LOADMATCH: BALANCING MEMORY AND COMPUTE LOADS IN QUANTIZED MODELS USING SPECULATIVE DECODING

Anonymous authors
Paper under double-blind review

ABSTRACT

Quantization has become a standard practice for efficient large language model (LLM) inference. However, the exact mechanism behind this acceleration is often misunderstood. A common misunderstanding is that quantized weights directly speed up arithmetic operations: in reality, low-bit GEMM still incurs higher-precision dequantization, rescaling, and accumulation, so its compute-side speedup is much smaller than its memory-side gain. Consequently, the computation time itself remains largely unchanged. However, the perceived speedup stems from the efficiency gained by reducing the transfer time of quantized weights from the GPU memory to its cache. Because of this, quantized LLMs achieve significant speedups only in memory-bound scenarios, whereas their improvements in compute-bound regimes are marginal. To truly maximize the decoding throughput of quantized LLMs, a system must achieve a load-matched state where the memory transfer duration and computation time are equal so that neither the memory bus nor the compute units remain idle. To reach this state, we propose LoadMatch, which balances memory and computation costs using speculative decoding. We leverage speculative decoding, in which a draft model predicts multiple future tokens, called candidate tokens, and the target model verifies them in parallel. By precisely controlling the candidate-token count processed at once, our method deliberately matches the memory and computation workload. Empirically, LoadMatch compounds the speedups from quantization and speculative decoding, delivering a measured end-to-end speedup of $8.8\times$ over FP16 autoregressive greedy decoding on standard benchmarks. We will release the project upon acceptance.

1 INTRODUCTION

Does quantizing a 16-bit LLM to 4-bit guarantee a $4\times$ faster computation? While it is an undeniable fact that quantization (Frantar et al., 2023) drastically reduces the memory footprint, which makes it highly useful for deploying massive models to edge devices, its impact on decoding throughput requires a deeper architectural reflection. A naive assumption suggests that matrix multiplications are performed directly using 4-bit quantized weights, leading to faster arithmetic. However, the reality of hardware execution tells a different story. During inference, modern GPUs execute general matrix multiplication (GEMM) by using low-precision arithmetic together with higher-precision dequantization and accumulation, as practical schemes (Lin et al., 2024; Zhao et al., 2024) employ group-wise scaling factors. Therefore, the true driver of the observed LLM acceleration is the time saved when transferring weight parameters and activations from the GPU memory to the cache.

This dequantization overhead primarily stems from group-wise quantization, which is commonly employed to preserve model accuracy at low precision (Frantar et al., 2023). Specifically, for each group g, let the quantized activation and weight vectors be $\hat{\mathbf{x}}_g, \hat{\mathbf{w}}_g \in \mathbb{Z}^d$. Their dequantized vectors are $\mathbf{x}_g = s_{x,g}\left(\hat{\mathbf{x}}_g - z_{x,g}\right), \mathbf{w}_g = s_{w,g}\left(\hat{\mathbf{w}}_g - z_{w,g}\right)$, where $s_{x,g}$ and $s_{w,g}$ are the group-wise scales, and $z_{x,g}$ and $z_{w,g}$ denote the corresponding zero points. The resulting inner product is $y = \sum_g \mathbf{x}_g^\top \mathbf{w}_g = \sum_g s_{x,g} s_{w,g} \left(\hat{\mathbf{x}}_g - z_{x,g}\right)^\top \left(\hat{\mathbf{w}}_g - z_{w,g}\right)$. Because the quantization parameters vary across groups, quantized GEMM requires additional higher-precision dequantization and accumulation, whose lower arithmetic throughput introduces compute overhead (Zhao et al., 2024;

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.

Lin et al., 2025). This overhead appears across the three widely adopted low-bit configurations: weight-only quantization (`W4A16`), mixed weight-activation quantization (`W4A8`), and fully low-bit quantization (`W4A4`). Although their arithmetic paths differ, all three configurations incur additional computation during GEMM, leading to the speedup collapse observed in this paper.

For example, we compare three state-of-the-art quantized GEMM methods: (1) Marlin (Frantar et al., 2025) with weight-only quantization (`W4A16`), (2) QoQ (Lin et al., 2025) with mixed weight-activation quantization (`W4A8`), and (3) APEX4 (Guo et al., 2026) with fully low-bit quantization (`W4A4`). As shown in Figure 1, the results exhibit that the speedup gains of all three methods substantially narrow in compute-bound regimes. Section F details why the speedup collapses in a compute-bound setting.

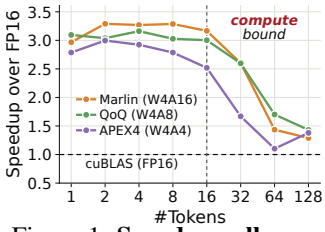


Figure 1: **Speedup collapse.**

This architectural reality reveals a critical limitation: quantization provides substantial speedups only when the system is operating in a memory-bound state. Once the workload shifts to compute-bound, the time saved in memory transfer is eclipsed due to the remaining arithmetic overhead, rendering the actual improvement in response time marginal. From this perspective, the most effective way to minimize the response time of quantized LLMs is to ensure that the memory transfer time and the computation time are balanced. When these two distinct pipelines achieve a strict load match (see Figure 2), the workload sits exactly at the ridge point of the GPU roofline model (Williams et al., 2009), where both the memory and compute units run at zero idle. Consequently, this balance enables maximum throughput of quantized LLMs. A natural way to fill the idle compute units is to batch multiple requests, which raises the arithmetic load per weight transfer in proportion to the batch size. However, in practice, using the optimal batch size is not always possible. Interactive services must return each response as soon as it is ready and cannot hold requests until stacking the optimal batch sizes (Agrawal et al., 2024). On edge devices, a single user typically issues one request at a time, so inference runs at a batch size of one or a few (Li et al., 2025a). In addition, even when employing multiple samples in a batch, it is difficult to identify the ridge point of the roofline because the token lengths generated by each sample are not the same (Yu et al., 2022). This leads to an overhead in which the entire system is required to pause for the sample with the longest generated sequence to finish. Therefore, the compute units are left underutilized in many real applications, so we must balance the memory-footprint and compute cost for a single request.

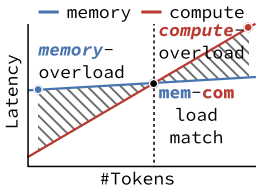


Figure 2: **LoadMatch.**

To actively reach this ridge point, we leverage speculative decoding (SD) (Li et al., 2025b; Chen et al., 2026; Huang et al., 2026) in the quantized LLMs as a mechanism for balancing memory and compute workloads. SD accelerates inference by letting a small draft model propose several future tokens as candidates, which the original LLM, called the target model, verifies in a single forward pass. The crucial advantage of this process is that the number of generated candidate tokens serves as a controllable knob for GPU workload modulation. Specifically, in a memory-bound scenario, where compute units remain idle while awaiting weights, we intentionally increase the candidate-token count. This effectively scales up the computational cost until it matches the memory transfer time. Conversely, when the system enters a compute-bound state due to excessive token generation, we decrease the candidate-token count to shed the arithmetic burden until the computation load aligns back with the memory I/O. By bidirectionally adjusting the degree of speculative parallelism during calibration, we can precisely match the computational load to the accelerated memory transfer load of the quantized LLMs. Importantly, this calibration is performed offline rather than dynamically at runtime; once adjusted, the candidate-token count remains fixed and reused across tasks.

This paper presents LoadMatch (LM), a method designed to automatically find the load-match point of the existing SD methods on a quantized target. Rather than relying on blind heuristic tuning, our approach introduces a systematic two-stage profiling mechanism that first identifies the load-match point and then refines it to maximize the throughput. Unlike Chen et al. (2024), which optimizes the structure, size, and depth of speculative token trees for the target hardware, LoadMatch retains the underlying SD method and adjusts only the candidate-token count verified by the quantized target. Additionally, to ensure that the drafting phase itself does not disrupt this delicate balance, we introduce drafter slimming, a compression method that jointly prunes and quantizes both the draft model and its heavy vocabulary projection layer, which is known as `lm_head`. By matching the two workloads, LoadMatch brings the speedup of quantized speculative decoding close to the product

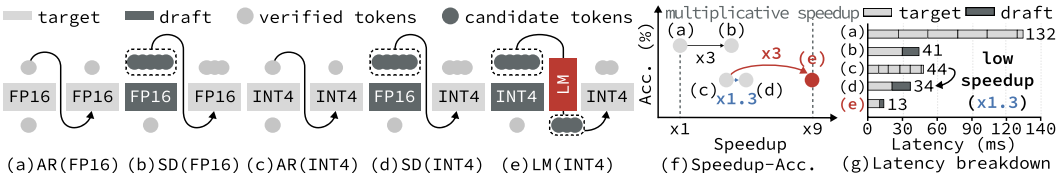


Figure 3: **Motivation illustration.** We compare auto-regressive AR and speculative decoding SD on quantized LLMs INT4. In (e), LM denotes for the LoadMatch algorithm. In (f,g), we employ EAGLE-3 for speculative decoding and GPTQ for quantization on Llama3-8B.

of the two individual speedups. For example, on the GSM8K dataset, LoadMatch achieves a 9.35× speedup, 98% of the product of the speedups that each approach attains separately (Table 1).

To report the speedup over the real applications, we implement an inference system that runs the entire speculative decoding pipeline inside CUDA Graphs, which requires fully vectorized, graph-compatible drafting and verification logic. Against vLLM (Kwon et al., 2023), a highly optimized serving engine, this system still delivers reasonable speedups on local devices. On top of that, we show that LoadMatch adapts existing SD methods to quantized LLMs without any retraining. We evaluate it on eleven benchmarks spanning math reasoning, code generation, conversation, and general NLP tasks. LoadMatch improves over the existing methods in both speed and energy efficiency.

2 METHOD

Motivation: Before presenting the technical details, we first ask whether a quantized LLM can be further accelerated simply by applying existing speculative decoding. This subsection explains why such a direct application, without balancing the memory and compute loads, yields far less speedup than expected. The key reason is the additional computation of parallel verification. Although this extra computation has little effect on verification latency for an FP16 target model, it can moderately increase verification latency for a quantized target model by pushing execution out of the memory-bound regime in which quantization is effective. As a result, the increased verification latency offsets the reduction in the number of target-model calls. Specifically, speculative decoding cuts the number of target model calls of autoregressive decoding (AR) by verifying multiple candidate tokens in parallel. Quantization (INT4) lowers the latency of each call by lowering weight precision (Figure 3(a)-(c)). Their combination SD(INT4) in Figure 3(d) is thus expected to improve both dimensions. However, the same speculative decoding method that accelerates AR(FP16) by 3× provides only a 1.3× for AR(INT4) (Figure 3(f)). The latency breakdown in Figure 3(g) identifies the target model as the source of this loss. Quantization reduces the single-token latency of the target model by 3×, but this reduction is nearly lost when the target model verifies multiple candidate tokens at once, leaving verification as the dominant cost of SD(INT4).

(i) Why does the same increase in compute raise the latency only in the quantized model? Because quantization pulls the load-match point of the target model forward to a smaller token count. The load-match point is the token count at which the memory load and the compute load of the GPU are matched. Figure 4 visualizes this point via the standard roofline analysis (Williams et al., 2009) of the matrix multiplications that account for 65-85% of LLM computation (Dettmers et al., 2022). Below it, added compute leaves the latency unchanged, and above it, the latency grows with the added compute. Quantization reduces the memory time (*e.g.*, 3×) much more than the effective compute time per token, so the two loads are matched at a smaller token count than in FP16. For the FP16 target model, the load-match point lies far above the candidate-token count used by existing speculative decoding methods, so the compute they add through parallel verification does not raise the latency. Without the proposed load matching, the same added compute pushes the quantized target model past its shifted load-match point, resulting in a verification cost no lower than that of FP16.

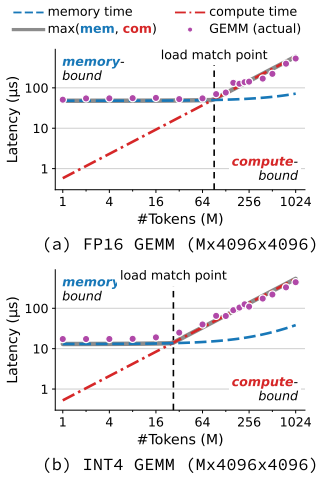


Figure 4: **Roofline analysis.**

Table 1: **Overhead breakdown.**

Method	$\tau(\gamma)\uparrow$	$\frac{\mathrm{D}(\gamma)}{\mathrm{T}(1)}\downarrow$	$\frac{\mathrm{T}(\gamma)}{\mathrm{T}(1)}\downarrow$	$\kappa(\gamma)\downarrow$	$\eta(\gamma)\uparrow$
SD(FP16)	5.86	0.59	1.14	1.73	N/A
SD(INT4)	5.88	1.61	2.40	4.01	0.43
LM(INT4)	5.20	0.47	1.09	1.56	0.98

(ii) Overhead measurement and compounding reference. We formalize how much speculative overhead must be removed for the two speedups to compound, taking their product as the reference. Let γ denote the candidate-token count per step, $\tau(\gamma)$ the acceptance length, $\mathrm{D}(\gamma)$ and $\mathrm{T}(\gamma)$ the drafting and verification latencies, where the subscript denotes the precision. Using FP16 autoregressive decoding as $\mathrm{T}_{\mathrm{FP}}(1)$ as the baseline, the speedups of speculative decoding, quantization, and their combination are $\mathcal{S}_{\mathrm{SD}}(\gamma) = \frac{\tau_{\mathrm{FP}}(\gamma)}{\kappa_{\mathrm{FP}}(\gamma)}$, $\mathcal{S}_{\mathrm{INT}} = \frac{\mathrm{T}_{\mathrm{FP}}(1)}{\mathrm{T}_{\mathrm{INT}}(1)} = \beta$, and $\mathcal{S}_{\mathrm{SD(INT)}}(\gamma) = \frac{\tau_{\mathrm{INT}}(\gamma)\beta}{\kappa_{\mathrm{INT}}(\gamma)}$, where β is the single-token verification speedup from quantization and κ is the speculative overhead, defined as the sum of the drafting and verification latencies normalized by the single-token verification latency of the target model, $\kappa_{\mathrm{FP}}(\gamma) = \frac{\mathrm{D}_{\mathrm{FP}}(\gamma)+\mathrm{T}_{\mathrm{FP}}(\gamma)}{\mathrm{T}_{\mathrm{FP}}(1)}$ and $\kappa_{\mathrm{INT}}(\gamma) = \frac{\mathrm{D}_{\mathrm{FP}}(\gamma)+\mathrm{T}_{\mathrm{INT}}(\gamma)}{\mathrm{T}_{\mathrm{INT}}(1)}$. The combined speedup $\mathcal{S}_{\mathrm{SD(INT)}}$ equals the product $\mathcal{S}_{\mathrm{SD}} \cdot \mathcal{S}_{\mathrm{INT}}$ (*e.g.*, 3×3 in Figure 3(f)), when $\kappa_{\mathrm{INT}}(\gamma) \approx \kappa_{\mathrm{FP}}(\gamma)$ and the acceptance length remains similar. We use this product as the compounding reference. In practice, $\kappa_{\mathrm{INT}}(\gamma) \gg \kappa_{\mathrm{FP}}(\gamma)$: as explained in (i), the verification delay $\frac{\mathrm{T}_{\mathrm{INT}}(\gamma)}{\mathrm{T}_{\mathrm{INT}}(1)}$ grows sharply once γ exceeds the load-match point, and the draft delay $\frac{\mathrm{D}_{\mathrm{FP}}(\gamma)}{\mathrm{T}_{\mathrm{INT}}(1)}$ is inflated by β because the same drafting latency is normalized by a shorter single-token latency. To measure how far the combined speedup falls from this reference, we define the compounding efficiency $\eta(\gamma) = \frac{\mathcal{S}_{\mathrm{SD(INT)}}(\gamma)}{\mathcal{S}_{\mathrm{SD}}(\gamma)\mathcal{S}_{\mathrm{INT}}} = \frac{\tau_{\mathrm{INT}}(\gamma)}{\tau_{\mathrm{FP}}(\gamma)} \cdot \frac{\kappa_{\mathrm{FP}}(\gamma)}{\kappa_{\mathrm{INT}}(\gamma)}$, which equals one when two speedups compound and decreases as quantization reduces the acceptance length or inflates the speculative overhead; Table 1 breaks it down into acceptance rate $\tau(\gamma)$, draft delay $\frac{D(\gamma)}{T(1)}$, and verification delay $\frac{T(\gamma)}{T(1)}$. Restoring $\kappa_{\mathrm{INT}}(\gamma) \approx \kappa_{\mathrm{FP}}(\gamma)$ is the goal of LoadMatch, which is explained in the following.

2.1 Load Match

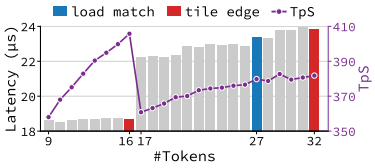


Figure 5: **LoadMatch ambiguity.**

We present LoadMatch, which finds an appropriate candidate-token count to align the memory and compute loads of quantized verification. In theory, the load-match point can be estimated from the memory bandwidth and compute throughput specifications. Nonetheless, these specifications do not directly reflect the execution of practical quantized GEMM kernels such as Marlin, QoQ, or APEX4. Therefore, we compute the load-match point from the actual verification latency. Specifically, we measure the latency with a single token and with a sufficiently large number of tokens γ_c (*e.g.*, 1024), which represent the memory- and compute-dominated regimes, respectively. Their ratio gives the number of tokens at which the two costs become comparable, $\hat{\gamma} = \mathrm{T}_{\mathrm{INT}}(1)\gamma_c/\mathrm{T}_{\mathrm{INT}}(\gamma_c)$. However, this estimate alone is not sufficient because quantized GEMM kernels process the token dimension in fixed-size tiles, causing several neighboring token counts to have similar verification latency (Frantar et al., 2025). This makes the exact load-match point ambiguous when $\hat{\gamma}$ lies inside a tiled region. For example, Marlin (Frantar et al., 2025) uses a tile size of 16 along the token dimension, so verifying 17 tokens can require nearly the same computation as verifying 32 tokens, as shown in Figure 5. Accordingly, we consider the lower and upper tile boundaries around $\hat{\gamma}$ (*e.g.*, 16 and 32) and evaluate their end-to-end throughput on a calibration set (*i.e.*, 80 randomly sampled sentences from the UltraChat-200K dataset). We select the point with higher throughput and use the chosen point γ^* across the tasks. See Algorithm 1 for details.

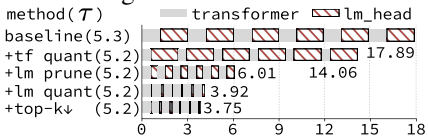


Figure 6: **Slimming analysis (ms).** `tf` is transformer and `lm` means `lm_head`.

Quantization also breaks another key assumption of speculative decoding, namely that the drafting cost is smaller than the cost of target model. This assumption holds for an FP16 target model, where the drafting latency $\mathrm{D}_{\mathrm{FP}}(\gamma)$ is small compared to the verification latency $\mathrm{T}_{\mathrm{FP}}(\gamma)$. Once the target model is quantized, however, verification is accelerated by a factor of β while the drafting latency remains unchanged, so the relative draft overhead $\frac{\mathrm{D}_{\mathrm{FP}}(\gamma)}{\mathrm{T}_{\mathrm{INT}}(1)}$ grows and inflates the speculative overhead $\kappa_{\mathrm{INT}}(\gamma)$. A straightforward remedy is to quantize the drafter as well. Since the drafters are also built on a transformer architecture, they can use the same widely adopted post-training quantization (PTQ) schemes (Frantar et al., 2023; Lin et al., 2024) as the target model. However, quantizing only the transformer blocks of the drafters

is not sufficient. Drafters (Cai et al., 2024; Ankner et al., 2024; Li et al., 2025b) interleave a final fully-connected (FC) layer, called `lm_head`, whose cost cannot be ignored once the transformer blocks are accelerated. To address this, we jointly apply quantization and pruning to `lm_head`. Prior works (Zhao et al., 2025b; Li et al., 2025b) reduce its cost through frequency-based pruning that removes less frequent word vectors; we take this one step further by additionally quantizing the pruned layers and controlling the width of draft tree, driving their cost down to a negligible level. Together with the quantized transformer blocks, this drafter slimming restores the cheap drafting assumption for quantized speculative decoding, as shown in Figure 6.

2.2 Advanced Baseline on vLLM Framework

Table 2: **Accelerator framework analysis.**

Framework	Method	TpS	Speedup	τ
HuggingFace	AR	25	1.00×	1.00
HuggingFace	SD (slow)	106	4.29×	5.71
vLLM (non-eager)	AR	42	1.00×	1.00
vLLM (non-eager)	SD (slow)	106	2.58×	5.71
vLLM (non-eager)	SD (fast)	136	3.23×	5.71

We build the advanced autoregressive and speculative decoding baseline for reporting the realistic speedups of LoadMatch. The existing speculative decoding studies have achieved speedups over autoregressive decoding, but the magnitude of these gains varies considerably across inference accelerator frameworks. Since our goal is to improve efficiency under a practical scenario, we use the widely adopted vLLM framework, and measure speedups within this environment. As shown in Table 2, we evaluate Llama3-8B with EAGLE-3 on the MT-Bench dataset. While vLLM improves the throughput of the AR baseline, SD (slow) benefits little, reducing its relative speedup. Therefore, we further improve the efficiency of the baseline SD (slow) by enabling CUDA Graph capture (fast). For this, we vectorize complex control-flow branches previously implemented with if-conditions and replace non-deterministic operations with deterministic counterparts for static replay. These changes allow the entire drafting, verification, and KV-cache update pipeline to be captured by CUDA Graphs, resulting in a reasonable baseline. In experimental results, we use this strong baseline as a comparison target.

3 Experimental Results

3.1 Experimental Setup

Dataset. We evaluate the proposed method across the various domains, including science with GPQA (Rein et al., 2024); math with GSM8K (Cobbe et al., 2021), MATH500 (Lightman et al., 2024), and AIME25; code with HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and LiveCodeBench (LCB) (Jain et al., 2025); chat with MT-Bench (Zheng et al., 2023) and Alpaca;

Table 3: **Speedup evaluation.** Speedup is measured relative to FP16 autoregressive decoding, and τ denotes the acceptance length, across math, code, chat, and general NLP tasks.

Model	Method	Math & Code				Chat				General NLP				Overall	
		GSM8K		HumanEval		MT-Bench		Alpaca		Summary		QA		*Avg.*	
Temp. = 0		Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ
L3-8B	SD(FP16)	3.32×	5.86	3.91×	5.70	3.23×	5.71	3.48×	6.44	2.76×	5.10	2.90×	5.34	3.27×	5.69
	AR(INT4)	2.85×	1.00	2.83×	1.00	2.78×	1.00	2.83×	1.00	2.89×	1.00	2.82×	1.00	2.83×	1.00
	SD(INT4)	3.87×	5.88	3.73×	5.74	3.67×	5.59	4.20×	6.47	3.30×	5.11	3.42×	5.05	3.70×	5.64
	LM(INT4)	9.35×	5.20	10.03×	5.53	8.46×	4.91	9.52×	5.74	7.61×	4.35	7.90×	4.36	8.81×	5.02
DS-8B	SD(FP16)	3.90×	5.68	3.52×	5.12	2.99×	5.31	3.04×	5.42	2.08×	4.16	2.81×	4.95	3.06×	5.11
	AR(INT4)	2.83×	1.00	2.81×	1.00	2.79×	1.00	2.81×	1.00	2.80×	1.00	2.81×	1.00	2.81×	1.00
	SD(INT4)	3.79×	6.13	4.00×	5.78	3.51×	5.21	3.55×	5.26	3.30×	5.10	3.10×	4.93	3.54×	5.40
	LM(INT4)	10.32×	5.61	8.91×	4.97	7.62×	4.42	7.43×	4.29	5.20×	3.25	6.85×	4.14	7.72×	4.45
Temp. = 1		Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ
L3-8B	SD(FP16)	2.30×	5.21	2.96×	5.46	1.79×	4.26	2.62×	5.57	2.30×	4.34	1.39×	3.94	2.23×	4.80
	AR(INT4)	2.79×	1.00	2.79×	1.00	2.77×	1.00	2.79×	1.00	2.76×	1.00	2.79×	1.00	2.78×	1.00
	SD(INT4)	2.04×	5.05	3.23×	5.24	1.81×	4.19	2.90×	5.56	2.87×	4.43	1.63×	3.81	2.42×	4.71
	LM(INT4)	5.10×	4.32	7.53×	4.76	4.59×	3.41	6.72×	4.56	5.91×	3.64	3.77×	3.15	5.60×	3.97
DS-8B	SD(FP16)	3.01×	5.68	2.80×	5.07	2.35×	4.41	2.44×	4.55	1.89×	3.79	2.15×	4.24	2.44×	4.62
	AR(INT4)	2.79×	1.00	2.78×	1.00	2.79×	1.00	2.79×	1.00	2.79×	1.00	2.79×	1.00	2.79×	1.00
	SD(INT4)	3.04×	5.59	3.33×	5.00	2.69×	4.20	2.65×	4.29	2.02×	3.36	2.21×	4.12	2.66×	4.43
	LM(INT4)	7.95×	4.92	6.80×	4.07	5.54×	3.47	5.40×	3.51	4.35×	2.90	5.25×	3.35	5.88×	3.70

Table 4: **Comprehensive evaluation.** Accuracy, model memory, decoding speedup, and acceptance length τ for L3/DS-8B under GPTQ and AWQ quantization, across science, math, and code tasks. Acc is pass@1 for HumanEval and MBPP and exact-match accuracy elsewhere. Mem denotes model weight memory footprint (GB). Speedup is relative to FP16 autoregressive decoding.

Model Config	Method	Mem	SCIENCE			MATH						CODE						OVERALL		
			GPQA			GSM8K			MATH500			HumanEval			MBPP			*Avg.*		
	Temp. = 0		Acc	Speedup	τ	Acc	Speedup	τ	Acc	Speedup	τ	Acc	Speedup	τ	Acc	Speedup	τ	Acc	Speedup	τ
L3-8B FP16	AR(FP16)	15.0 GB	15.2	1.00×	1.00	85.2	1.00×	1.00	50.0	1.00×	1.00	67.7	1.00×	1.00	62.5	1.00×	1.00	56.1	1.00×	1.00
	SD(FP16)	15.8 GB	15.2	2.98×	4.39	85.2	3.32×	5.86	50.0	3.22×	4.68	67.7	3.91×	5.70	62.5	3.97×	5.76	56.1	3.48×	5.28
L3-8B GPTQ	AR(INT4)	5.3 GB	18.2	2.82×	1.00	82.0	2.85×	1.00	50.0	2.81×	1.00	65.9	2.83×	1.00	58.6	2.83×	1.00	54.9	2.83×	1.00
	SD(INT4)	6.1 GB	18.2	3.11×	4.80	82.0	3.87×	5.88	50.0	3.48×	5.20	65.9	3.73×	5.74	58.6	3.76×	5.80	54.9	3.59×	5.48
	LM(INT4)	5.6 GB	18.2	7.82×	4.36	82.0	9.35×	5.20	50.0	8.59×	4.77	65.9	10.03×	5.53	58.6	10.20×	5.54	54.9	9.20×	5.08
L3-8B AWQ	AR(INT4)	5.4 GB	16.2	2.79×	1.00	87.5	2.84×	1.00	48.4	2.80×	1.00	66.5	2.82×	1.00	59.4	2.81×	1.00	55.6	2.81×	1.00
	SD(INT4)	6.2 GB	16.2	3.30×	4.76	87.5	3.51×	5.61	48.4	3.57×	5.13	66.5	3.70×	5.64	59.4	3.72×	5.37	55.6	3.56×	5.30
	LM(INT4)	5.6 GB	16.2	7.86×	4.43	87.5	9.30×	5.00	48.4	8.16×	4.56	66.5	9.74×	5.45	59.4	9.43×	5.23	55.6	8.90×	4.93
DS-8B FP16	AR(FP16)	15.0 GB	3.0	1.00×	1.00	78.9	1.00×	1.00	58.6	1.00×	1.00	41.5	1.00×	1.00	42.2	1.00×	1.00	44.8	1.00×	1.00
	SD(FP16)	15.8 GB	3.0	2.90×	4.22	78.9	3.90×	5.68	58.6	4.05×	5.95	41.5	3.52×	5.12	42.2	3.54×	5.13	44.8	3.58×	5.22
DS-8B GPTQ	AR(INT4)	5.3 GB	3.0	2.81×	1.00	72.7	2.83×	1.00	54.7	2.82×	1.00	39.0	2.81×	1.00	39.1	2.81×	1.00	41.7	2.82×	1.00
	SD(INT4)	6.1 GB	3.0	2.95×	4.84	72.7	3.79×	6.13	54.7	4.19×	6.32	39.0	4.00×	5.78	39.1	4.04×	5.83	41.7	3.79×	5.78
	LM(INT4)	5.6 GB	3.0	7.34×	4.11	72.7	10.32×	5.61	54.7	10.39×	5.75	39.0	8.91×	4.97	39.1	8.94×	4.95	41.7	9.18×	5.08
DS-8B AWQ	AR(INT4)	5.4 GB	4.0	2.82×	1.00	75.8	2.80×	1.00	58.6	2.81×	1.00	39.6	2.81×	1.00	40.6	2.81×	1.00	43.7	2.81×	1.00
	SD(INT4)	6.2 GB	4.0	3.34×	4.80	75.8	3.87×	6.10	58.6	4.30×	6.35	39.6	3.99×	5.75	40.6	3.75×	5.85	43.7	3.85×	5.77
	LM(INT4)	5.6 GB	4.0	7.46×	4.15	75.8	10.11×	5.56	58.6	10.29×	5.77	39.6	8.89×	4.97	40.6	8.94×	4.91	43.7	9.14×	5.07

Table 5: **Concurrency evaluation.** Decoding throughput (TpS, tokens/s) and speedup over FP16 autoregressive decoding at the same model and batch size.

DOMAIN	Task	Method	Llama3-8B								DS-8B							
			BS = 1		BS = 2		BS = 4		BS = 8		BS = 1		BS = 2		BS = 4		BS = 8	
	Temp. = 0		TpS	Speedup	TpS	Speedup	TpS	Speedup	TpS	Speedup	TpS	Speedup	TpS	Speedup	TpS	Speedup	TpS	Speedup
SCIENCE	GPQA	AR(FP16)	42	1.00×	79	1.00×	144	1.00×	256	1.00×	42	1.00×	82	1.00×	162	1.00×	314	1.00×
		SD(FP16)	125	2.98×	199	2.51×	351	2.44×	578	2.25×	122	2.90×	194	2.37×	365	2.26×	614	1.96×
		AR(INT4)	118	2.82×	220	2.77×	390	2.71×	661	2.58×	118	2.81×	228	2.79×	438	2.71×	817	2.60×
		SD(INT4)	131	3.11×	211	2.65×	411	2.86×	740	2.88×	124	2.95×	234	2.87×	434	2.68×	835	2.66×
		LM(INT4)	328	7.82×	493	6.21×	744	5.17×	1038	4.05×	308	7.34×	497	6.08×	751	4.65×	1088	3.47×
MATH	GSM8K	AR(FP16)	42	1.00×	83	1.00×	163	1.00×	308	1.00×	42	1.00×	82	1.00×	161	1.00×	295	1.00×
		SD(FP16)	139	3.32×	224	2.69×	434	2.66×	767	2.49×	164	3.90×	244	2.96×	463	2.87×	775	2.63×
		AR(INT4)	120	2.85×	234	2.82×	446	2.73×	686	2.23×	119	2.83×	232	2.82×	449	2.78×	829	2.81×
		SD(INT4)	163	3.87×	263	3.16×	454	2.78×	796	2.59×	159	3.79×	279	3.38×	555	3.44×	1070	3.63×
		LM(INT4)	393	9.35×	600	7.22×	929	5.69×	1220	3.96×	433	10.32×	639	7.75×	1004	6.22×	1454	4.93×
	MATH500	AR(FP16)	42	1.00×	82	1.00×	160	1.00×	286	1.00×	42	1.00×	82	1.00×	161	1.00×	314	1.00×
		SD(FP16)	135	3.22×	215	2.62×	404	2.53×	650	2.27×	170	4.05×	248	3.02×	481	2.98×	823	2.62×
		AR(INT4)	118	2.81×	224	2.73×	418	2.61×	734	2.57×	118	2.82×	230	2.79×	440	2.73×	828	2.64×
		SD(INT4)	146	3.48×	248	3.01×	470	2.94×	895	3.13×	176	4.19×	257	3.12×	492	3.06×	1067	3.40×
		LM(INT4)	361	8.59×	563	6.84×	842	5.27×	1223	4.27×	436	10.39×	636	7.73×	993	6.16×	1433	4.57×
CODE	HumanEval	AR(FP16)	42	1.00×	81	1.00×	159	1.00×	295	1.00×	42	1.00×	82	1.00×	161	1.00×	312	1.00×
		SD(FP16)	164	3.91×	243	2.99×	463	2.91×	765	2.59×	148	3.52×	221	2.69×	417	2.59×	714	2.29×
		AR(INT4)	119	2.83×	228	2.81×	433	2.72×	785	2.66×	118	2.81×	229	2.79×	436	2.70×	815	2.61×
		SD(INT4)	157	3.73×	288	3.54×	533	3.35×	971	3.29×	168	4.00×	223	2.71×	425	2.64×	846	2.71×
		LM(INT4)	421	10.03×	639	7.86×	976	6.13×	1380	4.67×	374	8.91×	570	6.93×	885	5.49×	1288	4.13×
	MBPP	AR(FP16)	42	1.00×	83	1.00×	162	1.00×	289	1.00×	42	1.00×	83	1.00×	158	1.00×	285	1.00×
		SD(FP16)	167	3.97×	250	3.02×	488	3.01×	785	2.71×	149	3.54×	224	2.72×	425	2.69×	716	2.51×
		AR(INT4)	119	2.83×	234	2.83×	445	2.74×	810	2.80×	118	2.81×	231	2.79×	424	2.68×	780	2.74×
		SD(INT4)	158	3.76×	224	2.70×	520	3.20×	780	2.70×	170	4.04×	257	3.11×	459	2.91×	866	3.04×
		LM(INT4)	428	10.20×	640	7.72×	971	5.98×	1371	4.74×	375	8.94×	572	6.93×	882	5.59×	1245	4.37×

and general NLP with CNN/Daily Mail (Nallapati et al., 2016) and Natural Questions (Kwiatkowski et al., 2019), which are used for summarization and question answering (QA), respectively.

Protocol. We assess the proposed method in terms of speedup ratio and acceptance length τ. Speedup ratio measures the improvement of tokens per second (TpS) over FP16 autoregressive decoding method on the same hardware and batch size. The acceptance length τ denotes the number of tokens committed per verification step, including the bonus token from the target, so $\tau = 1$ for autoregressive decoding. SD(FP16) and SD(INT4) retain the FP16 draft head with the original candidate-token count of the drafter; LM(INT4) uses the slimmed INT4 draft head with the calibrated count γ^*. Unless stated otherwise, the batch size is one and the GPU is a power-capped

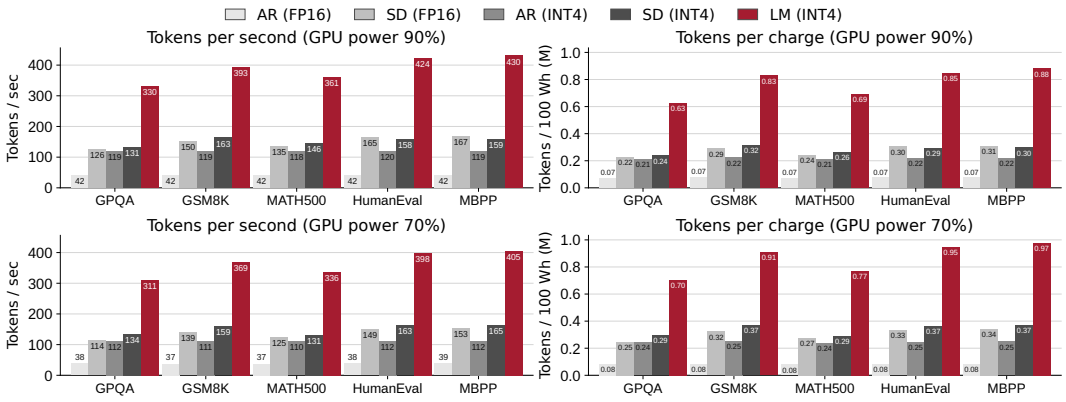


Figure 7: **Low-power energy efficiency evaluation.** This analysis employs Llama3-8B equipped with an EAGLE-3 draft head, under greedy decoding with a batch size of one. The tokens per charge represents the number of tokens (in million) generated from a 100 Wh battery (360 kJ).

A5000 (210W, Section C). We choose the widely adopted GPTQ (Frantar et al., 2023) method to quantize the model. For practical evaluation, we employ the vLLM as the inference engine. We set the maximum generation length to 2048, except for Table 3, where we use 1024 following previous work (Li et al., 2025b). A detailed experimental setup and more results exist in Sections B and E.

Baseline methods. We apply the proposed method to three recently proposed drafters: autoregressive of EAGLE-3 (Li et al., 2025b), and diffusion of DFlash (Chen et al., 2026) and DOMINO (Huang et al., 2026). We use Llama3-8B (Dubey et al., 2024), DeepSeek-R1-distill-Llama-8B (Guo et al., 2025a), Qwen3-4/8B (Yang et al., 2025), and Qwen3.5-27B (Team, 2026).

3.2 Experimental Results

Speedup evaluation. We compare five decoding settings in Table 3: FP16 autoregressive decoding, FP16 speculative decoding, autoregressive decoding with an INT4 target model, speculative decoding with an INT4 target model, and LoadMatch with both the target and draft models quantized to INT4. Among these settings, our method achieves the favorable throughput speedup, confirming that LoadMatch facilitates the compounding effect of quantization and speculative decoding.

Comprehensive evaluation. Table 4 evaluates the proposed method across multiple metrics, including accuracy, weight memory footprint, and speedup. INT4 quantization changes accuracy by a small amount on the 8B models and by up to five points on average for Qwen3-4B (Table 10) while substantially reducing the memory and improving throughput; since speculative decoding does not alter the output of the target, SD(INT4) and LM(INT4) have the accuracy of AR(INT4). In particular, because our method quantizes both the target and draft models, it achieves a lower memory footprint than the target-only quantization approach.

Concurrency evaluation. As shown in Table 5, we observe that our LoadMatch remains effective in batch inference scenarios. More results are placed in Section E.

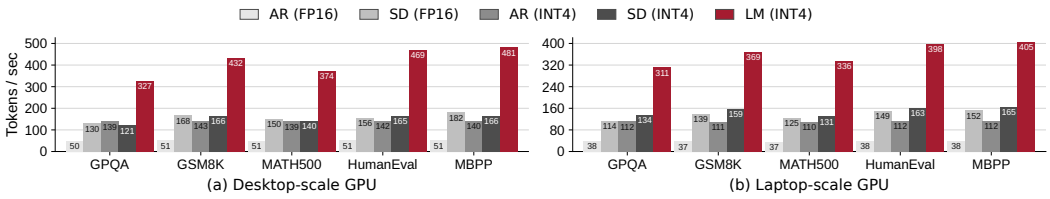


Figure 8: **Cross device evaluation.** We evaluate LoadMatch on desktop-scale (RTX 3090) and laptop-scale (power-capped A5000) GPUs. For this experiment, we use the Llama3-8B model with an EAGLE-3 draft head, with a batch size of one and greedy decoding.

Table 6: **Drafter evaluation.** Decoding performance across mathematics, code, and chat tasks. Speedup is measured relative to FP16 autoregressive decoding, and τ is the acceptance length.

Model	Method	MATH						CODE						CHAT				OVERALL	
		GSM8K		MATH500		AIME25		HumanEval		MBPP		LCB		MT-Bench		Alpaca		*Avg.*	
Temp. = 0		Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ
Q3-4B	INT4	2.38×	1.00	2.38×	1.00	2.35×	1.00	2.38×	1.00	2.38×	1.00	2.41×	1.00	2.41×	1.00	2.37×	1.00	2.38×	1.00
	EAGLE-3	1.45×	3.97	1.34×	3.73	1.36×	3.67	1.31×	3.60	1.29×	3.52	1.17×	3.52	1.10×	3.70	1.24×	3.75	1.28×	3.68
	+LM	4.67×	3.31	4.38×	3.13	4.18×	3.07	4.33×	3.06	4.16×	2.98	4.01×	3.00	4.25×	3.08	4.00×	3.16	4.25×	3.10
	DFlash	2.67×	5.93	3.43×	7.39	3.17×	6.70	2.74×	6.05	2.50×	5.49	2.83×	6.59	1.54×	3.28	1.28×	2.79	2.52×	5.53
	+LM	8.94×	5.92	10.89×	7.26	10.29×	6.96	8.87×	5.91	8.39×	5.32	7.20×	6.23	5.12×	3.31	4.41×	2.80	8.01×	5.46
	DOMINO	2.89×	8.49	2.84×	7.61	2.51×	6.89	2.25×	6.41	2.12×	6.18	2.08×	6.41	1.36×	3.81	1.21×	3.43	2.16×	6.15
	+LM	10.66×	8.32	10.47×	8.00	8.80×	6.89	8.32×	6.40	8.03×	5.97	6.16×	5.97	5.42×	4.09	4.68×	3.45	7.82×	6.14
Q3-8B	INT4	2.66×	1.00	2.69×	1.00	2.60×	1.00	2.68×	1.00	2.68×	1.00	2.66×	1.00	2.67×	1.00	2.65×	1.00	2.66×	1.00
	EAGLE-3	2.30×	3.82	2.22×	3.64	2.09×	3.54	2.24×	3.72	2.09×	3.45	2.17×	3.35	1.85×	3.37	1.60×	3.19	2.07×	3.51
	+LM	5.38×	3.17	5.09×	3.03	4.80×	2.92	5.36×	3.15	4.98×	2.89	5.22×	2.86	4.52×	2.84	4.33×	2.70	4.96×	2.94
	DFlash	4.80×	5.98	6.32×	7.62	5.76×	7.06	4.99×	6.15	4.55×	5.69	4.96×	6.77	2.97×	3.57	2.32×	2.83	4.58×	5.71
	+LM	10.39×	6.02	12.97×	7.43	12.02×	7.04	10.81×	6.25	10.19×	5.70	9.59×	6.74	6.01×	3.44	5.03×	2.79	9.62×	5.68
	DOMINO	5.09×	8.80	5.28×	8.46	4.64×	7.40	4.22×	6.83	3.98×	6.52	3.93×	6.82	2.64×	4.23	2.24×	3.56	4.00×	6.58
	+LM	13.06×	8.64	12.65×	8.14	11.04×	7.26	10.60×	6.92	10.28×	6.48	8.84×	6.75	6.26×	4.00	5.67×	3.56	9.79×	6.47
Temp. = 1		Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ
Q3-4B	INT4	2.33×	1.00	2.37×	1.00	2.32×	1.00	2.38×	1.00	2.38×	1.00	2.31×	1.00	2.41×	1.00	2.33×	1.00	2.35×	1.00
	EAGLE-3	1.30×	3.82	1.23×	3.56	1.11×	3.21	1.24×	3.53	1.18×	3.46	0.96×	3.51	1.18×	3.51	0.93×	3.69	1.14×	3.53
	+LM	4.03×	3.19	3.72×	3.00	3.35×	2.72	3.78×	3.01	3.68×	2.93	3.58×	2.98	3.55×	2.92	3.49×	3.11	3.64×	2.98
	DFlash	2.40×	5.30	2.67×	5.62	1.93×	4.02	2.50×	5.44	2.24×	4.83	2.53×	5.85	1.41×	3.02	1.21×	2.66	2.11×	4.59
	+LM	7.65×	5.16	8.00×	5.45	6.10×	4.34	7.84×	5.29	7.22×	4.74	6.73×	5.75	4.77×	3.14	3.99×	2.61	6.53×	4.56
	DOMINO	2.39×	7.10	2.01×	5.58	1.56×	4.25	1.98×	5.62	1.80×	5.12	2.05×	6.16	1.26×	3.48	1.15×	3.22	1.77×	5.06
	+LM	8.85×	6.94	7.25×	5.66	5.24×	4.24	7.27×	5.64	6.84×	5.15	5.88×	5.50	4.67×	3.57	4.12×	3.13	6.26×	4.98
Q3-8B	INT4	2.63×	1.00	2.70×	1.00	2.56×	1.00	2.63×	1.00	2.66×	1.00	2.57×	1.00	2.63×	1.00	2.61×	1.00	2.63×	1.00
	EAGLE-3	2.07×	3.68	1.90×	3.39	1.68×	3.05	2.02×	3.54	1.90×	3.30	1.71×	3.22	1.41×	3.16	1.68×	3.09	1.80×	3.30
	+LM	4.71×	3.04	4.47×	2.84	3.95×	2.56	4.66×	2.98	4.34×	2.77	4.05×	2.75	3.85×	2.66	3.83×	2.61	4.24×	2.78
	DFlash	4.13×	5.08	4.42×	5.25	3.49×	4.25	4.18×	5.08	3.78×	4.72	3.99×	5.49	2.39×	2.90	2.08×	2.53	3.56×	4.41
	+LM	8.57×	4.99	9.43×	5.46	6.87×	4.16	8.56×	5.04	8.08×	4.64	7.31×	5.32	5.02×	2.91	4.45×	2.52	7.28×	4.38
	DOMINO	4.08×	6.72	3.40×	5.37	2.61×	4.25	3.32×	5.40	2.92×	4.84	2.96×	5.18	2.04×	3.29	1.91×	3.03	2.90×	4.76
	+LM	9.58×	6.27	8.35×	5.44	6.16×	4.17	8.18×	5.37	7.73×	5.00	6.38×	5.04	4.91×	3.23	4.70×	3.02	6.99×	4.69

Table 7: **Thinking mode evaluation.** Speedup over FP16 autoregressive decoding and acceptance length τ, with thinking mode enabled and disabled.

DOMAIN	Task	Method	Qwen3-4B				Qwen3-8B			
			Thinking = off		Thinking = on		Thinking = off		Thinking = on	
Temp. = 0			Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ
SCI	GPQA	INT4	2.38×	1.00	2.35×	1.00	2.64×	1.00	2.29×	1.00
		DFlash	2.79×	5.83	1.53×	3.32	3.42×	5.43	2.52×	3.13
		+LM	8.34×	5.61	4.76×	3.31	9.46×	5.47	5.25×	3.17
MATH	GSM8K	INT4	2.38×	1.00	2.34×	1.00	2.66×	1.00	2.75×	1.00
		DFlash	2.67×	5.93	1.65×	3.59	4.80×	5.98	2.78×	3.48
		+LM	8.94×	5.92	5.26×	3.54	10.39×	6.02	5.95×	3.46
	MATH500	INT4	2.38×	1.00	2.33×	1.00	2.69×	1.00	2.49×	1.00
		DFlash	3.43×	7.39	1.69×	3.66	6.32×	7.62	2.92×	3.63
		+LM	10.89×	7.26	5.32×	3.63	12.97×	7.43	6.10×	3.60
CODE	HumanEval	INT4	2.38×	1.00	2.34×	1.00	2.68×	1.00	2.42×	1.00
		DFlash	2.74×	6.05	1.51×	3.31	4.99×	6.15	2.58×	3.26
		+LM	8.87×	5.91	4.73×	3.28	10.81×	6.25	5.41×	3.13
	MBPP	INT4	2.38×	1.00	2.36×	1.00	2.68×	1.00	2.42×	1.00
		DFlash	2.50×	5.49	1.58×	3.37	4.55×	5.69	2.58×	3.18
		+LM	8.39×	5.32	5.15×	3.39	10.19×	5.70	5.64×	3.23

Table 8: **Long-context evaluation.** We vary context length on the LB dataset, using Qwen3.5-27B with DFlash.

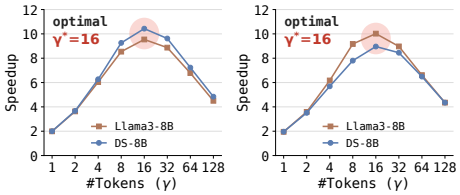
Context	Method	Qwen3.5-27B					
		HotpotQA		Qasper		GovReport	
Temp. = 0		Speedup	τ	Speedup	τ	Speedup	τ
0.5K	INT4	3.85×	1.00	4.73×	1.00	4.72×	1.00
	DFlash	2.63×	4.60	4.64×	5.15	2.16×	3.47
	+LM	9.09×	4.60	8.27×	5.10	8.34×	3.43
1K	INT4	3.83×	1.00	3.31×	1.00	4.52×	1.00
	DFlash	2.78×	4.50	4.10×	5.26	2.51×	3.74
	+LM	9.30×	4.51	11.06×	5.21	8.04×	3.67
2K	INT4	3.84×	1.00	3.82×	1.00	4.51×	1.00
	DFlash	2.77×	4.57	2.82×	5.22	2.37×	3.67
	+LM	8.36×	4.60	9.27×	5.09	7.66×	3.62
4K	INT4	3.74×	1.00	3.97×	1.00	4.46×	1.00
	DFlash	2.75×	4.58	3.17×	5.36	2.50×	3.67
	+LM	8.07×	4.61	9.52×	5.16	7.45×	3.56
Avg.	INT4	3.81×	1.00	3.96×	1.00	4.55×	1.00
	DFlash	2.73×	4.56	3.68×	5.25	2.38×	3.64
	+LM	8.71×	4.58	9.53×	5.14	7.87×	3.57

Low-power energy efficiency evaluation. We report the number of generated tokens using the limited battery in Figure 7. We adopt the recent AI-laptop power configurations for the GPU power cap (*e.g.*, 170-210 W) and the budget (*e.g.*, 100 Wh battery). In this configuration, we find that the proposed method reduces the energy usage in the low-powered GPU while maintaining the speedup.

Cross device evaluation. Figure 8 compares decoding throughput on desktop- and a laptop-scale GPUs under constrained compute and power budgets. While quantization and speculative decoding

Table 9: **Controlled factor analysis.** We disentangle two factors of ours: drafter slimming (Slim) and LoadMatch (LM). All variants are evaluated under the same protocol; only the factor specified in table is added.

Model	Method	Sci		Math				Code			
		GPQA		GSM8K		MATH500		HumanEval		MBPP	
	T= 0	Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ	Speedup	τ
L3-8B	Baseline	3.11×	4.80	3.87×	5.88	3.48×	5.20	3.73×	5.74	3.76×	5.80
	+Slim	5.25×	4.75	6.46×	5.59	5.80×	5.15	6.42×	5.69	6.54×	5.73
	+Slim+LM	7.82×	4.36	9.35×	5.20	8.59×	4.77	10.03×	5.53	10.20×	5.54
DS-8B	Baseline	2.95×	4.84	3.79×	6.13	4.19×	6.32	4.00×	5.78	4.04×	5.83
	+Slim	5.05×	4.71	6.99×	6.08	6.97×	6.25	6.17×	5.73	6.45×	5.73
	+Slim+LM	7.34×	4.11	10.32×	5.61	10.39×	5.75	8.91×	4.97	8.94×	4.95



(a) Math (GSM8K) (b) Code (HumanEval)

Figure 9: **#Tokens (γ) analysis.** Speedup over FP16 autoregressive decoding on the math and code benchmarks as the number of candidate tokens varies.

each improve throughput, naively combining them yields little additional gain. Our LoadMatch alleviates the overhead of speculative decoding, achieving the favorable throughput on both devices.

Drafter evaluation. We present a systematic evaluation of our method across different drafters, while keeping all other experimental factors fixed in Table 6. Specifically, we apply LoadMatch to the tree-based EAGLE-3 and the diffusion-based DFlash and DOMINO drafters. LoadMatch works well across both drafter types, demonstrating its robustness to the choice of drafter.

Thinking mode evaluation. To evaluate robustness under reasoning-intensive generation, we apply the proposed method to Qwen3 with thinking mode enabled in Table 7. Despite the more challenging decoding dynamics, LoadMatch consistently improves speedup over baseline under different model and task settings while maintaining a comparable acceptance length.

Long-context evaluation. We employ the LongBench (LB) (Bai et al., 2024) dataset to assess the proposed method across different context length in Table 8, using the Qwen3.5-27B model. Because the FP16 model of this size does not fit on a single 24 GB GPU, the FP16 runs on four GPUs whereas all INT4 variants run on two (Section C). LoadMatch works well under the diverse context lengths.

4 Analysis

Factor analysis. Table 9 applies the proposed factors in a step-wise manner to observe their effect on the speedup. We use EAGLE-3 as the baseline and keep every other factor the same for a controlled analysis. We confirm that our factors favorably improve the baseline across different tasks.[1]

#Tokens (γ) analysis. We report the correlation between the candidate-token count and the speedup of the proposed method in Figure 9. The results show that finding the optimal γ^* is critical, since the number of tokens γ considerably affects the memory-compute trade-off in quantized speculative decoding. Moreover, the optimal number of tokens is rather stable across different tasks.

5 Conclusion

This paper studies the integration of quantization and speculative decoding for running LLMs on devices whose memory, compute, and battery are far more constrained than those of servers. Naively applying speculative decoding to a quantized LLM yields little additional speedup, which we attribute to the load match point shift problem: quantization shrinks the memory time but not the compute time, so the candidate-token count that keeps an FP16 model memory-bound pushes the quantized model into the compute-bound regime, where every additional token slows verification. To address this problem, we present LoadMatch, which restores the balance by calibrating the load-match point of the deployed hardware from a short latency profile and refining the candidate-token count by measured throughput, without retraining. In the experimental results, we observe that LoadMatch favorably improves LLM inference speed and energy efficiency on diverse benchmarks while maintaining accuracy and memory usage comparable to the baseline, and we hope this work can serve as a milestone for researchers and practitioners in the on-device AI community.

[1] Related work is deferred to Section A.

AI USE STATEMENT

In this work, we have not used generative AI tools for "develop theoretical models or conceptual frameworks", "formulate mathematical claims", "provide critical ingredients for proving mathematical claims", "assist in the writing of proofs", "propose or refine hypotheses", "design or provide feedback on research methodology or experiments", "implement methods", "assist with translation", "support qualitative and thematic data analysis", "interpret results", and "generate synthetic data sets" and "clean and reformat dataset" are not applicable to this work. Additionally, we used generative AI tools for "edit software code" and "edit a research paper to improve readability". We have reviewed all AI-assisted work. For example, LLM-edited code was manually verified and tested for correctness by the authors. AI-assisted writing was mostly used for improving the language and fixing grammatical errors, ensuring that the entire content of the manuscript was originally written and carefully edited by the authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

The authors of this paper have read and understood the ICLR Code of Ethics in detail. After a self-evaluation of the original manuscript, the authors confirm that this paper does not hold any potential ethical concerns, including but not limited to studies involving human subjects, practices related to dataset releases, potentially harmful insights, methodologies, and applications, potential conflicts of interest and sponsorship, discrimination/bias/fairness concerns, privacy and security issues, legal compliance, and research integrity issues.

REPRODUCIBILITY STATEMENT

We have described the general experimental setups (*e.g.*, dataset, model, evaluation protocols) in Section 3.1. For more details, we explain the hyperparameter choices of baseline and ours, and evaluation metrics in Section B. We describe software and hardware setups that we employ to run experiments in Section C. For algorithm, we leave the pseudo code of our method in Section D. Upon the acceptance, the code and checkpoints will be publicly available.

REFERENCES

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In *USENIX Symposium on Operating Systems Design and Implementation*, 2024.

Zachary Ankner, Rishab Parthasarathy, Aniruddha Nrusimha, Christopher Rinard, Jonathan Ragan-Kelley, and William Brandon. Hydra: Sequentially-dependent draft heads for medusa decoding. *Conference on Language Modeling*, 2024.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv*, 2021.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, et al. Longbench: A bilingual, multitask benchmark for long context understanding. In *Association for Computational Linguistics*, 2024.

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D Lee, Deming Chen, and Tri Dao. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *International Conference on Machine Learning*, 2024.

Kevin Kai-Wei Chang, Rachata Ausavarungnirun, Chris Fallin, and Onur Mutlu. Hat: Heterogeneous adaptive throttling for on-chip networks. In *IEEE International Symposium on Computer Architecture and High Performance Computing*, 2012.

Jian Chen, Yesheng Liang, and Zhijian Liu. Dflash: Block diffusion for flash speculative decoding. *International Conference on Machine Learning*, 2026.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv*, 2021.

Zhuoming Chen, Avner May, Ruslan Svirschevski, Yuhsun Huang, Max Ryabinin, Zhihao Jia, and Beidi Chen. Sequoia: Scalable, robust, and hardware-aware speculative decoding. *Advances in Neural Information Processing Systems*, 2024.

Hsiang-Yun Cheng, Chung-Hsiang Lin, Jian Li, and Chia-Lin Yang. Memory latency reduction via thread throttling. In *IEEE/ACM International Symposium on Microarchitecture*, 2010.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv*, 2021.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems*, 2022.

Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. Enhancing chat language models by scaling high-quality instructional conversations. In *Empirical Methods in Natural Language Processing*, 2023.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv*, 2024.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *International Conference on Learning Representations*, 2023.

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

Evangelos Georganas, Dhiraj Kalamkar, Alexander Kozlov, and Alexander Heinecke. Ml-specqd: Multi-level speculative decoding with quantized drafts. *arXiv*, 2025.

Jayanth Gummaraju and Mendel Rosenblum. Stream programming on general-purpose processors. In *IEEE/ACM International Symposium on Microarchitecture*, 2005.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv*, 2025a.

Hong Guo, Nianhui Guo, Weixing Wang, Jona Otholt, Christoph Meinel, and Haojin Yang. Apex4: Efficient pure w4a4 llm inference via intra-sm compute rebalancing. *arXiv*, 2026.

Tianyu Guo, Xianwei Zhang, Jiangsu Du, Zhiguang Chen, Nong Xiao, and Yutong Lu. Gllm: Global balanced pipeline parallelism systems for distributed llms serving with token throttling. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2025b.

Jianuo Huang, Yaojie Zhang, Qituan Zhang, Hao Lin, Hanlin Xu, and Linfeng Zhang. Domino: Decoupling causal modeling from autoregressive drafting in speculative decoding. *arXiv*, 2026.

Naman Jain, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, 2025.

Onur Kayiran, Nachiappan Chidambaram Nachiappan, Adwait Jog, Rachata Ausavarungnirun, Mahmut T Kandemir, Gabriel H Loh, Onur Mutlu, and Chita R Das. Managing gpu concurrency in heterogeneous architectures. In *IEEE/ACM International Symposium on Microarchitecture*, 2014.

Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, et al. Natural questions: a benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 2019.

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

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, 2023.

Jindong Li, Tenglong Li, Guobin Shen, Dongcheng Zhao, Qian Zhang, and Yi Zeng. Pushing up to the limit of memory bandwidth and capacity utilization for efficient llm decoding on embedded fpga. In *Design, Automation and Test in Europe Conference*, 2025a.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle: Speculative sampling requires rethinking feature uncertainty. *International Conference on Machine Learning*, 2024a.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle-2: Faster inference of language models with dynamic draft trees. In *Empirical Methods in Natural Language Processing*, 2024b.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. Eagle-3: Scaling up inference acceleration of large language models via training-time test. *International Conference on Machine Learning*, 2025b.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *International Conference on Learning Representations*, 2024.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. In *Conference on Machine Learning and Systems*, 2024.

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

Jing Liu, Ruihao Gong, Xiuying Wei, Zhiwei Dong, Jianfei Cai, and Bohan Zhuang. Qllm: Accurate and efficient low-bitwidth quantization for large language models. In *International Conference on Learning Representations*, 2024.

Lian Liu, Long Cheng, Haimeng Ren, Zhaohui Xu, Yudong Pan, Mengdi Wang, Xiaowei Li, Yinhe Han, and Ying Wang. Comet: Towards practical w4a4kv4 llms serving. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2025.

Tianyu Liu, Qitan Lv, Yuhao Shen, Xiao Sun, and Xiaoyan Sun. Talon: Confidence-aware speculative decoding with adaptive token trees. *arXiv*, 2026.

Ramesh Nallapati, Bowen Zhou, Cicero Dos Santos, Çağlar Gulçehre, and Bing Xiang. Abstractive text summarization using sequence-to-sequence rnns and beyond. In *Conference on Computational Natural Language Learning*, 2016.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. Gpqa: A graduate-level google-proof q&a benchmark. *Conference on Language Modeling*, 2024.

Wenqi Shao, Mengzhao Chen, Zhaoyang Zhang, Peng Xu, Lirui Zhao, Zhiqian Li, Kaipeng Zhang, Gao Peng, Yu Qiao, and Ping Luo. Omniquant: Omnidirectionally calibrated quantization for large language models. In *International Conference on Learning Representations*, 2024.

Qwen Team. Qwen3.5-omni technical report. *arXiv*, 2026.

Rishabh Tiwari, Haocheng Xi, Aditya Tomar, Coleman Hooper, Sehoon Kim, Maxwell Horton, Mahyar Najibi, Michael W Mahoney, Kurt Keutzer, and Amir Gholami. Quantspec: Self-speculative decoding with hierarchical quantized kv cache. In *International Conference on Machine Learning*, 2025.

Jikai Wang, Yi Su, Juntao Li, Qingrong Xia, Zi Ye, Xinyu Duan, Zhefeng Wang, and Min Zhang. Opt-tree: Speculative decoding with adaptive draft tree structure. *Transactions of the Association for Computational Linguistics*, 2025.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 2009.

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

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

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *USENIX Symposium on Operating Systems Design and Implementation*, 2022.

Ying Zhang, Peng Zhang, Mincong Huang, Jingyang Xiang, Yujie Wang, Chao Wang, Yineng Zhang, Lei Yu, Chuan Liu, and Wei Lin. Qqq: Quality quattuor-bit quantization for large language models. *arXiv*, 2024.

Zongpu Zhang, Pranab Dash, Qiang Xu, Y Charlie Hu, Jian Li, and Haibing Guan. Rethinking dvfs for mobile llms: Unified energy-aware scheduling with core. *Conference on Machine Learning and Systems*, 2026.

Juntao Zhao, Wenhao Lu, Sheng Wang, Lingpeng Kong, and Chuan Wu. Qspec: Speculative decoding with complementary quantization schemes. In *Empirical Methods in Natural Language Processing*, 2025a.

Weilin Zhao, Tengyu Pan, Xu Han, Yudi Zhang, Ao Sun, Yuxiang Huang, Kaihuo Zhang, Weilun Zhao, Yuxuan Li, Jie Zhou, et al. Fr-spec: Accelerating large-vocabulary language models via frequency-ranked speculative sampling. In *Association for Computational Linguistics*, 2025b.

Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. Atom: Low-bit quantization for efficient and accurate llm serving. *Conference on Machine Learning and Systems*, 2024.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 2023.

A Related Work

A.1 Combining Speculative Decoding with Quantization

A few studies have explored merging the benefits of speculative decoding and quantization. A long-standing challenge in speculative decoding is preparation of an appropriate drafter: training a separate fast model can be time-consuming, whereas introducing an external model may introduce the distributional mismatch. To address the challenge, Zhao et al. (2025a) use a quantized target model as the drafter, which offers a promising alternative because it can be obtained efficiently while retaining relatively high accuracy. Similarly, Georganas et al. (2025) introduce multi-level speculative decoding with a quantized intermediate step between the target and draft models, which filters out unnecessary candidate tokens from the draft model and thereby improves the efficiency of target model computation. Moreover, Tiwari et al. (2025) propose hierarchical quantization to reduce key-value (KV) cache memory usage, enabling efficient long-context self-speculative decoding.

A.2 Load Matching in LLM Inference

Matching the compute load to the available memory bandwidth has been widely studied in system architecture to mitigate performance degradation caused by excessive parallelism (Gummaraju & Rosenblum, 2005; Chang et al., 2012; Kayiran et al., 2014). Prior studies have shown that maximizing concurrency does not always improve performance because excessive parallelism can saturate shared hardware resources such as memory bandwidth and on-chip interconnects. Accordingly, concurrency can be limited according to the available memory bandwidth to reduce resource contention and improve execution efficiency (Cheng et al., 2010). More recently, similar load-matching strategies have been applied to accelerate LLM inference. In LLM serving systems, Agrawal et al. (2024) controls the number of tokens processed together to balance workloads, while Guo et al. (2025b) presents a finer-grained scheduler that regulates the numbers of prefill and decode tokens. In on-device LLM, Zhang et al. (2026) dynamically adjusts hardware operating conditions, such as CPU, GPU, and memory frequencies, based on runtime statistics (*e.g.*, device temperature and utilization). However, limited attention has been given to matching the load of speculative decoding to the reduced memory load of quantized weights.

A.3 Efficient Draft Generation

Recent studies have focused on optimizing draft-tree structures to improve the efficiency of draft tokens. Although tree-based drafting enables multiple candidate continuations to be verified in parallel, many existing methods (Cai et al., 2024; Li et al., 2024a; Ankner et al., 2024) rely on fixed heuristic tree structures, which can be suboptimal because the acceptance probability of draft tokens varies across contexts and confidence levels. To address this limitation, Li et al. (2024b); Liu et al. (2026) replace static draft trees with context-aware dynamic structures that adapt their depth and width based on draft-model confidence. In addition, Wang et al. (2025) formulate draft-tree construction as an optimization problem that maximizes the expected acceptance length at each decoding step, while Chen et al. (2024) introduce a hardware-aware optimizer that selects the tree size and depth according to the underlying hardware characteristics. Sequoia is the closest method to our second stage because it also uses measured drafting and verification time to choose a hardware-aware speculation setting. Sequoia optimizes the tree structure and size for an FP16 target, while LoadMatch keeps the existing drafter and tree and adjusts only the number of tokens verified by a quantized target. LoadMatch first estimates the shifted load-match point using two latency measurements, and its selected γ^* can also serve as a size budget for tree-based optimizers. In parallel, diffusion LLMs (Chen et al., 2026; Huang et al., 2026) explore non-autoregressive draft generation for faster inference.

B EXPERIMENTAL DETAILS

B.1 TRAINING HYPERPARAMETERS

Our method involves two training processes: (1) draft head learning and (2) post-training quantization. For draft head learning, our method uses three different advanced draft head methods: (1) EAGLE-3 (Li et al., 2025b), (2) DFlash (Chen et al., 2026), and (3) DOMINO (Huang et al., 2026). Since our goal is to develop a plug-and-play algorithm that integrates existing draft heads into quantized LLMs without a significant amount of further training, we do not require extra training of draft head and simply use the publicly available draft head checkpoints, which are mostly released by the authors in their official project repositories. For post-training quantization, we adopt the widely used GPTQ (Frantar et al., 2023) and AWQ (Lin et al., 2024) methods to quantize the weights of both the LLMs and the draft heads, using 128 samples with a sequence length of 2048 from the UltraChat-200K dataset (Ding et al., 2023). Since the PTQ methods have very low computational complexity, the entire training process of our method can be completed quickly on a single GPU.

B.2 EVALUATION HYPERPARAMETERS

We evaluate the proposed method in terms of five metrics: (1) speedup, (2) acceptance length (τ), (3) memory footprint, (4) accuracy, and (5) the number of tokens generated using a 100 Wh battery. In the following, we describe the definition of each metric and the rationale behind its choice.

- **Speedup** measures the decoding throughput (tokens per second) of each method relative to the FP16 autoregressive decoding of the same target model, obtained under identical hardware and batch-size configurations. Because the absolute tokens-per-second value fluctuates considerably across models, tasks, and devices, we report the relative ratio rather than the raw throughput, which enables a fair and interpretable comparison of the acceleration attributable to each method. This serves as the primary metric of our work, as the ultimate goal of combining speculative decoding with model compression is to reduce end-to-end inference latency.
- **Acceptance length (τ)** denotes the average number of tokens committed per drafting-verification round, counting the accepted draft tokens plus the one token the target model always produces (the bonus token); hence $\tau = 1$ for autoregressive decoding and $\tau \leq d+1$ for a draft tree of depth d. We use this convention in every table. Unlike speedup, which depends on hardware characteristics and implementation details, τ provides a hardware-agnostic measure of draft quality: a higher τ indicates that the draft head predicts the target model's outputs more faithfully. Employing this metric allows us to disentangle the algorithmic contribution of our method from system-level factors, verifying that the observed speedup indeed stems from well-accepted drafts rather than from implementation artifacts. It also reveals whether compressing the draft head degrades its predictive ability.
- **Memory footprint** refers to the GPU memory occupied by the model weights only, covering both the target LLM and the draft head, expressed in gigabytes; it excludes the KV cache, activations, and framework buffers, so the peak GPU memory of a run is higher. Whether a model fits on memory-constrained hardware, such as consumer GPUs or edge devices, is directly determined by this metric, and such deployability constitutes one of the main motivations for combining quantization with speculative decoding. Presenting the memory footprint also exposes the additional overhead introduced by the draft head, so that readers can assess the trade-off between the acceleration benefit and the extra memory cost.
- **Accuracy** quantifies the downstream task performance of each configuration; we adopt pass@1 for the code generation benchmarks (HumanEval and MBPP) and exact-match accuracy for the remaining benchmarks. Note that speculative decoding is theoretically lossless with respect to the target model's output distribution, and hence any accuracy degradation observed in our setting originates from quantization of the target model. Tracking this metric confirms that the proposed method preserves the generation quality of the quantized target model.
- **The number of tokens generated using a 100 Wh battery** estimates how many tokens (in millions) can be produced within a fixed energy budget of 100 Wh (360 kJ). For practition-

ers, this metric directly translates the per-token energy cost into the amount of useful work obtainable from a battery of a typical laptop capacity. It becomes particularly relevant in on-device and low-power deployment scenarios, where the battery life, rather than the raw latency, often imposes the primary constraint. Specifically, we measure GPU power draw using the NVIDIA Management Library (NVML) and integrate it over the inference duration to obtain the total GPU energy consumption. We then compute the energy consumed per generated token and extrapolate it to the number of tokens that can be generated with 100 Wh of GPU energy.

B.3 Our Method Hyperparameters

Our method consists of three components: (1) LoadMatch, (2) drafter slimming, and (3) an advanced baseline. In the following, we describe each component in turn and explain its associated hyperparameters along with our default choices. First, (1) the LoadMatch method determines the candidate-token count once per target model, drafter, and device using 80 sentences from a calibration dataset; the count is then fixed for all decoding. For the calibration dataset, we use the widely adopted UltraChat-200K dataset (Ding et al., 2023). The determined number of tokens is used across all evaluation datasets. Although it is also possible to search for an optimal number per evaluation task, we omit this process for simplicity. Next, (2) drafter slimming combines three different compression approaches. Each approach has a tunable hyperparameter, but we stick to the base hyperparameters for simplicity. For quantization, we use the GPTQ (Frantar et al., 2023) method with the default configuration of W4A16g128. For `lm_head` pruning, we adopt the existing frequency-based structure pruning (Zhao et al., 2025b), which removes the embedding vectors of less frequently occurring words from `lm_head`; following the existing works (Zhao et al., 2025b; Li et al., 2025b), we collect the word-frequency statistics on the SlimPajama-627B dataset and prune 96K out of the 128K vocabulary entries. We also reduce the width of the draft tree structure, where the children at the next level are chosen from the current-level nodes with the b highest confidences; we reduce this branching factor b from its default value of ten down to eight to obtain further speedup. Lastly, since (3) the advanced baseline is an efficient implementation of existing speculative decoding, it does not require any hyperparameter search.

C Computing Resource and Software Stack Specification

We specify the computing resources and software stack used in our experiments, covering both hardware and software configurations. For hardware, we use a Xeon Platinum CPU with 104 threads running at a clock frequency of 2.7 GHz, together with widely used consumer-grade NVIDIA GPUs (*e.g.*, RTX 3090 with 24 GB VRAM and A5000 with 24 GB VRAM) to evaluate the proposed method on low-powered computing devices (*e.g.*, edge servers, desktops, or laptops). Among them, we adopt the A5000 as our default GPU because its compute capability, memory specification, and power consumption are comparable to those of the low-power GPUs equipped in recent AI laptops. For example, the RTX 5090 Laptop GPU delivers 33.5 TFLOPS of FP32 compute and 67.0 TFLOPS of FP16 tensor-core compute with 24 GB of VRAM and 896 GB/s memory bandwidth, while the A5000 offers a comparable profile of 27.8 TFLOPS FP32 and 55.5 TFLOPS FP16 with the same 24 GB of VRAM and 768 GB/s memory bandwidth. To match the power budget of laptop GPUs, we limit the A5000 to 210 W throughout our evaluation. We refer to this configuration as laptop-scale because its compute and memory characteristics, together with the constrained power budget, are comparable to those of recent high-end laptop GPUs. On the other hand, we employ Ubuntu 24.04 LTS as the operating system, `PyTorch==2.11.0+cu130` as the deep learning framework, and `vLLM==0.23.0` as the inference serving engine. We use a single GPU for training and evaluating the models, except for Qwen3.5-27B, which requires four GPUs for running the FP16 model and two GPUs for the INT4 model during the inference phase.

D Pseudo Code

Algorithm 1 summarizes the overall compression pipeline of our method, which consists of three stages: (i) *drafter slimming* (Slim, Algorithm 2), which compresses the target model, the draft model, and the draft `lm_head`; (ii) and (iii) a two-round *LoadMatch* (LM, Algorithms 3 and 4),

Algorithm 1 Pseudocode for our compression.

Require: target θ, draft ϕ, draft `lm_head` weights ω, calibration samples $\mathcal{X}$, word frequencies $\mathcal{F}$, latency function $\mathrm{L}(\cdot,\cdot)$, a large count γ_c, tile size q

$\theta_{\text{INT4}}, \phi_{\text{INT4}}, \hat{\omega}_{\text{INT4}} \leftarrow \textsc{Slim}(\theta, \phi, \omega, \mathcal{X}, \mathcal{F})$
$\hat{\gamma} \leftarrow \textsc{LMOne}(\theta_{\text{INT4}}, \gamma_c)$
$\gamma^* \leftarrow \textsc{LMTwo}(\theta_{\text{INT4}}, \phi_{\text{INT4}}, \hat{\omega}_{\text{INT4}}, \hat{\gamma}, q, \mathcal{X})$

Algorithm 2 Pseudocode for our draft slim.

function $\textsc{Slim}(\theta, \phi, \omega, \mathcal{X}, \mathcal{F})$
  $\theta_{\text{INT4}} \leftarrow \mathrm{GPTQ}(\mathcal{X}, \theta, \mathrm{bit}_w{=}4, \mathrm{bit}_a{=}16,$ group_size=128)
  $\phi_{\text{INT4}} \leftarrow \mathrm{GPTQ}(\mathcal{X}, \phi, \mathrm{bit}_w{=}4, \mathrm{bit}_a{=}16,$ group_size=128)
  $\hat{\omega} \leftarrow \mathrm{Pruning}(\omega, \mathrm{word_to_frequency}{=}\mathcal{F})$
  $\hat{\omega}_{\text{INT4}} \leftarrow \mathrm{GPTQ}(\mathcal{X}, \hat{\omega}, \mathrm{bit}_w{=}4, \mathrm{bit}_a{=}16,$ group_size=128)
  return $\theta_{\text{INT4}}, \phi_{\text{INT4}}, \hat{\omega}_{\text{INT4}}$

Algorithm 3 Pseudocode for our LM round-1.

function $\textsc{LMOne}(\theta_{\text{INT4}}, \gamma_c)$
  $t_{\text{mem}} \leftarrow \mathrm{L}(\theta_{\text{INT4}}, 1)$
  $t_{\text{com}} \leftarrow \mathrm{L}(\theta_{\text{INT4}}, \gamma_c)/\gamma_c$
  $\hat{\gamma} \leftarrow t_{\text{mem}}/t_{\text{com}}$ ▷ *load-match point*
  return $\hat{\gamma}$

Algorithm 4 Pseudocode for our LM round-2.

function $\textsc{LMTwo}(\theta_{\text{INT4}}, \phi_{\text{INT4}}, \hat{\omega}_{\text{INT4}}, \hat{\gamma}, q, \mathcal{X})$
  $k \leftarrow \lfloor \hat{\gamma}/q \rfloor$
  $\gamma_- \leftarrow \max\{1, qk\}$
  $\gamma_+ \leftarrow q(k+1)$ ▷ *tile boundaries*
  $s_- \leftarrow \mathrm{TpS}(\gamma_-; \mathcal{X}_{\text{LM}})$
  $s_+ \leftarrow \mathrm{TpS}(\gamma_+; \mathcal{X}_{\text{LM}})$
  if $s_- \geq s_+$ **then**
    $\gamma^* \leftarrow \gamma_-$
  else
    $\gamma^* \leftarrow \gamma_+$
  return γ^*

which determines the candidate-token count γ used for speculative decoding on the compressed models. Given the target weights θ, the draft weights ϕ, and the draft `lm_head` weights ω, the pipeline first produces their compressed counterparts θ_{INT4}, ϕ_{INT4}, and $\hat{\omega}_{\text{INT4}}$, then obtains the load-match point $\hat{\gamma}$ in LM round-1, and finally refines it to the optimal γ^* in LM round-2.

Drafter slimming (Algorithm 2). Our drafter slimming quantizes the target and draft model weights to 4-bit integers using GPTQ (Frantar et al., 2023), an advanced post-training quantization method, with 16-bit activations and a group size of 128, calibrated on 128 sequences of length 2048 randomly extracted from the UltraChat-200K (Ding et al., 2023) dataset. For the draft `lm_head`, we first prune 96K of its 128K rows based on word frequency statistics on the SlimPajama-627B dataset, keeping only the vocabulary entries that appear frequently in the calibration corpus, and then apply the same GPTQ quantization to the pruned weights. This yields a compact draft whose vocabulary projection cost is substantially reduced, as the `lm_head` accounts for a large fraction of the parameters and computation in small draft models, while the retained high-frequency vocabulary still covers the vast majority of tokens generated in practice.

Pruning the draft `lm_head` only limits which tokens the draft model can propose; it does not restrict the output of the target model. The target model, its `lm_head`, and the acceptance rule remain unchanged. Under greedy decoding, a draft token is accepted only when it matches the target model's argmax, and the target prediction is used whenever the draft is rejected or ends. Under sampling, speculative decoding applies the standard acceptance and correction rule (Leviathan et al., 2023). This rule remains valid even when the draft distribution covers only part of the vocabulary, because tokens that cannot be proposed by the draft are still sampled from the target distribution during correction. Therefore, pruning the draft vocabulary does not change the output distribution of the quantized target model, consistent with the matching accuracies of AR(INT4), SD(INT4), and LM(INT4) in Tables 4 and 10. The graph-compatible implementation in Section 2.2 only vectorizes this procedure and does not alter its sampling distribution.

LM round-1 (Algorithm 3). The parallelism of speculative decoding is determined by the candidate-token count γ verified in a single target forward pass. Round-1 estimates the load-match point from the actual verification latency of the deployed quantized target. We define $\mathrm{L}(\theta, \gamma)$ as the wall-clock latency when model θ processes γ tokens in parallel. To measure it, we run the INT4 target model on dummy inputs and average 30 repeated runs after 10 warm-up iterations. We measure a single token to represent the memory-dominated regime and a sufficiently large token count γ_c to represent the compute-dominated regime. Their measured latencies give

$$\hat{\gamma} = \frac{\mathrm{L}(\theta_{\text{INT4}}, 1)\,\gamma_c}{\mathrm{L}(\theta_{\text{INT4}}, \gamma_c)}. \tag{1}$$

This estimate indicates where the memory and compute costs of quantized verification become comparable. Round-2 further adjusts this point according to the tiled execution of the GEMM kernel.

Table 10: **Extended comprehensive evaluation.** Accuracy, model memory, decoding speedup, and acceptance length τ for Qwen3-4/8B under GPTQ and AWQ quantization, across science, math, and code tasks. Acc is pass@1 for HumanEval and MBPP and exact-match accuracy elsewhere. Mem denotes model weight memory footprint (GB). We use DFlash as SD method.

Model Config	Method	Mem	SCIENCE			MATH						CODE						OVERALL		
			GPQA			GSM8K			MATH500			HumanEval			MBPP			*Avg.*		
	Temp. = 0		Acc	Speedup	τ	Acc	Speedup	τ	Acc	Speedup	τ	Acc	Speedup	τ	Acc	Speedup	τ	Acc	Speedup	τ
Q3-4B GPTQ	FP16	7.56 GB	39.9	1.00×	1.00	86.0	1.00×	1.00	81.0	1.00×	1.00	83.5	1.00×	1.00	65.8	1.00×	1.00	71.2	1.00×	1.00
	INT4	2.50 GB	35.9	2.38×	1.00	80.3	2.38×	1.00	77.6	2.38×	1.00	78.7	2.38×	1.00	61.4	2.38×	1.00	66.8	2.38×	1.00
	DFlash	3.56 GB	35.9	2.79×	5.83	80.3	2.67×	5.93	77.6	3.43×	7.39	78.7	2.74×	6.05	61.4	2.50×	5.49	66.8	2.83×	6.14
	+LM	2.82 GB	35.9	8.34×	5.61	80.3	8.94×	5.92	77.6	10.89×	7.26	78.7	8.87×	5.91	61.4	8.39×	5.32	66.8	9.09×	6.00
Q3-4B AWQ	INT4	2.50 GB	32.8	2.35×	1.00	80.3	2.39×	1.00	80.0	2.38×	1.00	76.2	2.34×	1.00	61.2	2.39×	1.00	66.1	2.37×	1.00
	DFlash	3.61 GB	32.8	2.19×	4.98	80.3	2.55×	5.72	80.0	3.46×	7.37	76.2	2.45×	5.78	61.2	2.37×	5.27	66.1	2.60×	5.82
	+LM	2.84 GB	32.8	7.16×	4.88	80.3	8.93×	5.69	80.0	11.04×	7.17	76.2	8.48×	5.64	61.2	8.10×	5.11	66.1	8.74×	5.70
Q3-8B GPTQ	FP16	15.27 GB	46.0	1.00×	1.00	87.3	1.00×	1.00	79.6	1.00×	1.00	84.2	1.00×	1.00	66.4	1.00×	1.00	72.7	1.00×	1.00
	INT4	5.68 GB	39.4	2.64×	1.00	87.2	2.66×	1.00	81.8	2.69×	1.00	84.2	2.68×	1.00	66.4	2.68×	1.00	71.8	2.67×	1.00
	DFlash	7.71 GB	39.4	3.42×	5.43	87.2	4.80×	5.98	81.8	6.32×	7.62	84.2	4.99×	6.15	66.4	4.55×	5.69	71.8	4.82×	6.17
	+LM	6.27 GB	39.4	9.46×	5.47	87.2	10.39×	6.02	81.8	12.97×	7.43	84.2	10.81×	6.25	66.4	10.19×	5.70	71.8	10.76×	6.17
Q3-8B AWQ	INT4	5.71 GB	38.4	2.63×	1.00	86.3	2.67×	1.00	75.8	2.64×	1.00	86.6	2.67×	1.00	66.0	2.65×	1.00	70.6	2.65×	1.00
	DFlash	7.77 GB	38.4	3.80×	4.88	86.3	4.68×	5.81	75.8	5.82×	7.37	86.6	4.89×	6.29	66.0	4.31×	5.40	70.6	4.70×	5.95
	+LM	6.31 GB	38.4	8.38×	4.86	86.3	10.33×	5.74	75.8	12.82×	7.33	86.6	10.96×	6.21	66.0	10.16×	5.55	70.6	10.53×	5.94

Table 11: **Extended concurrency evaluation.** Decoding throughput (TpS, tokens/s) and speedup over FP16 autoregressive decoding at the same model and batch size.

DOMAIN	Task	Method	Qwen3-4B								Qwen3-8B							
			BS = 1		BS = 2		BS = 4		BS = 8		BS = 1		BS = 2		BS = 4		BS = 8	
	Temp. = 0		TpS	Speedup	TpS	Speedup	TpS	Speedup	TpS	Speedup	TpS	Speedup	TpS	Speedup	TpS	Speedup	TpS	Speedup
SCI	GPQA	FP16	70	1.00×	139	1.00×	265	1.00×	472	1.00×	41	1.00×	78	1.00×	151	1.00×	268	1.00×
		INT4	167	2.38×	314	2.26×	554	2.09×	949	2.01×	108	2.64×	203	2.62×	381	2.52×	648	2.42×
		DFlash	195	2.79×	307	2.21×	615	2.32×	1034	2.19×	140	3.42×	321	4.14×	523	3.46×	648	2.42×
		+LM	584	8.34×	865	6.22×	1108	4.18×	1312	2.78×	388	9.46×	559	7.21×	781	5.17×	846	3.16×
MATH	GSM8K	FP16	73	1.00×	144	1.00×	276	1.00×	529	1.00×	41	1.00×	78	1.00×	151	1.00×	268	1.00×
		INT4	171	2.38×	339	2.35×	616	2.23×	1138	2.15×	108	2.66×	206	2.64×	359	2.38×	724	2.70×
		DFlash	194	2.67×	362	2.51×	675	2.44×	1222	2.31×	196	4.80×	367	4.70×	635	4.21×	828	3.09×
		+LM	650	8.94×	1062	7.36×	1363	4.93×	1630	3.08×	425	10.39×	718	9.19×	968	6.42×	1120	4.18×
	MATH500	FP16	71	1.00×	143	1.00×	263	1.00×	450	1.00×	40	1.00×	78	1.00×	146	1.00×	245	1.00×
		INT4	169	2.38×	327	2.29×	586	2.23×	936	2.08×	108	2.69×	207	2.65×	381	2.61×	651	2.66×
		DFlash	245	3.43×	385	2.69×	828	3.15×	1363	3.03×	256	6.32×	434	5.54×	773	5.29×	979	4.00×
		+LM	778	10.89×	1344	9.40×	1676	6.38×	1975	4.39×	525	12.97×	869	11.10×	1088	7.45×	1267	5.18×
CODE	HumanEval	FP16	72	1.00×	143	1.00×	274	1.00×	502	1.00×	41	1.00×	79	1.00×	153	1.00×	290	1.00×
		INT4	171	2.38×	335	2.34×	618	2.25×	1113	2.22×	109	2.68×	213	2.70×	401	2.62×	748	2.58×
		DFlash	199	2.74×	365	2.55×	716	2.61×	1289	2.57×	203	4.99×	380	4.82×	678	4.43×	875	3.02×
		+LM	642	8.87×	1054	7.37×	1403	5.11×	1685	3.36×	440	10.81×	697	8.85×	909	5.94×	1145	3.95×
	MBPP	FP16	72	1.00×	145	1.00×	281	1.00×	527	1.00×	41	1.00×	80	1.00×	156	1.00×	294	1.00×
		INT4	171	2.38×	336	2.31×	644	2.29×	1180	2.24×	109	2.68×	216	2.71×	414	2.65×	737	2.51×
		DFlash	181	2.50×	342	2.35×	644	2.29×	1217	2.31×	186	4.55×	347	4.35×	634	4.06×	834	2.84×
		+LM	607	8.39×	1002	6.89×	1369	4.87×	1565	2.97×	416	10.19×	685	8.58×	912	5.84×	1068	3.64×

LM round-2 (Algorithm 4). The estimated load-match point can lie inside a tiled execution region, where neighboring token counts may exhibit similar verification latency. Round-2 therefore considers two candidate points around $\hat{\gamma}$. Let q denote the tile size along the token dimension. We define the lower candidate as $\gamma_- = \max\{1, q\lfloor \hat{\gamma}/q \rfloor\}$ and the upper candidate independently as $\gamma_+ = q(\lfloor \hat{\gamma}/q \rfloor + 1)$. This definition ensures that the upper candidate remains aligned with the next tile boundary even when $\hat{\gamma} < q$. For example, when $\hat{\gamma} = 27$ and $q = 16$, the two candidates are 16 and 32; when $\hat{\gamma} = 8$ and $q = 16$, they are 1 and 16. The two candidates provide different trade-offs between verification latency and acceptance length. We evaluate their end-to-end throughput on 80 samples from the calibration set using the compressed target and draft models. The measurement includes both drafting and target verification, so the resulting TpS reflects the actual decoding cost and acceptance length. We select the candidate with higher throughput as γ^*. The selected candidate-token count is fixed and reused across the evaluation tasks.

Table 12: **Statistic analysis.** We report the mean and standard deviation of speedup and acceptance length over three runs with seeds 44, 46, and 48. Speedup is measured relative to FP16 autoregressive decoding at temperature 1 across math, code, chat, and general NLP tasks.

Model	Method	GSM8K	HumanEval	MT-Bench	Alpaca	Summary	QA	*Avg.*
(a) Speedup over FP16 autoregressive decoding								
L3-8B	SD(FP16)	2.30 ±0.25×	2.96 ±0.04×	1.79 ±0.13×	2.62 ±0.19×	2.30 ±0.02×	1.39 ±0.16×	2.23×
	AR(INT4)	2.79 ±0.01×	2.79 ±0.02×	2.77 ±0.00×	2.79 ±0.01×	2.76 ±0.11×	2.79 ±0.01×	2.78×
	SD(INT4)	2.04 ±0.17×	3.23 ±0.21×	1.81 ±0.18×	2.90 ±0.04×	2.87 ±0.05×	1.63 ±0.27×	2.42×
	LM(INT4)	5.10 ±0.70×	7.53 ±0.15×	4.59 ±0.20×	6.72 ±0.39×	5.91 ±0.02×	3.77 ±0.34×	5.60×
DS-8B	SD(FP16)	3.01 ±0.14×	2.80 ±0.03×	2.35 ±0.02×	2.44 ±0.07×	1.89 ±0.03×	2.15 ±0.03×	2.44×
	AR(INT4)	2.79 ±0.00×	2.78 ±0.00×	2.79 ±0.00×	2.79 ±0.00×	2.79 ±0.00×	2.79 ±0.00×	2.79×
	SD(INT4)	3.04 ±0.02×	3.33 ±0.01×	2.69 ±0.04×	2.65 ±0.04×	2.02 ±0.00×	2.21 ±0.03×	2.66×
	LM(INT4)	7.95 ±0.15×	6.80 ±0.02×	5.54 ±0.09×	5.40 ±0.09×	4.35 ±0.08×	5.25 ±0.07×	5.88×
(b) Acceptance length τ								
L3-8B	SD(FP16)	5.21 ±0.05	5.46 ±0.04	4.26 ±0.08	5.57 ±0.09	4.34 ±0.04	3.94 ±0.15	4.80
	AR(INT4)	1.00 ±0.00	1.00 ±0.00	1.00 ±0.00	1.00 ±0.00	1.00 ±0.00	1.00 ±0.00	1.00
	SD(INT4)	5.05 ±0.08	5.24 ±0.07	4.19 ±0.20	5.56 ±0.04	4.43 ±0.04	3.81 ±0.07	4.71
	LM(INT4)	4.32 ±0.08	4.76 ±0.05	3.41 ±0.14	4.56 ±0.07	3.64 ±0.01	3.15 ±0.04	3.97
DS-8B	SD(FP16)	5.68 ±0.04	5.07 ±0.03	4.41 ±0.05	4.55 ±0.05	3.79 ±0.03	4.24 ±0.01	4.62
	AR(INT4)	1.00 ±0.00	1.00 ±0.00	1.00 ±0.00	1.00 ±0.00	1.00 ±0.00	1.00 ±0.00	1.00
	SD(INT4)	5.59 ±0.02	5.00 ±0.02	4.20 ±0.03	4.29 ±0.07	3.36 ±0.00	4.12 ±0.11	4.43
	LM(INT4)	4.92 ±0.04	4.07 ±0.01	3.47 ±0.03	3.51 ±0.07	2.90 ±0.04	3.35 ±0.08	3.70

E Extended Experimental Results

In Tables 4 and 5 and Figure 7 of the manuscript, due to space constraints, we evaluated the proposed method only on EAGLE-3 (Li et al., 2025b), a recent autoregressive drafter head, with the Llama3-8B and DeepSeek-R1-distill-Llama-8B models. To address this limitation, we have conducted additional experiments that extend our comprehensive evaluation of Table 10 and concurrency analysis of Table 11 in two directions. First, we apply our method to DFlash (Chen et al., 2026), an advanced speculative decoding approach that adopts a diffusion-based drafter head instead of an autoregressive one. Second, we evaluate our method on more recent LLMs, such as Qwen3-4B/8B. These additional experiments demonstrate that the proposed method is not tied to a specific drafter architecture or model family, but generalizes well across both. The results confirm that our method consistently performs well across a variety of metrics, including inference speed, accuracy, and memory consumption, as well as across diverse deployment scenarios, such as serving concurrent queries.

Extended comprehensive evaluation. We present the extended comprehensive evaluation on Qwen3-4B/8B with DFlash under GPTQ and AWQ quantization in Table 10. Across science, math, and code tasks, our method generally improves the throughput speedup over the DFlash baseline while preserving the accuracy of the quantized models. The acceptance length remains comparable to that of the baseline, which suggests that the speed gain does not come at the cost of draft quality. In addition, the memory footprint tends to stay close to that of the quantized-only models, indicating that the proposed method scales reasonably well to a diffusion-based drafter head and more recent LLMs.

Extended concurrency evaluation. We summarize the extended concurrency evaluation across varying batch sizes in Table 11. Our method sustains higher decoding throughput than the baselines as the batch size grows, although the relative speedup gradually narrows, which is a commonly observed tendency in speculative decoding under batched inference. Nevertheless, the throughput advantage remains meaningful even at larger batch sizes, suggesting that the proposed method serves concurrent queries reasonably well in practical deployment scenarios.

F Quantized GEMM Analysis

It has long been challenging to accelerate the quantized GEMM over the FP16 baseline across both memory- and compute-bound scenarios (Xiao et al., 2023). To attain this speedup, we need to represent both weights and activations in low precision (*e.g.*, 4 bits), enabling matrix multiplication using fast INT4 Tensor Core operations that accumulate results in an INT32 container to provide about $4\times$ speedup over FP16 precision. With such high arithmetic throughput, INT4 GEMM could ideally

deliver a consistent $4\times$ speedup across a wide range of workloads. In practice, however, realizing this speedup is difficult due to dequantization overhead. Prior works (Shao et al., 2024; Liu et al., 2024) have shown that low-bit weight-activation quantization (`W4A4`) suffers from significant accuracy degradation compared with the FP16 baseline. To mitigate this problem, recent work (Zhao et al., 2024) introduces per-group quantization, which stores fine-grained scales and zero points to reduce reconstruction error. However, such fine-grained quantization increases the dequantization overhead during matrix multiplication. Computation within each group is performed in INT4, while the partial results across groups must be rescaled and accumulated in FP16 using the corresponding group-wise quantization parameters. To reduce this dequantization overhead, several studies (Lin et al., 2025; Zhang et al., 2024; Guo et al., 2026; Liu et al., 2025) have developed efficient GEMM kernels that decompose the dequantization parameters into group-wise integer and channel-wise floating-point components, enabling matrix multiplication on fast INT8 Tensor Cores while applying only lightweight channel-wise scaling afterward. Nevertheless, these approaches still provide limited benefits in compute-bound regimes, leaving consistent speedup across both memory- and compute-bound workloads an open challenge.

G Statistical Analysis

To assess the run-to-run variation of our throughput measurements, we repeat the experiments three times with different random seeds (*i.e.*, 44, 46, and 48) and report the mean and standard deviation in Table 12. We use a temperature of one so that each seed produces different generation trajectories, while keeping all other model, decoding, and hardware configurations fixed. As shown in Table 12, both speedup and acceptance length exhibit small variations across runs, and the relative improvements of our method remain consistent across models and tasks.