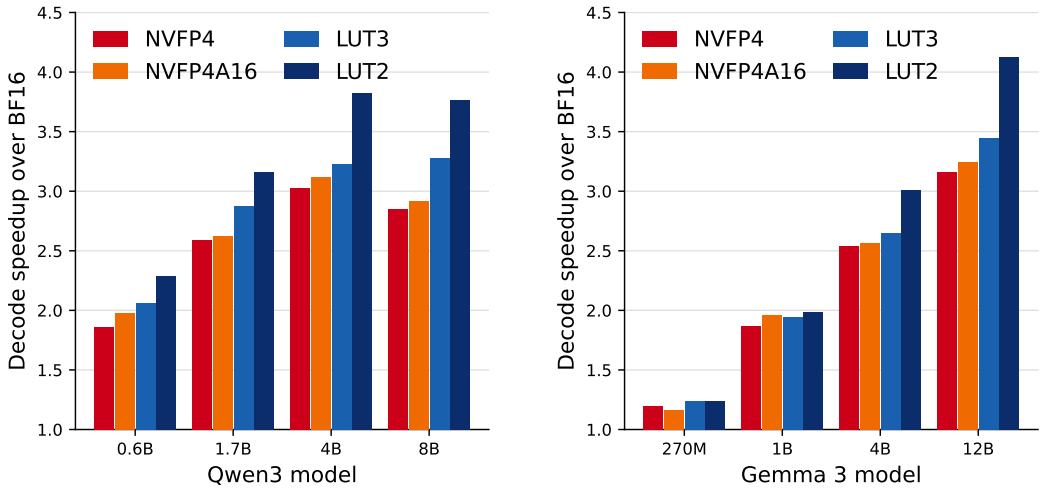
Model	S=16384			S=32768		
	BF16 (ms)	NVFP4	+ODP	BF16 (ms)	NVFP4	+ODP
Qwen3-0.6B	634	1.13×	1.12×	1984	1.08×	1.05×
Qwen3-1.7B	1044	1.24×	1.19×	2887	1.15×	1.15×
Qwen3-4B	2560	1.37×	1.34×	7335	1.14×	1.15×
Qwen3-8B	3801	1.49×	1.47×	9563	1.24×	1.24×
Gemma-3-270M	124	1.22×	1.16×	281	1.14×	1.11×
Gemma-3-1B	473	1.45×	1.34×	999	1.43×	1.41×
Gemma-3-4B	1548	1.59×	1.49×	3231	1.53×	1.49×
Gemma-3-12B	4650	1.67×	1.58×	9607	1.59×	1.55×

Table 11: Per-component prefill latency at S=16384. Components sum to profiled device time for one layer; attention is averaged over each model's layer-type mix. Fused operations are grouped: NVFP4 `norms` includes activation quantization, and `gate_up`/`down` include all chunked GEMM launches. The final row gives full-stack latency divided by the layer count.

Component	BF16 (ms)	NVFP4 (ms)	Speedup	% NVFP4
Gemma-3-12B: 48 layers, 40 sliding attention / 8 global attention				
qkv	10.23	5.57	1.84×	10
o	6.51	1.46	4.47×	3
gate_up	38.47	11.66	3.30×	21
down	20.26	6.32	3.21×	11
MLP non-linearity	6.00	5.19	1.16×	9
Activation quant. (standalone)	-	2.24	-	4
Norms + RoPE + residual	6.48	16.36	0.40×	29
Attention	7.40	7.38	1.00×	13
Device busy (sum of kernels)	95.35	56.17	1.70×	100
Launch/idle gap	-0.24	-0.24	-	-
Layer (wall clock)	95.12	55.94	1.70×	-
Full stack, per layer	**96.87**	**58.09**	**1.67**×	-
Qwen-3-8B: 36 layers, all global attention				
qkv	8.16	3.68	2.21×	6
o	6.27	1.59	3.95×	2
gate_up	33.68	9.72	3.46×	15
down	18.54	4.63	4.01×	7
MLP non-linearity	4.79	4.19	1.14×	6
Activation quant. (standalone)	-	2.34	-	4
Norms + RoPE + residual	6.44	15.81	0.41×	24
Attention	25.59	24.59	1.04×	37
Device busy (sum of kernels)	103.46	66.54	1.55×	100
Launch/idle gap	-0.08	-0.01	-	-
Layer (wall clock)	103.38	66.54	1.55×	-
Full stack, per layer	**105.57**	**71.09**	**1.49**×	-

C.2 DECODE KERNELS

We back up our claims of decode speed depending on the degree of compression by benchmarking memory-bound GEMV kernels when used for LLM decoding in vLLM. Figure 9 shows real



(a) Qwen 3

(b) Gemma 3

Figure 9: End-to-end single-user decode speedup over dense BF16, measured in vLLM on DGX Spark. The speedup is whole-model per-output-token latency, so it includes attention, normalisation and the BF16 language-modelling head, none of which any of these formats accelerates.

speedups measured on DGX Spark for NVFP4 and NVFP4A16, natively shipped with vLLM, as well as LUT3 and LUT2 kernels that we implemented and integrated.

Protocol. All numbers are end-to-end vLLM generations at batch one. The same 168-token prompt is generated to n_1=8 and n_2=128 output tokens, and the per-output-token latency is estimated as $(t_2 - t_1)/(n_2 - n_1)$, which removes the common prefill contribution without relying on engine-internal metrics. Each point is the median of three repetitions after a warm-up generation at that shape. Prefix caching is disabled. CUDA graphs are enabled.

Kernel specification. Decode at batch one is bandwidth-bound: the arithmetic is a matrix-vector product, so time is set almost entirely by the bytes of weight pulled per token. The four formats differ mainly in that quantity — BF16 at 16, NVFP4 at 4.5, LUT3 at 3.5 and LUT2 at 2.5 bits per weight. NVFP4 is vLLM's native W4A4 CUTLASS path, which quantizes the activation before every projection and issues an FP4×FP4 GEMM. NVFP4A16 keeps activations in BF16 and uses a Marlin-like (Frantar et al., 2024) mixed-precision kernel. Our LUT3 and LUT2 kernels store weights bit-planed as `int32` over $(N, b, K/32)$ with signed `e4m3` block scales every 16 elements and one FP32 global scale per tensor, and reconstruct values by indexing a 2^b-entry lookup table in shared memory. This lookup is implemented in software on DGX Spark and does not require native LUT arithmetic support.

Notably, format-disaggregated NVFP4 (identical to NVFP4A16 on decode) outperforms NVFP4 by skipping activation quantization and using a kernel tailored to the weight-only matrix-vector product. The speedup over BF16 increases from 2.86× to 2.93× on Qwen3-8B and from 3.18× to 3.27× on Gemma3-12B.

Speedup dilution. LUT2 moves 6.4× fewer weight bits than BF16 yet delivers 3.82× and 4.15× speedups, respectively. Two effects account for the gap. Firstly, the output head stays in BF16 in every arm, and on the smaller models its weights account for a substantial fraction of the bytes moved per token. Secondly, embedding lookup, attention, the normalizations and the residual adds are identical work in every arm, so they dilute the speedup. The same principle applies to unaccelerated operations in prefill (Table 11). On its own, the LUT2 kernel reaches 96% of the bandwidth its weight traffic permits.

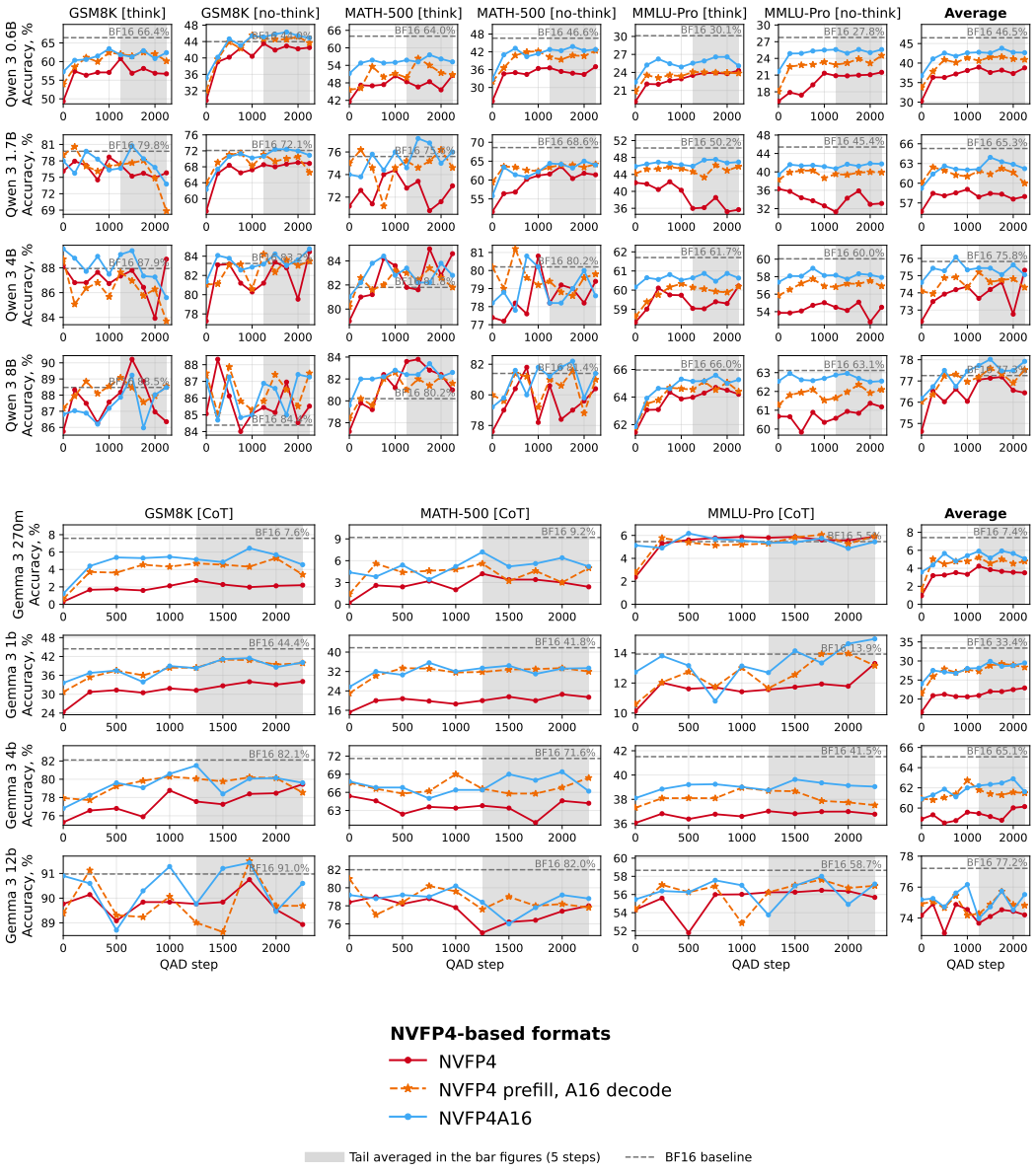


Figure 10: Breakdown of Figure 3 by benchmark and QADD step.

D FULL EVALUATION RESULTS

Figures 10, 12 and 14 break down decode-heavy QADD accuracy recovery by benchmark and training step. Figures 13, 11 and 15 provide the corresponding prefill-heavy breakdowns by context length and training step.

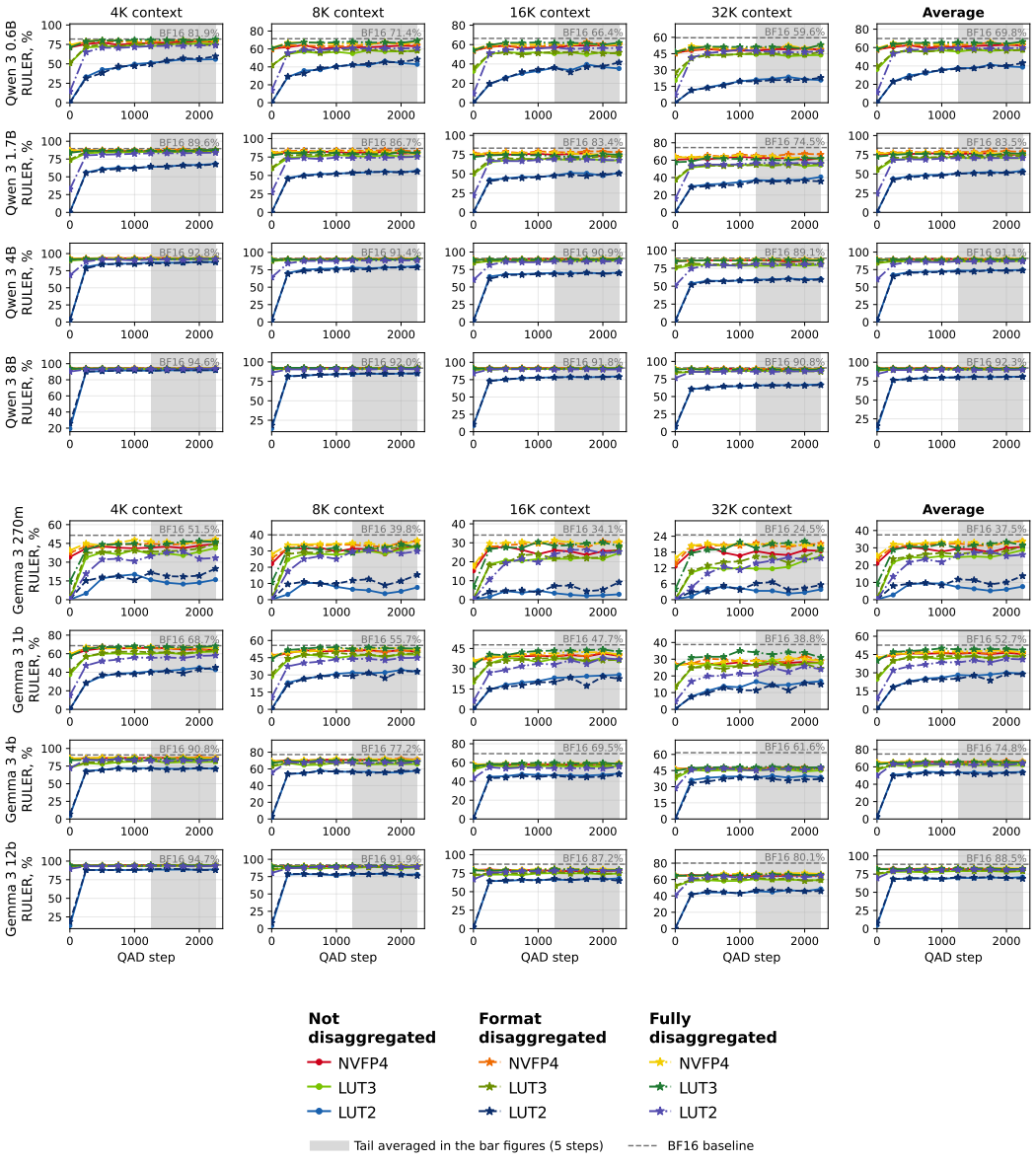


Figure 11: Breakdown of Figure 4 by context length and QADD step for prefill-heavy tasks.

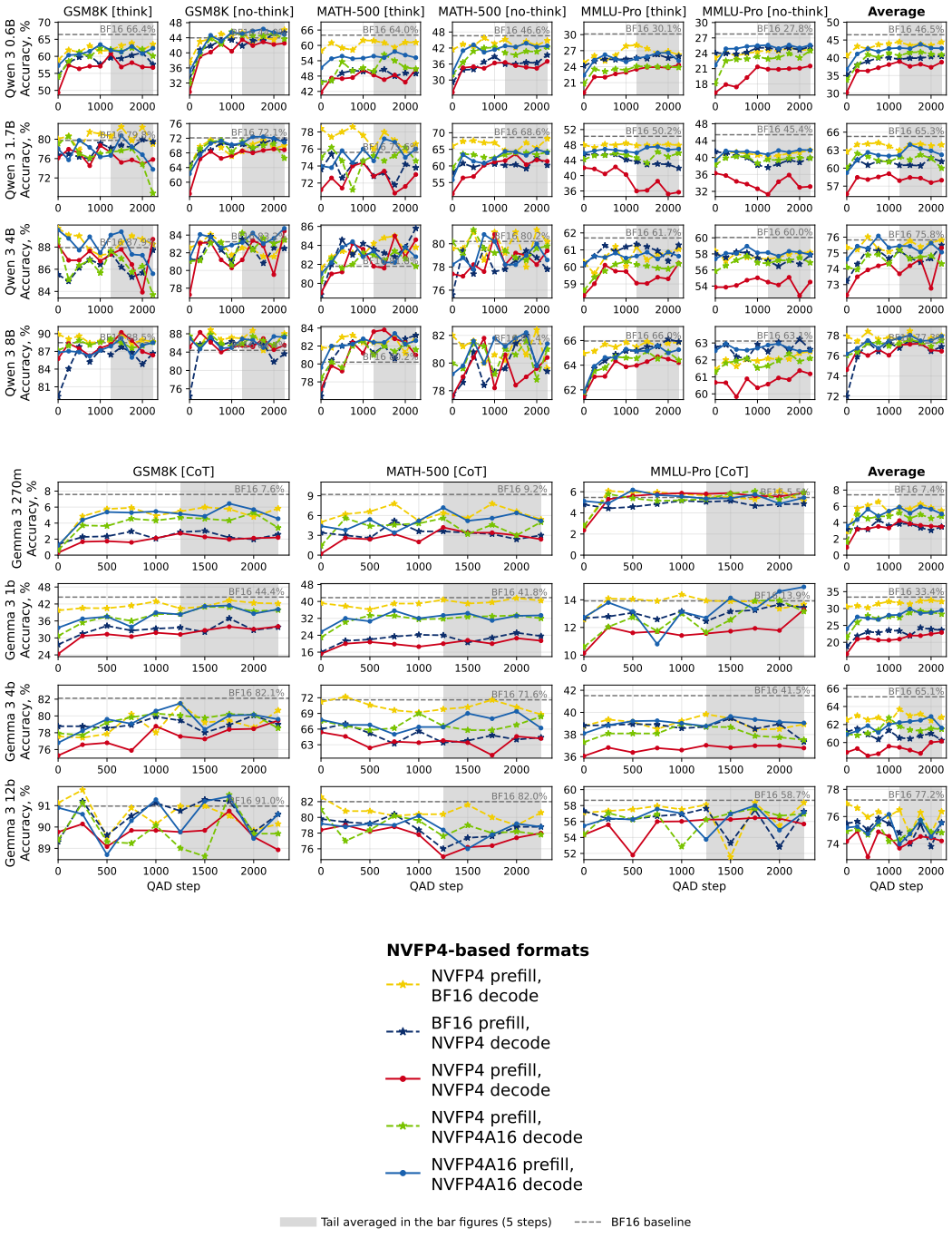


Figure 12: Breakdown of Figure 3 by benchmark and QADD step for decode-heavy tasks.

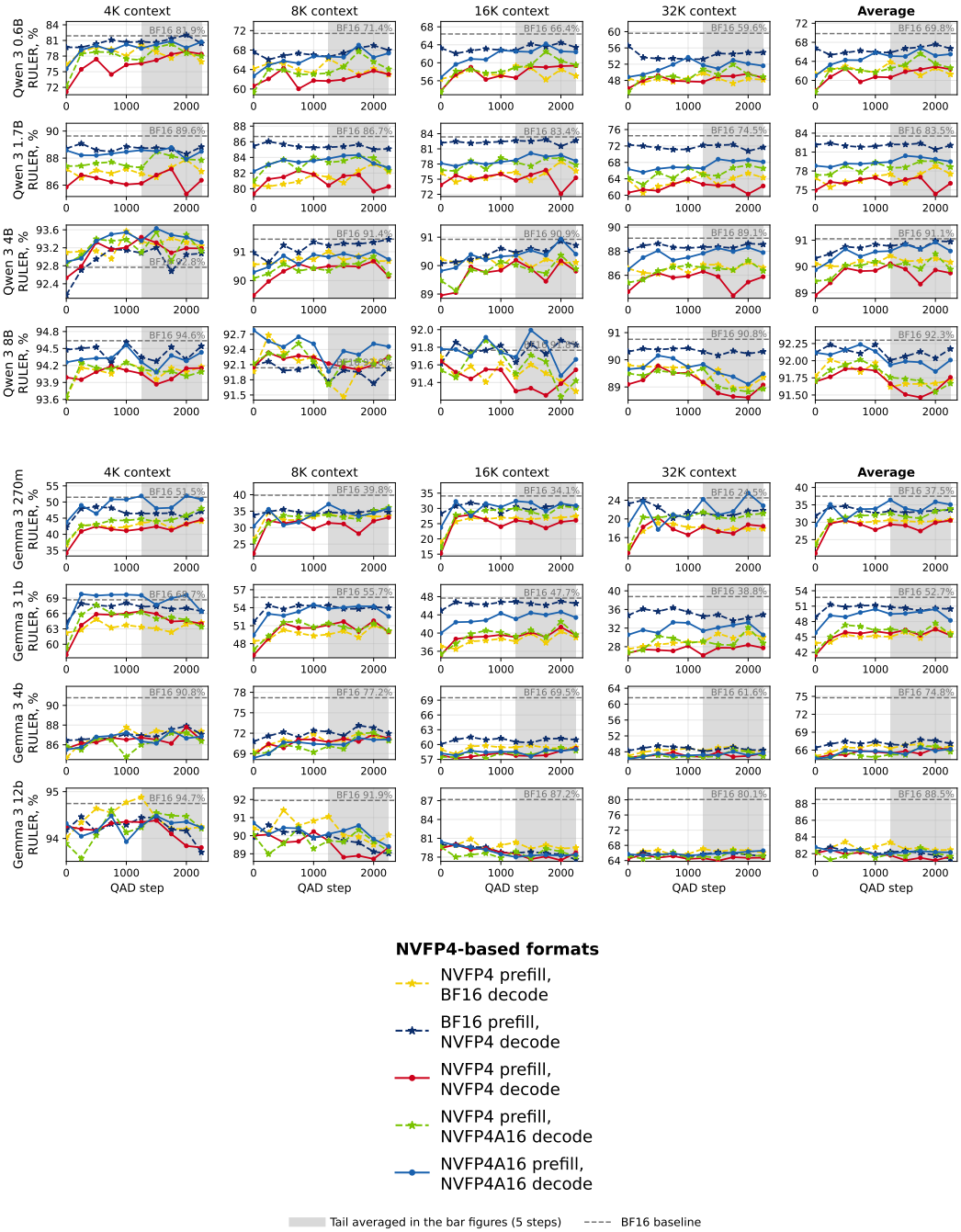


Figure 13: Breakdown of Figure 3 by context length and QADD step for prefill-heavy tasks.

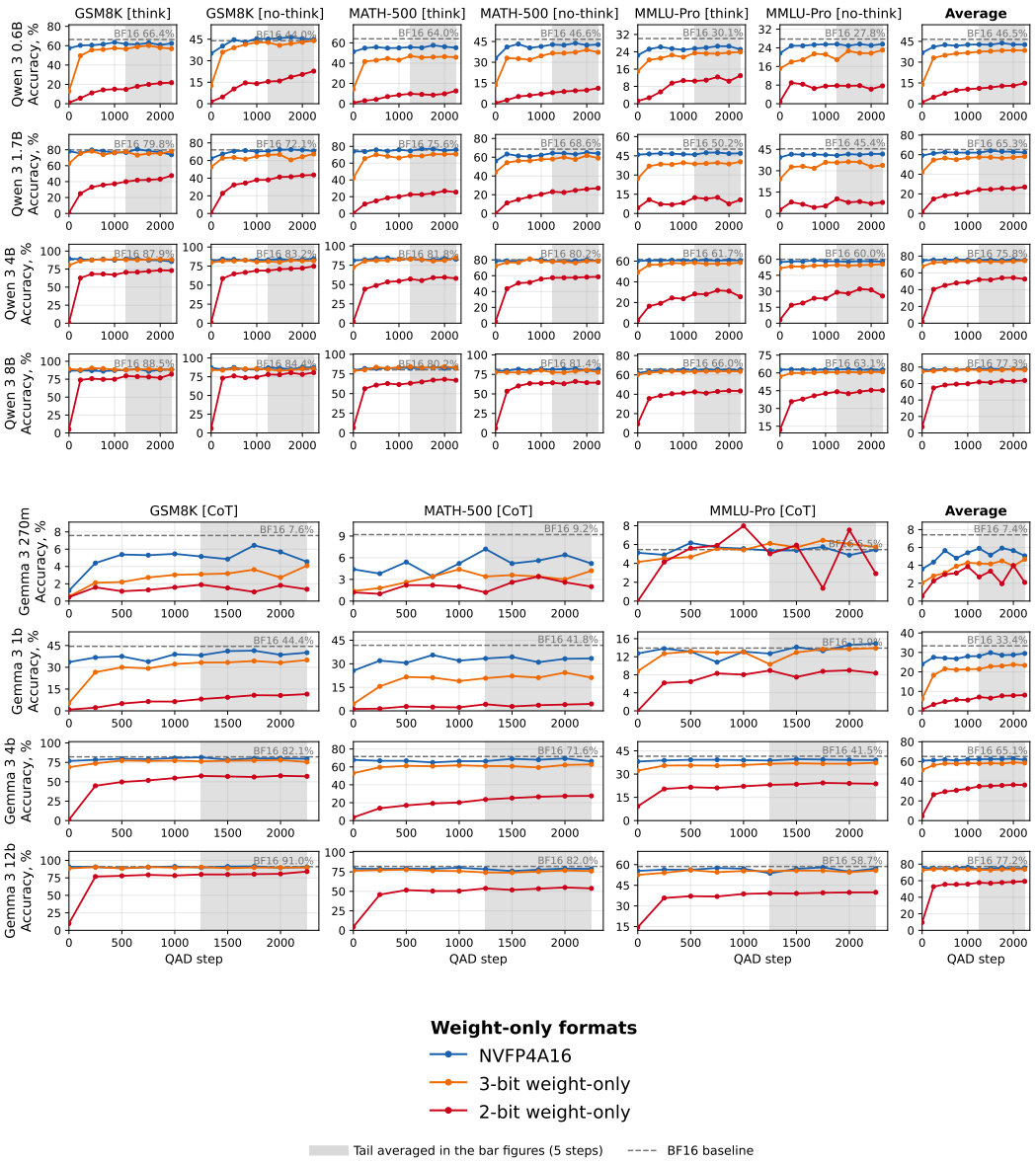


Figure 14: Performance breakdown for weight-only quantization formats by benchmark and QADD step.

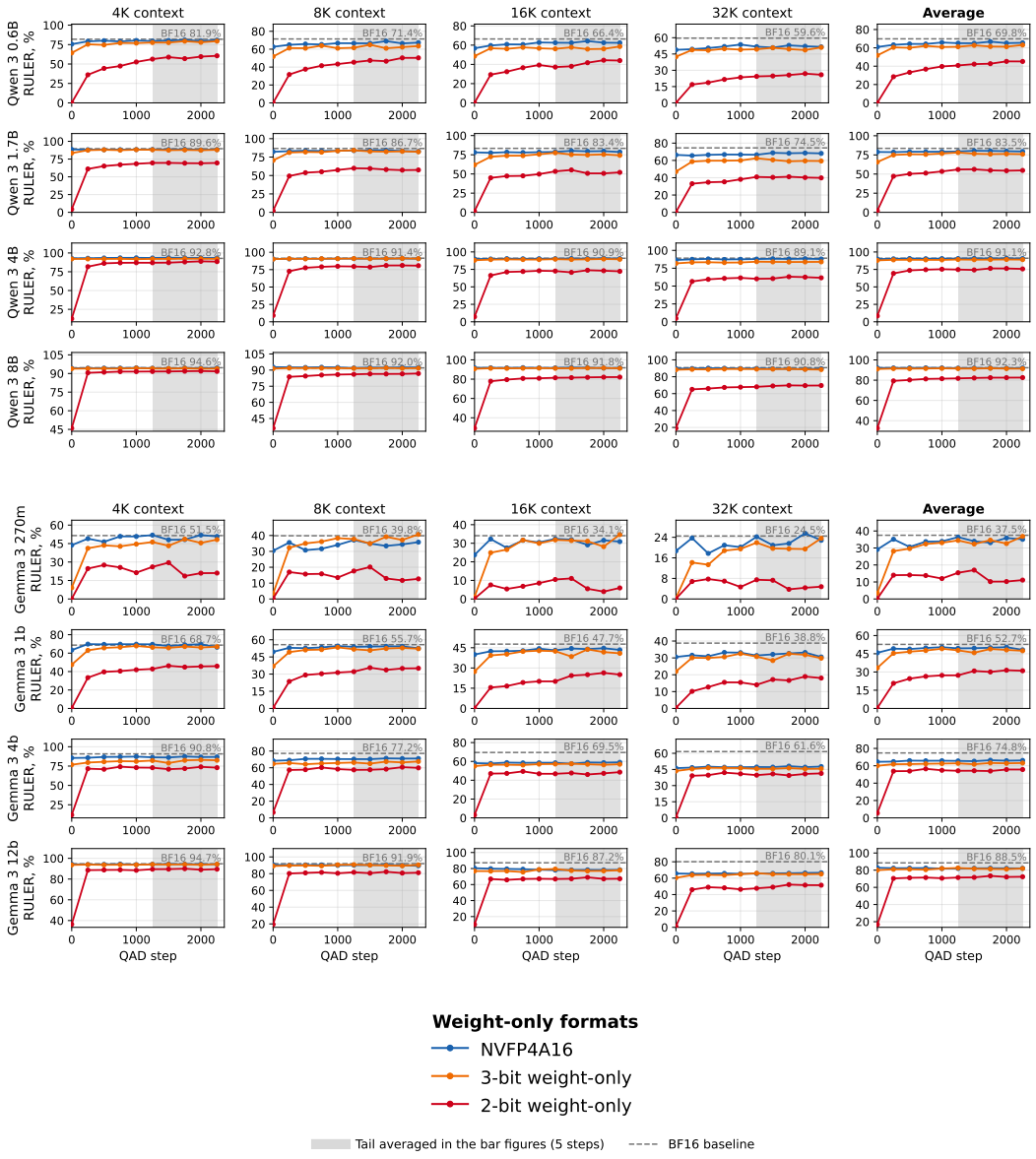


Figure 15: Performance breakdown for weight-only quantization formats on prefill-heavy tasks by context length and QADD step. The average excludes 64K evaluations.