MATGPTQ: EFFICIENT AND ACCURATE INFERENCE OVER NESTED QUANTIZED MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Matryoshka Quantization (MatQuant), Any-Precision-LLM (AP) and AnyBCQ (AB) are recent quantization approaches showing that a single integer-quantized model can be served across multiple precisions. In this paradigm, lower-precision models are extracted from a higher-precision model by simply reading fewer bits of the weights. This enables a single checkpoint to cover a wide range of memory and latency budgets, but makes both quantization and efficient execution substantially harder. Existing methods rely on expensive quantization-aware training (QAT) or gradient-based post-training quantization (PTQ) rather than fast one-shot PTQ, and offer limited system support: dedicated kernels are either missing or restricted to single- or small-batch decoding. We address these limitations with *Post-Training Matryoshka Quantization* (MatGPTQ), an end-to-end pipeline for nested-model quantization and inference. MatGPTQ casts Matryoshka quantization as multi-precision error compensation, producing a single "sliceable" parent model jointly optimized for multiple target precisions in one pass over a small calibration set. We further refine the MatQuant representation so that an r-bit model reads exactly r bits, and introduce the first dedicated inference kernels for this format, supporting batch sizes beyond one and integrated into vLLM. Across standard LLMs and benchmarks, MatGPTQ outperforms MatQuant while remaining competitive with AP and AB at the smallest checkpoint size, and our kernels achieve end-to-end speedups of up to 3.5$\times$ over BF16 at the low-bit regime. Overall, MatGPTQ makes nested quantized models practical to serve from a single, compact checkpoint.

1 INTRODUCTION

Quantization is currently the most practical way to shrink the memory and bandwidth costs of large language models (LLMs): by mapping full-precision weights to low-bit representations, one can enable single-GPU or edge deployment, while mostly retaining model accuracy. One-shot, post-training quantization (PTQ) is attractive for its low cost and strong accuracy in the 3–4 bit regime (Frantar et al., 2023; Kurtic et al., 2025), whereas optimization-based methods such as QAT and block-wise, gradient-calibrated PTQ (e.g., OmniQuant (Shao et al., 2024)) achieve higher accuracy in the low-bit regime (i.e: 2–3 bits), but at the expense of extra computational cost.

Recently, *Matryoshka Quantization* (MatQuant) (Nair et al., 2025), *Any-Precision-LLM* (AP) (Park et al., 2024) and *AnyBCQ* (AB) (Park et al., 2026) provided a fresh approach to model compression by turning integer bit representations into a *nested representation*: a single checkpoint supports several inference precisions. Even though MatQuant, AP and AB pursue this goal, they achieve it differently. MatQuant jointly optimizes a high-precision parent representation whose prefixes define lower-precision models, whereas AP and AB construct the hierarchy progressively from a low-precision model. Therefore, these approaches trade off quantization cost, representation complexity and hardware execution in different ways.

Despite their attractive storage flexibility, existing nested representations remain difficult to deploy efficiently. MatQuant relies on a single shared-scale integer representation, but is reliant on gradient-based methods such as QAT and OmniQuant. Furthermore, MatQuant's lower-precision models are not *true-to-size*, since extracting an r-bit model requires reading $r + 1$ bits from the parent representation. We refer to this operation as *slicing* throughout the paper. The original work also did

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.

not provide open-source code or dedicated kernels for the format, making efficient deployment difficult. On the other hand, AP and AB kernels have limitations: AP kernels only support small batch sizes ≤ 16 during decoding, while AB kernels are only available for single-batch decoding. For prefill and higher-batch decoding, both methods fall back to dequantization followed by cuBLAS. In addition, both methods rely on more complex and overall more memory-heavy representations, with AP requiring bit-width-specific centroid lookup tables and AB requiring individual scales for each supported bit-width. Consequently, there remains a gap between storing many precision levels efficiently and executing them efficiently across realistic workloads.

Contributions. We revisit the MatQuant representation from a deployment perspective and ask whether its nested, shared-scale integer structure can retain the accuracy of recent multi-precision approaches while admitting efficient end-to-end execution across prefill, decoding, and batch sizes beyond one. We answer this with *MatGPTQ*, an end-to-end framework that combines one-shot multi-precision PTQ with a deployment-friendly nested representation and dedicated GPU kernels. In summary, our contributions are:

1. **Comparative view of Nested Quantization**. We study MatQuant, AP, and AB as alternative realizations of single-checkpoint multi-precision inference and characterize their trade-offs in construction, representation flexibility and hardware execution.
2. **Multi-precision One-Shot PTQ**. We adapt GPTQ to jointly optimize a single parent model across multiple target precisions using cross-bit error compensation.
3. **Deployment-friendly representation and execution.** We introduce true-prefix slicing, improving over *MatQuant*, while keeping the single shared scale format. Furthermore, we provide GPU kernels for nested inference at 2, 3, 4, 6 and 8 bits, supporting prefill, decoding, and larger batch sizes, together with efficient packing and vLLM integration (Kwon et al., 2023).
4. **Empirical Validation**. We evaluate MatGPTQ on Llama3.1-8B-Instruct and Qwen3-8B against independently quantized GPTQ, MatQuant, AP, and AB. MatGPTQ substantially improves over MatQuant, remains competitive at every precision with AP and AB, while using the least complex representation.

2 Background & Related Work

2.1 Post-training Quantization

Quantization is often applied as a *post-training (PTQ)* step, in one-shot to already trained models. We focus on such one-shot PTQ methods in this paper, by contrast to more expensive quantization-aware training (QAT). For PTQ, one can differentiate between *data-free* approaches, which use simpler round-to-nearest (RTN) projections (Dettmers, 2016; Badri & Shaji, 2023), but tend to drop accuracy even at 4-bits, and *data-aware* approaches such as GPTQ (Frantar et al., 2023) or AWQ (Lin et al., 2024), which utilize a calibration set that is passed through the model during quantization. Specifically, a common approach in PTQ is to quantize each linear layer ℓ with weights W_ℓ, in linear order, to produce the quantized weights Q_ℓ. Please see Section 3 for an overview of the MatGPTQ algorithm.

2.2 Nested multi-precision quantization

We call models nested when higher bit-width models "contain" the lower-bit-width models, within a single checkpoint.

Matryoshka Quantization (MatQuant) (Nair et al., 2025) is a shared-scale framework for producing nested models: from a c-bit master code q, an r-bit model ($0 < r \leq c$) is obtained by *slicing*,

$$S(q, r) = \mathrm{clamp}\Big(\mathrm{round}\Big(\frac{q}{2^{c-r}}\Big),\ 0,\ 2^r - 1\Big) \cdot 2^{c-r}. \tag{1}$$

Here, the right shift keeps the r most significant bits, while the left shift maps the value back into the c-bit code space, allowing the same scale and zero-point to be reused while approximating the original c-bit value. Truncation, however, introduces a systematic underestimation and therefore

loses information. To reduce this effect, MatQuant uses the $(r+1)$-th bit for rounding, which can overflow the valid r-bit range and therefore requires an additional clamp operation. Consequently, extracting an r-bit model requires reading $r+1$ bits per weight together with additional rounding and clamping operations, complicating efficient deployment. See Appendix A.1 for more details.

Nair et al. (2025) then define their objective function, which minimizes over θ, representing the set of model and auxiliary parameters being optimized. The loss function $\mathcal{L}(\cdot)$ is evaluated for a fixed target set of bit-widths $R = \{r_1, r_2, ..., r_K\}$, under the quantization method $Q(\cdot)$ and forward pass $F(\cdot)$, and is calculated as:

$$\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} \sum_{r \in R} \lambda_r \, \mathcal{L}(F(S(Q(\theta, c), r)); \mathcal{D}_i) \,, \tag{2}$$

Here, $\mathcal{D}_i$ represents either the input-label pair used for QAT or the layer inputs and full-precision outputs used for OmniQuant. The parameter λ_r is used to weight the importance of individual bit-widths, allowing the optimization to emphasize different bit-widths accordingly. Overall, the MatQuant representation follows the default quantization representation, which is a combination of quantized weights, scales and zero-points that are grouped for better performance.

Relatedly, TruncQuant (Kim et al., 2025) enables variable-precision inference through truncation of low-order bits, but relies on truncation-aware training rather than one-shot PTQ and does not target the nested LLM deployment setting considered here.

Any-Precision-LLM (AP) (Park et al., 2024) introduces an *up-scaling* method based on SqueezeLLM's (Kim et al., 2024) clustered, non-uniform quantization. AP starts by constructing the lowest-bit *seed* model using the corresponding centroid-based representation. Afterward, additional bits and therefore centroids are progressively added to increase the number of representable centroids and construct higher-bit-width models. Generally, the centroid values are stored within lookup tables, while the quantized weights store indices pointing to the respective centroid values.

The corresponding quantized weights are stored using bit-planes, with each additional bit represented by a separate plane. At inference time, an r-bit model is obtained by loading the first r bit-planes and combining them to recover the corresponding centroid indices, which are then used to look up the respective centroid values. This introduces additional representation complexity, as AP relies on centroid lookup tables, which also need to be stored for each supported bit-width and layer, to reconstruct the quantized weights. They choose this approach since their up-scaling method cannot be directly applied to GPTQ or AWQ.

AnyBCQ (AB) (Park et al., 2026) builds on Binary-Coded Quantization (BCQ). AB starts by constructing the lowest-bit *base* model using BCQ. Afterward, higher bit-widths are progressively constructed by freezing the previously obtained binary bases and adding a new binary basis together with its corresponding scale. The newly added basis is initialized from the residual error of the lower-bit model and subsequently refined, allowing each higher bit-width to reuse all previously constructed binary bases.

The corresponding binary bases are stored as individual bit-planes. More precisely, for a bit-width r, the quantized weight matrix $\widehat{\mathbf{W}}_r$ is represented as a linear combination of r binary bases and their corresponding real-valued scales, $\widehat{\mathbf{W}}_r = \sum_{i=1}^{r} \alpha_i \mathbf{B}_i$, where each $\mathbf{B}_i$ is a binary matrix with $\{-1, 1\}$ entries and $\alpha_i \in \mathbb{R}$ represents its corresponding scale. At inference time, an r-bit model is obtained by loading the first r binary bit-planes together with their respective scales and combining them to reconstruct the quantized weights. Unlike AP, this representation does not require centroid lookup tables and instead allows computation directly on the binary bit-planes. However, each additional bit introduces an additional scale and potentially an additional zero-point for each group, resulting in multiple quantization parameters being required for the nested representation.

Further Related Work. D^2MoE (Wang et al., 2025) also introduces a nested multi-precision representation, but specifically targets Mixture-of-Experts (MoE) layers and expert-wise execution. NestQuant (Xie et al., 2025) also studies nested integer representations, but focuses on on-device DNN deployment and constructs nested models through a high-/low-bit decomposition. Therefore, we do not include these methods in our comparison.

Table 1: Comparison of multi-precision quantization methods.

Property	**MatQuant**	**AP**	**AB**	**MatGPTQ**
Uses gradients during quantization	✓	∼	✓	×
One-shot multi-precision PTQ	×	×	×	✓
Single multi-precision checkpoint	✓	✓	✓	✓
Attention+FFN quantization	∼	✓	✓	✓
Exact r-bit loading	×	✓	✓	✓
Inference kernel	×	∼	∼	✓
Open-source implementation	×	✓	✓	✓
Nested checkpoint size (Llama-3.1-8B)	8 GB	10 GB	13 GB	8 GB
Quantization time (Llama-3.1-8B, A6000)	$\sim 7\,h$	$\sim 35\,min$	$\sim 18\,h$	$\sim 35\,min$

2.3 Comparison of nested representations

Next, we compare key characteristics of MatQuant, AP, and AB, as summarized in Table 1. First, all three methods rely on gradient information, which increases the computational cost and complexity of quantization and motivates the question of whether comparable performance can be achieved without gradients. However, they differ in when gradients are required: AP uses gradients only prior to quantization, whereas MatQuant and AB rely on them during the quantization process itself. Second, all methods provide a single checkpoint. However, MatQuant was only evaluated using FFN quantization for its PTQ experiments. Furthermore, MatQuant's representation does not allow exact r-bit loading, as an additional bit, for rounding, needs to be read. MatQuant does not provide any open-source implementation or kernel support, making adaptation and usage difficult. AP and AB, on the other hand, quantize Attention+FFN while providing exact r-bit loading and their implementations are open-sourced, but they only provide limited kernel support. Specifically, AP only provides GEMM support for low batch decoding scenarios, while AB provides SIMT kernels for batch-size-1 decoding. Both rely on dequantization followed by cuBLAS for prefill and larger batch sizes. This represents a *limitation* and leaves this as an open problem. Lastly, the complexity of the different representations can be observed in the storage overhead for the same model, as seen in Table 1. Here, MatQuant's simple format results in the smallest model-size, while especially AB requires substantially more storage, making deployment on memory-constrained devices more difficult.

In this context, our work presents the first post-training quantization (PTQ) and open-source variant of an improved MatQuant representation, enabling nested quantization of both Attention and FFN layers from a single checkpoint. We evaluate accuracy at 3, 4, 6 and 8 bits, while our kernels additionally support 2-bit execution. We introduce exact r-bit loading from a single checkpoint while retaining MatQuant's compact shared-scale representation. MatGPTQ achieves performance that is on par with independently quantized, non-nested PTQ models across the supported bit-widths. Furthermore, in direct comparison, it substantially improves over MatQuant's OmniQuant implementation and is competitive with or exceeds AP and AB at every precision. Finally, we provide dedicated CUDA kernels that enable efficient nested inference for both prefill and decoding, including batch sizes larger than one, together with the corresponding memory savings.

3 Matryoshka GPTQ

Since the original MatQuant construction is gradient-based and its slicing operation does not allow exact r-bit loading, we address both limitations from a PTQ and deployment perspective. We first simplify the representation for efficient system execution, and then adapt GPTQ to enable one-shot, full-model multi-precision quantization.

3.1 Improved Representation

Recall that the original slicing operator $S(\cdot, r)$ from Equation 1 extracts the r most significant bits using a right shift, followed by a left shift to push values back into the c-bit regime. However, it

additionally requires reading the $(r+1)$-th bit for rounding, and a clamp operation to ensure that the rounded value remains within the valid r-bit range, which increases operation and memory complexity. To overcome these limitations, we introduce an exact r-bit slicing operation:

$$S'(q,r) = \left\lfloor \frac{q}{2^{c-r}} \right\rfloor \cdot 2^{c-r}, \quad q \in \{0,\dots,2^c-1\}, \tag{3}$$

which simply discards the lowest $d = c - r$ bits by performing only the originally intended right followed by left shift. Each slice is therefore a plain bit prefix of the master code: no additional bit is read, no carry can occur, and no clamping is required. Since truncation maps all codes of a cell to its lowest value, we recenter by $\delta_r = (2^d - 1)/2$, which minimizes the root-mean-squared error for uniformly distributed codes. This matches the error of MatQuant's rounding while retaining exact r-bit loading. Additionally, at inference, δ_r is folded into the zero-point. See Appendix A.1 for details.

3.2 The MatGPTQ Algorithm

For each layer ℓ with weights W_ℓ, we optimize a set of target bit-widths $R = \{r_1,\dots,r_K\} \subseteq \{2,\dots,c\}$, where $c \in \mathbb{N}$ is the highest *master* bit-width. This master representation is used as the basis for the shared scale and zero-point, while the lower bit-widths are derived from it. Recall that GPTQ solves the data-aware layer-wise quantization problem in Equation 4 w.r.t. Q_ℓ, for fixed W_ℓ, calibration data X_ℓ, and group-wise scale s_ℓ and zero-point z_ℓ:

$$\arg\min_{Q_\ell} \|\widehat{W}_\ell X_\ell - W_\ell X_\ell\|_2^2, \quad \widehat{W}_\ell = s_\ell\,(Q_\ell - z_\ell). \tag{4}$$

We now adapt this layer-wise GPTQ objective to the MatQuant formulation of Equation 2. Given W_ℓ and X_ℓ, GPTQ leverages the layer-wise Hessian $H_\ell = 2X_\ell X_\ell^\top$ to quantize weights, for a single precision, in a fixed order while compensating for the error through updates to the remaining unquantized weights. Instead, we replace this single-precision error with the λ-weighted error across all target bit-widths, so that one set of master codes Q_ℓ^c is chosen for every precision at once:

$$\arg\min_{Q_\ell^c,\, s_\ell,\, z_\ell} \sum_{r\in R} \lambda_r \left\|\widehat{W}_{\ell,r}\, X_\ell - W_\ell X_\ell\right\|_2^2, \quad \widehat{W}_{\ell,r} = s_\ell\left(S'(Q_\ell^c, r) + \delta_r - z_\ell\right), \tag{5}$$

where S' is the exact slicing operator of Equation 3 and δ_r its truncation offset.

From GPTQ to MatGPTQ. This formulation changes GPTQ from optimizing a single target precision to jointly selecting one master representation across all $r \in R$. Realizing this objective requires three corresponding modifications to the original GPTQ algorithm:

1. The group-wise scale and zero-point are no longer fitted against the master bit-width alone: we approximate MSE-optimal values via a grid search that evaluates each candidate pair (s_ℓ, z_ℓ) against every $r \in R$. The algorithm can be found in Appendix A.2.
2. The master codes Q_ℓ^c are not obtained by rounding to the nearest value on the quantization grid, since the nearest c-bit code may have poor lower-bit prefixes. Instead, each code q for a weight w is selected by minimizing the cross-bit-width weighted error of Equation 5 over its prefixes:

$$q^\star = \arg\min_{q\in\{0,\dots,2^c-1\}} \sum_{r\in R} \lambda_r \left(s_\ell\,(S'(q,r) + \delta_r - z_\ell) - w\right)^2 \tag{6}$$

   The algorithm can be found in Appendix A.2, together with its memory and runtime cost.
3. Error propagation must account for all bit-widths at once. Instead of propagating the single error that one bit-width would produce, as in native GPTQ, we propagate the λ-weighted average of the errors produced by all target bit-widths. Since all bit-widths share the remaining weights and the same Hessian, this is the OBS update minimizing Equation 5 over the remaining weights (for normalized λ_r).

Floating-Point. Extending the MatQuant framework to floating-point representations remains challenging: the discarded low-order bits then belong to the mantissa or exponent and contribute to the value multiplicatively rather than additively (Nair et al., 2025), so truncation no longer introduces a constant offset δ_r that can be folded into the zero-point. We leave this as a future direction.

4 FAST MATRYOSHKA KERNELS

We implement a CUDA kernel that serves the nested bit-widths $r \in \{2, 3, 4, 6, 8\}$ from a single packed model, dequantizing the weights on the fly. It targets low-batch latency, uses only `mma.sync`, `cp.async` and `ldmatrix`, and therefore runs on Ampere, Ada and Hopper. It could be extended for Hopper-specific features, similarly to how Marlin (Frantar et al., 2025) was extended to Machete (Wilkinson, 2024) for Hopper. Given dequantized weights W_D and FP16 or BF16 inputs $\mathbf{X} \in \mathbb{R}^{b \times d_{\text{in}}}$, it computes $\mathbf{Y} = \mathbf{X}W_D$, accumulating in FP32.

The implementation has three main components: First, we utilize the `mma.sync.m16n8k16` tensor-core instruction for computation. Since the batch b is small, we compute $\mathbf{Y} = (W_D^\top \mathbf{X}^\top)^\top$ internally, so that the weights fill the 16-row operand and the tokens the 8-column operand, while the layer keeps the `torch.nn.Linear` interface.

Second, we implement a batched conversion mechanism from quantized weights into their 16-bit form by assembling the weights into groups of 4 bits using the conversion techniques presented in Kim et al. (2022). In the 4-bit regime, we can directly apply their 4-bit conversion intrinsics. However, to support batched conversion for all five bit-widths at no additional storage cost, we split the codes into five bit-plane buffers holding bits 1–2, 3, 4, 5–6 and 7–8, counting from the most significant bit. Serving r bits therefore reads exactly r bits per weight, and one stored model can switch bit-width at run time. Offline, the weights are permuted into the tensor-core register layout. This layout supports two deployment modes: loading only the first r bits to serve an exact r-bit model with the memory footprint of a dedicated r-bit model, or keeping the full c-bit model resident and changing the bit-width dynamically during decoding, as suggested by Chen et al. (2025).

Third, we use two kernels depending on the batch size. Since inference is memory-bandwidth bound for $b \leq 4$, we use a direct-load kernel for streaming weights directly from memory while reading activations from L2, avoiding shared-memory staging and synchronization. For larger b, we use a batched kernel that reuses each dequantized weight fragment across $T \in \{8, 16, 32, 64\}$ tokens, stages activations with double-buffered `cp.async`, and selects tile size and reduction split per layer shape, batch size, and bit-width via the CUDA occupancy API.

5 EXPERIMENTS

5.1 GENERAL SETUP

Models & Data. We evaluate our method on the instruct-tuned versions of Llama3.1 8B (Dubey et al., 2024) and Qwen3 8B (Yang et al., 2025). To establish a point of comparison, we compare against independently quantized GPTQ, MatQuant, Any-Precision-LLM (AP), and AnyBCQ (AB) models. Details about the hyperparameters in Appendix B.

Evaluation Datasets. Following established methodology (Frantar et al., 2023), we evaluate the models on commonsense reasoning on the following zero-shot tasks: ARC (ARC-c, ARC-e) (Clark et al., 2018), HellaSwag (Zellers et al., 2019), PIQA (Bisk et al., 2020) and Winogrande (Sakaguchi et al., 2021). Additionally, we evaluate generation quality on the non-chain-of-thought versions of GSM8K (Cobbe et al., 2021), MMLU (Hendrycks et al., 2021), and IFEval (Zhou et al., 2023).

Implementation Details. Since delivering an open-source version was a major goal, we built our code on top of the widely used PyTorch-based Hugging Face framework (Wolf et al., 2020) and integrated the serving infrastructure into vLLM (Kwon et al., 2023). All experiments were performed on an NVIDIA RTX A6000 (Ampere) using CUDA 12.4.

5.2 ACCURACY EVALUATIONS FOR MATGPTQ

We now examine the accuracy of *MatGPTQ* across the models and benchmarks introduced previously. We compare MatGPTQ against independently quantized GPTQ, MatQuant, Any-Precision-LLM (AP) and AnyBCQ (AB). In contrast to MatQuant, which only quantizes FFN layers using OmniQuant, we quantize both FFN and attention layers using all methods.

For MatGPTQ and MatQuant, we use the target bit-width set $R = \{3, 4, 8\}$ with master bit-width $c = 8$, For MatGPTQ λ_r is chosen to balance accuracy across bit-widths (see Appendix D) that

Table 2: Unified accuracy comparison of Llama3.1-8B-Instruct and Qwen3-8B across bitwidths, for MatGPTQ against dedicated GPTQ, MatQuant, Any-Precision-LLM (AP) and AnyBCQ (AB). CSR averages ARC-c, ARC-e, HellaSwag, PIQA and WinoGrande; IFEval averages its prompt- and instance-level strict accuracies. The mean over three seeds is reported. For each format, top methods within variance are marked in **bold**, and methods within variance of the full-precision model (full recovery) are underlined.

		Llama3.1-8B-Instruct					Qwen3-8B				
Bits	Method	CSR	GSM8K	MMLU	IFEval	Avg	CSR	GSM8K	MMLU	IFEval	Avg
16	Baseline	73.77	85.75	69.32	79.01	76.96	71.52	87.57	74.88	83.76	79.43
8	GPTQ	73.80	84.84	69.32	78.53	**76.62**	71.58	87.11	74.90	83.95	**79.39**
	AP	73.81	85.65	69.33	78.16	**76.74**	71.57	87.43	74.90	84.48	**79.60**
	AB	73.60	84.10	69.11	78.07	76.22	71.74	87.49	74.71	84.48	**79.60**
	MatQuant	73.08	84.23	69.03	78.68	**76.26**	71.34	87.49	75.10	84.76	**79.68**
	MatGPTQ	73.75	84.69	69.07	77.98	**76.37**	71.30	87.36	74.73	84.65	79.51
4	GPTQ	73.66	81.27	68.65	76.79	**75.09**	71.45	86.96	73.81	83.71	**78.98**
	AP	72.67	81.27	67.27	77.35	74.64	70.81	86.44	73.55	84.24	**78.76**
	AB	72.84	80.49	67.55	77.48	74.59	70.84	86.48	73.49	84.62	**78.86**
	MatQuant	71.70	78.97	66.05	77.12	73.46	70.28	86.60	73.49	83.64	**78.50**
	MatGPTQ	73.08	82.84	68.33	77.23	**75.37**	71.28	87.14	73.64	84.09	**79.04**
	GPTQ sliced	72.49	81.43	66.78	75.70	74.10	70.00	84.61	73.05	85.99	78.41
3	GPTQ	71.72	69.90	63.62	70.82	**69.02**	69.32	80.74	70.66	79.23	**74.99**
	AP	67.49	58.50	60.64	69.66	64.07	67.08	80.83	69.43	80.33	**74.42**
	AB	71.29	64.39	63.90	75.73	**68.83**	69.50	76.35	69.11	77.62	73.14
	MatQuant	55.02	42.58	48.19	52.80	49.65	67.19	77.04	66.96	77.86	72.26
	MatGPTQ	70.45	68.28	62.96	73.62	**68.83**	68.41	80.46	69.08	79.43	**74.35**
	GPTQ sliced	65.16	33.51	50.71	55.25	51.16	57.89	54.44	57.63	55.13	56.27
6	GPTQ	73.46	84.46	69.05	78.16	**76.28**	71.17	87.04	74.84	84.31	**79.34**
	AP	73.87	84.31	69.13	78.63	**76.48**	71.55	87.45	74.97	84.41	**79.59**
	AB	73.63	84.26	69.07	77.94	**76.23**	71.65	87.29	74.77	83.77	**79.37**
	MatQuant	73.28	83.62	68.91	76.96	75.70	71.10	87.55	74.95	84.39	**79.50**
	MatGPTQ	73.69	84.82	68.93	77.77	**76.30**	71.46	87.34	74.76	83.60	79.29
	GPTQ sliced	73.51	84.53	69.11	78.78	76.48	71.33	87.72	74.83	84.31	79.55

chooses balance across bit-widths. For AP and AB, we start from a 3-bit seed model and progressively increase the bit-width up to 8 bits. We focus on this configuration because the 3–4 bit regime is particularly relevant for low-cost deployment, while still achieving strong recovery of model performance. Higher bit-widths nevertheless remain relevant, as full performance recovery is not guaranteed at 4 bits, especially on more challenging tasks (Kurtic et al., 2025; Park et al., 2024; 2026). Moreover, higher bit-widths enable non-uniform inference settings in which different layers are assigned different precisions (He et al., 2018; Frantar & Alistarh, 2022; Yin et al., 2024). Finally, MatGPTQ uses asymmetric quantization to make the comparison more fair against the other methods, which are asymmetric..

A direct comparison against the original MatQuant framework is challenging, since neither source code nor models are publicly available. Furthermore, they omit evaluation hyperparameters, making reproduction substantially more difficult. We re-implemented the OmniQuant-based MatQuant approach to produce the MatQuant models, while AP and AB models were generated using their publicly available code. Hyperparameters are detailed in Appendix B.

Main Results. In Table 2, we see the downstream performance of the models. First, we see that MatGPTQ is either better than or within variance of the independently quantized GPTQ version for all optimized bitwidths. This generally ensures that quality is maintained even though the model is optimized for different bit-widths. Second, we see in bold that MatGPTQ is within variance of the best-performing model for all bit-widths for both models, except for the 8-bit Llama3.1-8B-Instruct model, compared to MatQuant, AP, and AB. Third, we see that MatGPTQ outperforms MatQuant at nearly every bit-width for both models, especially in the low-bit regime, where MatQuant models drop in performance and do not scale to quantizing Attention+FFNs. Lastly, we apply our slicing

approach to an independently quantized 8-bit GPTQ model to evaluate whether the multi-precision objective is necessary. As shown by the gray results, the sliced 8-bit GPTQ model maintains performance at 6 and 8 bits, but increasingly degrades at 4 bits and drops substantially at 3 bits. Therefore, the MatGPTQ objective is necessary to achieve strong performance across bit-widths, while remaining competitive with AB and AP using the simpler and improved MatQuant representation and a smaller overall checkpoint footprint (see Table 1).

Interpolation. As mentioned in Nair et al. (2025), also MatGPTQ shows strong performance when interpolating bit-widths that are not directly optimized for. As we see in Table 2 both models are on par with the GPTQ baseline. Since we provide CUDA kernel support for 2, 3, 4, 6, and 8 bits, we are also interested in being able to use a 6-bit model. We can see in Table 2 that the interpolated 6-bit MatGPTQ model remains within variance of the independently quantized GPTQ model and achieves full performance recovery for both models, showcasing strong interpolation performance.

5.3 CUDA Kernel Performance

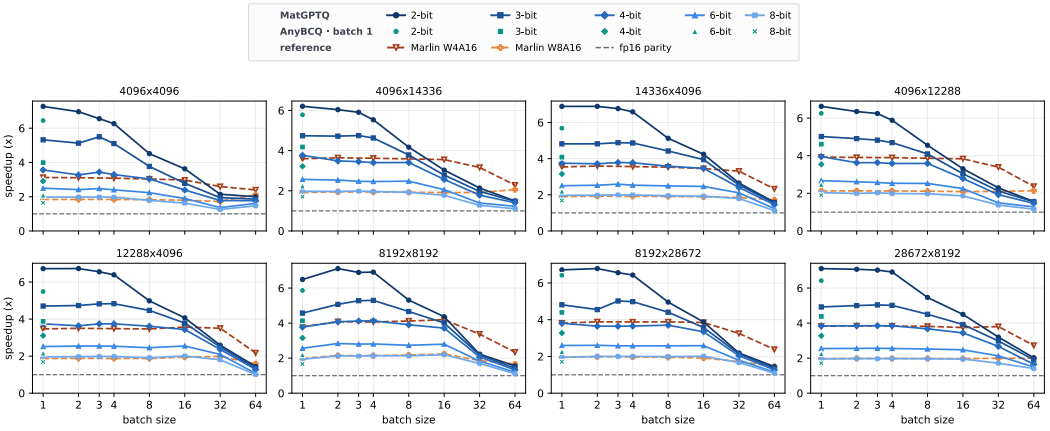


Figure 1: The figure shows the kernel speedup over FP16 `torch.matmul` (cuBLAS). The kernels are evaluated at 2–8 bit precision for several representative matrix shapes across different batch sizes. AnyBCQ and Marlin W4A16/W8A16 are included as reference baselines. However, AnyBCQ results are only shown for batch size 1.

We benchmark the kernel latency for common weight matrix sizes of models such as Llama3.1-8B, Qwen3-8B, and Llama3-70B, while varying the input size. These latency measurements were taken in an isolated environment, purely evaluating kernel performance and ignoring overhead introduced by Hugging Face or other libraries. We use `torch.matmul` (cuBLAS) as a baseline for comparison. All configurations use a group size of 128. As seen in Figure 1, our kernels achieve speedups between 2x and 6.5x in the memory-bound regime. The peak speedup is achieved for 2-bit single-batch multiplication, owing to the memory-bound nature of this setting. At higher batch sizes, the kernel transitions to a compute-bound regime, and we observe slowdowns at around batch size 32.

Furthermore, we integrated the AnyBCQ kernels into our benchmark harness, as AnyBCQ reports faster kernels than Any-Precision-LLM, as another point of comparison (Park et al., 2026). As seen in Figure 1, at batch size 1, AnyBCQ kernels, visualized with green dots, are consistently slower than MatGPTQ kernels for all bit-widths across all shapes. At larger batch sizes, AnyBCQ provides no dedicated kernel and falls back to dequantization followed by cuBLAS, whereas our kernels continue to operate directly on the packed representation. This leads to significant performance degradation, highlighting our contribution.

Additionally, we benchmarked the highly optimized vLLM-Marlin kernel for 4- and 8-bit quantization. As shown in Figure 1, indicated by the red and orange lines, MatGPTQ at 4 and 8 bits remains competitive with the dedicated Marlin kernels and only falls behind from batch size 16 onward.

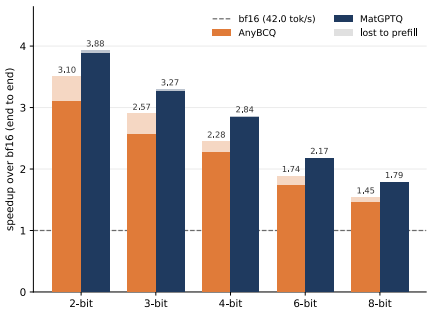


Figure 2: End-to-end speedups: Transformers – Llama3.1-8B-Instruct, Prompt/Decode 32/128.

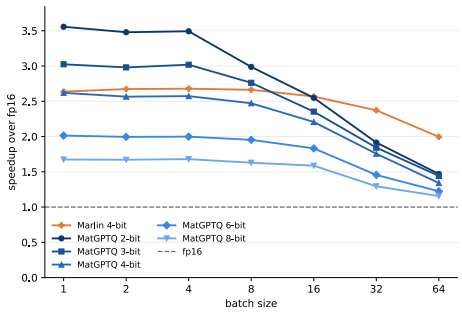


Figure 3: End-to-end speedups: vLLM – Llama3.1-8B-Instruct, Prompt/Decode 32/128.

5.4 End-to-End Performance

We integrate our kernels into the widely used PyTorch-based Hugging Face framework (Wolf et al., 2020), as well as vLLM (Kwon et al., 2023) for real-world deployment. Using those two system adaptations, we measure end-to-end throughput (token/s) while varying the model bit-width. For the Hugging Face framework, we utilize `torch.compile` under `reduce-overhead` and `full-graph` settings, while for vLLM we utilize the default torch.compile mode.

In Figure 2, we can see that the speed-up over the 16-bit version naturally diminishes as bit-widths increase. In the low-bit regime (2–4 bits), our kernels achieve end-to-end speedups of ≈2.8x to 3.9x. Furthermore, we see that we outperform the AnyBCQ kernels at every bit-width and can clearly see the difference in prefill time, especially in the low-bit regime. Additional results for larger prompt/decoding example can be found in Appendix E.

These improvements also translate to our vLLM integration. Overall speedups between 1.7x and 3.5x can be achieved in the memory-bound regime. Furthermore, we include results using the highly optimized vLLM-Marlin kernel execution at 4-bit precision. As shown in Figure 3, MatGPTQ remains competitive at smaller batch sizes and falls behind when leaving the memory-bound regime.

6 Discussion & Limitations

In this work, we close the gaps in nested LLM inference by introducing MatGPTQ, the first practical and efficient framework for post-training Matryoshka quantization. By adapting the GPTQ algorithm to optimize a single model jointly for multiple target precisions, MatGPTQ enables "sliceable" quantization in a one-shot manner. Furthermore, the improved representation enables true-prefix slicing. Together with a kernel that supports both prefill and decoding at any batch size, we address the limitations of current state-of-the-art methods. Therefore, a single quantized checkpoint can be obtained quickly and adapted at runtime to meet diverse memory and latency budgets across various hardware without the overhead of storing and maintaining multiple models. Our extensive evaluation demonstrates that this flexibility comes at virtually no cost to accuracy relative to the GPTQ baseline, outperforming or being on par with current state-of-the-art methods, while using the simplest format out of all methods.

In addition, we provide the first open-source implementation of Matryoshka quantization, built on the Hugging Face framework, along with efficient CUDA kernels and a vLLM integration.

Limitations and Future Work. Our experiments focused on a limited range of bit-widths and on integer quantization. Achieving high accuracy in the ultra-low-bit regime remains challenging with the current GPTQ algorithm at standard group sizes. In future work, we will explore integrating Matryoshka objectives with quantization methods better suited for ultra-low precisions, as well as floating-point quantization. Furthermore, while our CUDA kernels currently target NVIDIA Ampere, Ada and Hopper GPUs, future work will expand support to newer architectures such as Blackwell.

AI USE STATEMENT

In this work, we used generative AI tools to assist with prototyping, including PyTorch-based quantization algorithms, CUDA kernels, and evaluation code. All AI-assisted code was reviewed, tested, and validated by the authors to ensure correctness. Generative AI was also used to improve the clarity and readability of text initially written by the authors.

We take full responsibility for the final content of this work, including all text, claims, code, and artifacts produced with the assistance of generative AI.

REPRODUCIBILITY STATEMENT

All models, benchmarks, and system and hardware setups are described in Section 5.1, while all hyperparameters are described in Appendix B. Furthermore, all detailed benchmark results can be found in Appendix C. The PTQ pipeline, CUDA kernels, and benchmarking suites are attached as supplementary materials. All models and datasets used in this work are publicly available.

REFERENCES

Hicham Badri and Appu Shaji. Half-quadratic quantization of large machine learning models, November 2023. URL https://mobiusml.github.io/hqq_blog/.

Yonatan Bisk, Rowan Zellers, Ronan LeBras, Jianfeng Gao, and Yejin Choi. PIQA: reasoning about physical commonsense in natural language. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pp. 7432–7439. AAAI Press, 2020. URL https://ojs.aaai.org/index.php/AAAI/article/view/6239.

Hao Mark Chen, Fuwen Tan, Alexandros Kouris, Royson Lee, Hongxiang Fan, and Stylianos Venieris. Progressive mixed-precision decoding for efficient LLM inference. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=OVxmpus9NA.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

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. *arXiv preprint arXiv:2110.14168*, 2021.

Tim Dettmers. 8-bit approximations for parallelism in deep learning. *International Conference on Learning Representations (ICLR)*, 2016.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, and et al. The Llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Elias Frantar and Dan Alistarh. SPDY: Accurate pruning with speedup guarantees. In *International Conference on Machine Learning (ICML)*, 2022.

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. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '25, pp. 239–251, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400714436. doi: 10.1145/3710848.3710871. URL https://doi.org/10.1145/3710848.3710871.

Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. AMC: AutoML for model compression and acceleration on mobile devices. In *European Conference on Computer Vision (ECCV)*, 2018.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.

Jinhee Kim, Seoyeon Yoon, Taeho Lee, Joo Chan Lee, Kang Eun Jeon, and Jong Hwan Ko. Truncquant: Truncation-ready quantization for dnns with flexible weight bit precision. *arXiv preprint arXiv:2506.11431*, 2025.

Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W. Mahoney, and Kurt Keutzer. Squeezellm: dense-and-sparse quantization. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24, 2024.

Young Jin Kim, Rawn Henry, Raffy Fahim, and Hany Hassan. Who says elephants can't run: Bringing large scale MoE models into cloud scale production. In Angela Fan, Iryna Gurevych, Yufang Hou, Zornitsa Kozareva, Sasha Luccioni, Nafise Sadat Moosavi, Sujith Ravi, Gyuwan Kim, Roy Schwartz, and Andreas Rücklé (eds.), *Proceedings of the Third Workshop on Simple and Efficient Natural Language Processing (SustaiNLP)*, pp. 36–43, Abu Dhabi, United Arab Emirates (Hybrid), December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.sustainlp-1.6. URL https://aclanthology.org/2022.sustainlp-1.6/.

Eldar Kurtic, Alexandre Noll Marques, Shubhra Pandit, Mark Kurtz, and Dan Alistarh. "give me BF16 or give me death"? accuracy-performance trade-offs in LLM quantization. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 26872–26886, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.1304. URL https://aclanthology.org/2025.acl-long.1304/.

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 *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

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 llm compression and acceleration. In *Proceedings of the Learning on Systems (MLSys) Conference*, 2024.

Pranav Ajit Nair, Puranjay Datta, Jeff Dean, Prateek Jain, and Aditya Kusupati. Matryoshka quantization. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=phVWcUSGYP.

Gunho Park, Jeongin Bae, Beomseok Kwon, Byeongwook Kim, Se Jung Kwon, and Dongsoo Lee. Anybcq: Hardware efficient flexible binary-coded quantization for multi-precision llms. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 125106–125122, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/ca9eaef07eca2a50fc626cb929617b1c-Paper-Conference.pdf.

Yeonhong Park, Jake Hyun, SangLyul Cho, Bonggeun Sim, and Jae W. Lee. Any-precision llm: low-cost deployment of multiple, different-sized llms. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24, 2024.

Guilherme Penedo, Hynek Kydlíček, Loubna Ben allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. The fineweb datasets: Decanting the web for the finest text data at scale. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 30811–30849. Curran Associates, Inc., 2024. doi: 10.52202/079017-0970.

URL https://proceedings.neurips.cc/paper_files/paper/2024/file/370df50ccfdf8bde18f8f9c2d9151bda-Paper-Datasets_and_Benchmarks_Track.pdf.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2021.

Wenqi Shao, Mengzhao Chen, Zhaoyang Zhang, Peng Xu, Lirui Zhao, Zhiqian Li, Kaipeng Zhang, Peng Gao, Yu Qiao, and Ping Luo. Omniquant: Omnidirectionally calibrated quantization for large language models. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=8Wuvhh0LYW.

Haodong Wang, Qihua Zhou, Zicong Hong, and Song Guo. D^2moe: Dual routing and dynamic scheduling for efficient on-device moe-based llm serving. In *Proceedings of the 31st Annual International Conference on Mobile Computing and Networking*, ACM MOBICOM '25, pp. 574–588, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400711299. doi: 10.1145/3680207.3723493. URL https://doi.org/10.1145/3680207.3723493.

Lucas Wilkinson. Introducing machete, a mixed-input gemm kernel optimized for nvidia hopper gpus. Red Hat Developer, October 2024. URL https://developers.redhat.com/articles/2024/10/14/introducing-machete-mixed-input-gemm-kernel. Accessed: 2025-10-30.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-art natural language processing. In Qun Liu and David Schlangen (eds.), *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, Online, October 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-demos.6. URL https://aclanthology.org/2020.emnlp-demos.6/.

Jianhang Xie, Chuntao Ding, Xiaqing Li, Shenyuan Ren, Yidong Li, and Zhichao Lu. Nestquant: Post-training integer-nesting quantization for on-device dnn. *IEEE Transactions on Mobile Computing*, 24(11):12088–12102, 2025. doi: 10.1109/TMC.2025.3582583.

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.

Lu Yin, You Wu, Zhenyu Zhang, Cheng-Yu Hsieh, Yaqing Wang, Yiling Jia, Gen Li, Ajay Jaiswal, Mykola Pechenizkiy, Yi Liang, Michael Bendersky, Zhangyang Wang, and Shiwei Liu. Outlier weighed layerwise sparsity (owl): a missing secret sauce for pruning llms to high sparsity. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

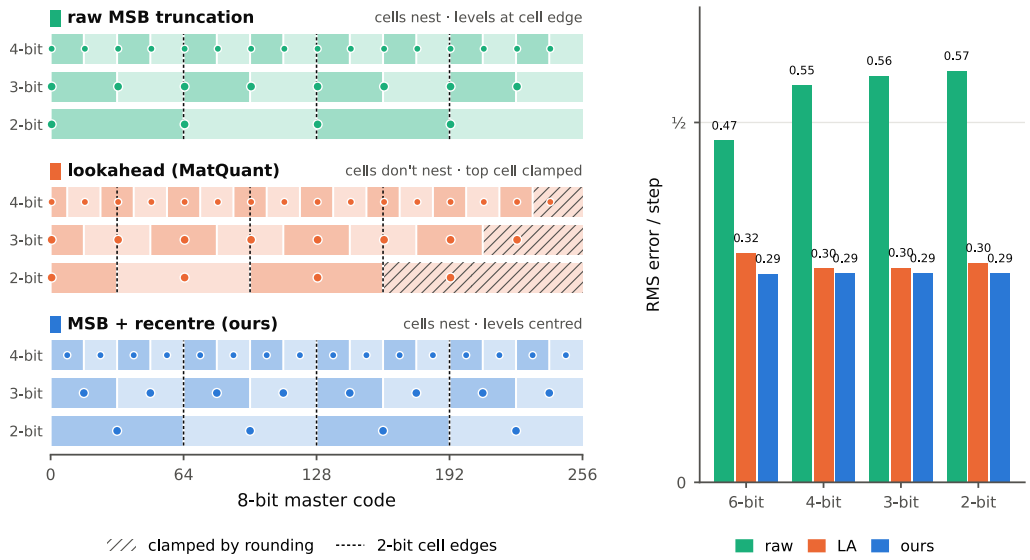
Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. HellaSwag: Can a machine really finish your sentence? In Anna Korhonen, David Traum, and Lluís Màrquez (eds.), *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 4791–4800, Florence, Italy, July 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1472. URL https://aclanthology.org/P19-1472/.

Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. Instruction-following evaluation for large language models, 2023. URL https://arxiv.org/abs/2311.07911.

A MatGPTQ implementation details

This section introduces additional implementation details about the MatGPTQ algorithm including the improved representation, the MSE-based scale search and the generation of quantized weights.

A.1 Improved nested representation



(a) Which master codes share a level.

(b) RMS slice error.

Figure 4: Slicing an 8-bit master code to lower bit-widths. a Raw truncation, MatQuant rounding, and our recentered truncation. Shaded regions indicate the corresponding cells, while dots denote the reconstructed quantized values. Our method preserves nested cells while centering the reconstruction within each cell. b RMS reconstruction error across bit-widths. Our recentered truncation achieves the lowest error.

The general idea behind slicing in MatQuant is to approximate a c-bit model using r bits. First, a right shift keeps the r most significant bits, effectively slicing the c-bit code. A subsequent left shift maps this r-bit prefix back into the c-bit code space, yielding the reconstructed quantized value. However, raw MSB truncation systematically reconstructs values at the lower edge of each cell, as seen in Figure 4, indicated by the green grid. As a result, the reconstructed r-bit values are, on average, farther from the corresponding c-bit values, leading to a higher root-mean-square (RMS) reconstruction error.

To avoid this issue, the MatQuant authors push values into higher buckets (Nair et al., 2025), which introduces the following slicing algorithm: MatQuant slicing (Equation 1) extracts the r most significant bits, uses the $(r+1)^{\text{th}}$ bit for rounding, and then left-shifts the result back into the c-bit code space to obtain the reconstructed quantized value. Since the rounding step can lead to overflow, $\text{clamp}(\cdot)$ is needed to ensure that the sliced values remain within the valid r-bit range. This can be seen in Figure 4, indicated by the orange grid.

To avoid extracting an additional bit and relying on the $\text{clamp}(\cdot)$ function, we analyzed raw MSB truncation. Since the right and left shifts reconstruct the sliced value at the lower edge of each cell, we shift this reconstructed quantized value toward the center of the cell. This can be achieved by simply adding δ_r, resulting in the blue grid shown in Figure 4. By centering the reconstructed value within each cell, the distances to the corresponding c-bit values are balanced, which minimizes the RMS reconstruction error for the fixed nested partition. Additionally, δ_r can be absorbed into the zero-point and therefore naturally integrated without additional overhead.

A.2 MatGPTQ adaptation

This section explains the adaptations of the GPTQ algorithm. Algorithm 1 adapts GPTQ's scale search to the multi-precision setting using an MSE-based objective that incorporates the bit-width-specific weights λ_r.

Algorithm 1 fit: shared $(\mathbf{s}, \mathbf{z})$ of a weight group, chosen against every served bit-width $r \in R$ with weights λ_r (per row)

input group $\mathbf{X} \in \mathbb{R}^{d_{\text{row}} \times g}$, Hessian diagonal $\mathbf{h}$, bit-widths R, weights λ_R, grid G, offsets K, shrink σ
$\mathbf{x}_{\min}, \mathbf{x}_{\max} \leftarrow$ row-wise min, max of $\mathbf{X}$; $\quad d \leftarrow c - \min R$
$\mathcal{S} \leftarrow \left\{ p\,(\mathbf{x}_{\max} - \mathbf{x}_{\min})/(2^c - 1) \;:\; p \in [\sigma, 1],\ G \text{ steps} \right\}$ ▷ shrunk scales
$\mathcal{Z}(\mathbf{s}) \leftarrow \left\{ \operatorname{round}(-\mathbf{x}_{\min}/\mathbf{s}) + k\,2^d/K \;:\; k < K \right\}$ ▷ shifted zeros
for $\mathbf{s} \in \mathcal{S}, \mathbf{z} \in \mathcal{Z}(\mathbf{s})$ **do**
  $\mathbf{Q} \leftarrow \operatorname{clip}\big(\operatorname{round}(\mathbf{X}/\mathbf{s} + \mathbf{z}),\ 0,\ 2^c - 1\big)$ ▷ round-to-nearest proxy
  $\mathcal{E}(\mathbf{s}, \mathbf{z}) \leftarrow \sum_{r \in R} \lambda_r \left\| \sqrt{\mathbf{h}} \odot \big(\mathbf{X} - \mathbf{s}(S'(\mathbf{Q}, r) + \delta_r - \mathbf{z})\big) \right\|_2^2$ ▷ weighted error of all slices
end for
return $\arg\min_{\mathbf{s},\mathbf{z}} \mathcal{E}(\mathbf{s}, \mathbf{z})$

Algorithm 2 returns the quantized weights, by first generating all possible quantized values in the range $[0, Q_{\max}]$ for each individual weight. For every target bit-width, these candidates are then dequantized using the respective scale and zero-point. The reconstruction error is computed per weight and accumulated across all candidates, weighted by their respective importance. This results in the total error for each potential quantized value of every weight. Finally, for each weight, the quantized value with the lowest total error is selected. Even though this is a brute-force approach and therefore memory and runtime concerns exist, the overhead is generally minimal.

Algorithm 2 $\text{quant}_{\text{MatGPTQ}}$: one master code per weight, chosen for every served bit-width at once. $S'(q, r)$ is the r-bit reconstruction of code q.

input column $\mathbf{w} \in \mathbb{R}^{d_{\text{row}}}$, scale s, zero-point z, bit-widths R, weights λ_R, master bit-width c
$\mathbf{q} \leftarrow \arg\min_{0 \le q < 2^c} \sum_{r \in R} \lambda_r \big(\mathbf{w} - s(S'(q, r) + \delta_r - z)\big)^2$ ▷ elementwise over rows
return $\mathbf{q}$

Runtime & memory considerations for $\texttt{quant}_{MatGPTQ}(\cdot)$. With Algorithm 2 we want to optimize the quantization grid for multiple bit-widths. We decided to take a simple brute-force approach instead of relying on any heuristics, as this approach is still highly vectorizable and the resulting overhead appeared acceptable considering our target of optimizing for multiple bit-widths.

Table 3: Runtime comparison of per-column functions and full quantization passes. GPTQ×3 is the sum of separate 3-, 4-, and 8-bit GPTQ passes. Speedup is (GPTQ×3)/MatGPTQ.

	Per-column function (ms)		Complete pass (s)			
Shape	`quant`	`quant_MatGPTQ`	GPTQ	GPTQ×3	MatGPTQ	Speedup
k/v (1024×4096)	0.0599	0.5360	1.45	4.34	3.59	1.21×
q/o (4096×4096)	0.0601	0.5368	1.47	4.41	3.59	1.23×
gate/up (14336×4096)	0.0610	1.3444	1.41	4.24	6.58	0.64×
down (4096×14336)	0.0601	0.5368	5.41	16.24	12.45	1.30×
Layer total	–	–	14.07	42.21	39.96	1.06×

To validate this, we measured execution times for layer sizes of a Llama3.1-8B-Instruct model, both for the isolated per-column functions and complete (Mat)GPTQ passes. Generally, we can see that `quant_MatGPTQ` is slower than the native `quant` function. Therefore, individual MatGPTQ quantization passes are slower than GPTQ passes. However, when comparing the cost of quantizing three separate GPTQ models with one MatGPTQ model, we see that MatGPTQ is faster on average.

Furthermore, Algorithm 2 would indeed have higher peak memory usage than basic rounding in GPTQ, due to maintaining a matrix size of $2^c \times d_{row} \times d_{col}$. However, in practice, both the original GPTQ and our MatGPTQ implementation quantize only one column at a time. Therefore, the matrix has size $2^c \times d_{row} \times 1$, which, for a matrix of 4096×4096, results in only 4 MB of overhead when stored in FP32 ($256 \times 4096 \times 1 \times 4$ bytes). This overhead is therefore insignificant in practice.

B Hyperparameter details

This section presents the hyperparameters used for the respective methods.

MatGPTQ and GPTQ. For both the native GPTQ and MatGPTQ, we use the Fineweb-Edu (Penedo et al., 2024) dataset as calibration data. Specifically, we use 2,097,152 tokens, corresponding to 1024 samples with a sequence length of 2048. Furthermore, we use a group size of 128, a dampening factor of 0.01, MSE-based scale search and asymmetric quantization. For completeness, we optimized for bit-widths $R = \{3, 4, 8\}$ and used the most balanced config as elaborated in Section D

MatQuant. For MatQuant (Nair et al., 2025), we followed the specific hyperparameters provided by the authors: For Llama3.1-8B-Instruct and Qwen3-8B we also used $\lambda_3 = 1$, $\lambda_4 = 0.1$, and $\lambda_8 = 0.1$. We used their constant learning rate of 10^{-3} and a batch size of 4. We only differ by using asymmetric quantization and optimizing for $r = \{3, 4, 8\}$. Additionally, we apply grouping of size 128 to make the comparison even fairer, even though they never mention anything regarding grouping.

Any-Precision-LLM. For Any-Precision-LLM (Park et al., 2024), we use the authors' reference implementation and its default hyperparameters. The sensitivity gradients and the k-means seed model are computed on the Fineweb-Edu (Penedo et al., 2024) dataset with 1024 calibration samples of sequence length 2048. We use a seed precision of 3 and a parent precision of 8, so that a single checkpoint supports $r \in \{3, \ldots, 8\}$.

AnyBCQ. For AnyBCQ (Park et al., 2026), we use the authors' reference implementation and its default hyperparameters. The binary weight planes and their per-group scales are fitted on the Fineweb-Edu (Penedo et al., 2024) dataset with 1024 calibration samples of sequence length 2048, using a group size of 128 and asymmetric quantization, with an input probability of 1.0 for Llama-3.1-8B-Inst and 0.5 for Qwen3-8B. We quantize from 3 to 8 bits, so that a single checkpoint stores 8 binary planes together with per-bit-width scales and offsets and supports $r \in \{3, \ldots, 8\}$, all other settings are left at their defaults.

C Detailed Accuracy Results

Tables 4 and 5 present detailed evaluation results of Llama3.1-8B-Instruct and Qwen3-8B with MatGPTQ versus independently quantized GPTQ, MatQuant, Any-Precision-LLM (AP) and AnyBCQ models. All methods use asymmetric schemes.

Measuring variance. Every entry in Table 2 is the mean over 3 independently quantized models, each fitted from a different calibration seed and evaluated once. Because quantization is fitted on a small random sample of calibration data, we wanted to measure how far a score moves when that draw changes. Writing s_i and n_i for the sample standard deviation and number of calibration seeds of method i, the standard error of the difference between two entries is

$$\mathrm{SE}_{a-b} = \sqrt{\frac{s_a^2}{n_a} + \frac{s_b^2}{n_b}}, \tag{7}$$

and we mark an entry **bold** when a 95% Welch interval, $t_{0.975,\nu}\,\mathrm{SE}_{a-b}$ with Welch-Satterthwaite degrees of freedom ν, contains its gap to the best entry in the block, underlined entries are indistinguishable from the full-precision model in the same sense. A bold entry is therefore one we cannot separate from the best given the seeds we have.

Table 4: Evaluation results for asymmetric MatGPTQ, GPTQ, MatQuant, Any-Precision-LLM (AP) and AnyBCQ (AB) on Llama3.1-8B-Instruct. Accuracy on the evaluation tasks (↑): ARC-c (ARC-Challenge), ARC-e (ARC-Easy), HellaSwag, PIQA, WinoG (WinoGrande), GSM8K, MMLU, and IF-p and IF-i, the prompt- and instance-level strict IFEval accuracies. The mean over three calibration seeds is reported.

Llama3.1-8B-Instruct										
Bitwidth	Method	ARC-c	ARC-e	HellaSwag	PIQA	WinoG	GSM8K	MMLU	IF-p	IF-i
16		55.20	79.55	79.22	81.01	73.88	85.75	69.32	75.42	82.61
8	GPTQ	$54.95_{0.19}$	$79.80_{0.13}$	$79.21_{0.03}$	$80.85_{0.05}$	$74.19_{0.28}$	$84.84_{0.50}$	$69.32_{0.05}$	$75.05_{0.71}$	$82.01_{0.61}$
	AP	$55.15_{0.27}$	$79.67_{0.11}$	$79.22_{0.04}$	$80.96_{0.14}$	$74.06_{0.05}$	$85.65_{0.42}$	$69.33_{0.06}$	$74.43_{0.59}$	$81.89_{0.32}$
	AB	$55.83_{0.74}$	$79.66_{0.53}$	$78.39_{0.15}$	$81.16_{0.03}$	$72.98_{0.12}$	$84.10_{0.66}$	$69.11_{0.15}$	$74.49_{0.67}$	$81.65_{0.62}$
	MatQuant	$53.90_{0.13}$	$78.63_{0.11}$	$78.93_{0.02}$	$81.08_{0.03}$	$72.88_{0.24}$	$84.23_{0.08}$	$69.03_{0.04}$	$75.11_{0.11}$	$82.25_{0.24}$
	MatGPTQ	$54.93_{0.27}$	$79.83_{0.19}$	$78.87_{0.06}$	$81.02_{0.17}$	$74.11_{0.06}$	$84.69_{0.23}$	$69.07_{0.15}$	$74.23_{0.43}$	$81.73_{0.51}$
4	GPTQ	$55.03_{0.34}$	$79.92_{0.41}$	$78.27_{0.14}$	$80.79_{0.27}$	$74.27_{0.40}$	$81.27_{1.02}$	$68.65_{0.18}$	$73.01_{1.74}$	$80.58_{1.23}$
	AP	$53.95_{0.51}$	$78.56_{0.49}$	$77.83_{0.26}$	$80.45_{0.49}$	$72.53_{1.32}$	$81.27_{1.07}$	$67.27_{0.32}$	$73.69_{1.20}$	$81.02_{0.84}$
	AB	$54.12_{0.58}$	$79.24_{0.38}$	$77.51_{0.10}$	$80.65_{0.08}$	$72.69_{0.34}$	$80.49_{0.16}$	$67.55_{0.08}$	$73.75_{0.37}$	$81.22_{0.30}$
	MatQuant	$51.59_{0.18}$	$78.03_{0.17}$	$77.31_{0.08}$	$79.83_{0.08}$	$71.74_{0.44}$	$78.97_{0.29}$	$66.05_{0.10}$	$73.26_{0.59}$	$80.98_{0.50}$
	MatGPTQ	$53.96_{0.50}$	$78.65_{0.78}$	$78.42_{0.10}$	$80.76_{0.11}$	$73.62_{0.72}$	$82.84_{0.94}$	$68.33_{0.27}$	$73.42_{1.40}$	$81.03_{0.99}$
	GPTQ sliced	$52.73_{0.30}$	$78.11_{0.41}$	$78.17_{0.12}$	$79.82_{0.23}$	$73.64_{0.43}$	$81.43_{1.01}$	$66.78_{0.20}$	$71.53_{1.85}$	$79.86_{1.20}$
3	GPTQ	$53.58_{1.37}$	$78.32_{1.25}$	$76.21_{0.25}$	$78.56_{0.51}$	$71.90_{0.72}$	$69.90_{0.84}$	$63.62_{0.35}$	$65.99_{1.55}$	$75.66_{1.27}$
	AP	$47.38_{2.69}$	$72.57_{2.24}$	$73.24_{1.62}$	$76.57_{1.56}$	$67.69_{1.28}$	$58.50_{3.11}$	$60.64_{0.63}$	$64.94_{0.91}$	$74.38_{0.82}$
	AB	$51.68_{0.18}$	$77.69_{0.11}$	$75.23_{0.07}$	$79.45_{0.17}$	$72.40_{0.12}$	$64.39_{0.36}$	$63.90_{0.08}$	$71.72_{0.64}$	$79.74_{0.52}$
	MatQuant	$29.81_{0.55}$	$49.09_{0.98}$	$67.94_{0.23}$	$65.14_{0.22}$	$63.14_{0.36}$	$42.58_{0.94}$	$48.19_{0.08}$	$47.01_{1.87}$	$58.59_{1.63}$
	MatGPTQ	$50.24_{0.38}$	$75.88_{0.97}$	$75.26_{0.28}$	$78.82_{0.38}$	$72.04_{0.65}$	$68.28_{0.75}$	$62.96_{0.55}$	$69.54_{2.67}$	$77.70_{2.11}$
	GPTQ sliced	$43.94_{1.32}$	$66.04_{1.21}$	$71.34_{0.23}$	$76.44_{0.53}$	$68.03_{0.71}$	$33.51_{0.84}$	$50.71_{0.40}$	$49.35_{1.50}$	$61.15_{1.24}$
6	GPTQ	$54.18_{0.35}$	$79.12_{0.44}$	$79.23_{0.01}$	$81.07_{0.15}$	$73.72_{0.30}$	$84.46_{0.40}$	$69.05_{0.16}$	$74.31_{0.55}$	$82.01_{0.40}$
	AP	$55.32_{0.44}$	$79.46_{0.15}$	$79.24_{0.11}$	$81.08_{0.30}$	$74.24_{0.55}$	$84.31_{0.42}$	$69.13_{0.16}$	$75.05_{0.81}$	$82.21_{0.88}$
	AB	$55.49_{0.13}$	$80.15_{0.38}$	$78.48_{0.14}$	$80.90_{0.05}$	$73.14_{0.58}$	$84.26_{0.77}$	$69.07_{0.13}$	$74.31_{1.95}$	$81.57_{1.14}$
	MatQuant	$54.41_{0.05}$	$78.37_{0.04}$	$78.89_{0.01}$	$81.34_{0.11}$	$73.40_{0.16}$	$83.62_{0.15}$	$68.91_{0.04}$	$73.07_{0.87}$	$80.86_{0.87}$
	MatGPTQ	$54.88_{0.42}$	$79.60_{0.18}$	$78.82_{0.07}$	$81.01_{0.18}$	$74.14_{0.49}$	$84.82_{1.06}$	$68.93_{0.12}$	$73.94_{0.61}$	$81.61_{0.60}$
	GPTQ sliced	$54.52_{0.33}$	$79.55_{0.45}$	$79.18_{0.01}$	$81.23_{0.17}$	$73.09_{0.31}$	$84.53_{0.38}$	$69.11_{0.10}$	$75.42_{0.58}$	$82.13_{0.41}$

Table 5: Evaluation results for asymmetric MatGPTQ, GPTQ, MatQuant, Any-Precision-LLM (AP) and AnyBCQ (AB) on Qwen3-8B. Accuracy on the evaluation tasks (↑): ARC-c (ARC-Challenge), ARC-e (ARC-Easy), HellaSwag, PIQA, WinoG (WinoGrande), GSM8K, MMLU, and IF-p and IF-i, the prompt- and instance-level strict IFEval accuracies. The mean over three calibration seeds is reported.

Qwen3-8B										
Bitwidth	Method	ARC-c	ARC-e	HellaSwag	PIQA	WinoG	GSM8K	MMLU	IF-p	IF-i
16		56.48	80.64	74.93	77.69	67.88	87.57	74.88	80.59	86.93
8	GPTQ	$56.66_{0.27}$	$80.81_{0.07}$	$74.98_{0.04}$	$77.58_{0.06}$	$67.88_{0.10}$	$87.11_{0.16}$	$74.90_{0.04}$	$80.96_{0.59}$	$86.93_{0.38}$
	AP	$56.52_{0.11}$	$80.92_{0.09}$	$74.93_{0.03}$	$77.70_{0.05}$	$67.80_{0.37}$	$87.43_{0.28}$	$74.90_{0.05}$	$81.55_{0.36}$	$87.41_{0.35}$
	AB	$56.40_{0.78}$	$80.88_{0.57}$	$74.72_{0.24}$	$77.51_{0.11}$	$69.17_{0.41}$	$87.49_{1.14}$	$74.71_{0.25}$	$81.52_{1.03}$	$87.45_{0.50}$
	MatQuant	$56.26_{0.29}$	$80.56_{0.07}$	$74.75_{0.08}$	$77.27_{0.12}$	$67.86_{0.31}$	$87.49_{0.33}$	$75.10_{0.07}$	$82.07_{0.23}$	$87.46_{0.14}$
	MatGPTQ	$56.03_{0.65}$	$80.13_{0.28}$	$74.68_{0.03}$	$77.53_{0.20}$	$68.14_{0.39}$	$87.36_{0.23}$	$74.73_{0.12}$	$81.76_{0.28}$	$87.53_{0.42}$
4	GPTQ	$55.89_{0.55}$	$80.60_{0.90}$	$74.66_{0.13}$	$77.53_{0.40}$	$68.59_{0.49}$	$86.96_{0.39}$	$73.81_{0.23}$	$80.96_{0.83}$	$86.45_{0.57}$
	AP	$55.34_{0.42}$	$79.32_{0.68}$	$73.77_{0.23}$	$77.38_{0.38}$	$68.26_{0.66}$	$86.44_{0.76}$	$73.55_{0.29}$	$81.40_{0.90}$	$87.07_{0.52}$
	AB	$55.20_{0.44}$	$79.63_{0.29}$	$73.55_{0.12}$	$77.86_{0.22}$	$67.96_{0.44}$	$86.48_{0.04}$	$73.49_{0.02}$	$81.95_{0.75}$	$87.29_{0.52}$
	MatQuant	$53.46_{0.42}$	$79.96_{0.19}$	$73.32_{0.14}$	$77.16_{0.30}$	$67.48_{0.60}$	$86.60_{0.47}$	$73.49_{0.16}$	$80.70_{0.98}$	$86.57_{0.63}$
	MatGPTQ	$55.97_{0.62}$	$80.18_{0.45}$	$73.93_{0.05}$	$77.69_{0.05}$	$68.64_{1.07}$	$87.14_{0.19}$	$73.64_{0.19}$	$81.33_{0.81}$	$86.85_{0.72}$
	GPTQ sliced	$53.58_{0.52}$	$77.10_{0.91}$	$73.61_{0.13}$	$77.09_{0.34}$	$68.59_{0.52}$	$84.61_{0.30}$	$73.05_{0.30}$	$83.36_{0.42}$	$88.61_{0.61}$
3	GPTQ	$52.05_{0.98}$	$73.95_{0.99}$	$72.41_{0.22}$	$77.42_{0.68}$	$70.80_{0.29}$	$80.74_{0.70}$	$70.66_{0.33}$	$75.60_{0.52}$	$82.85_{0.65}$
	AP	$49.78_{1.41}$	$74.67_{2.03}$	$69.95_{0.31}$	$76.16_{0.43}$	$64.83_{1.07}$	$80.83_{1.23}$	$69.43_{0.45}$	$76.97_{1.31}$	$83.69_{1.04}$
	AB	$53.07_{0.20}$	$79.59_{0.10}$	$70.71_{0.05}$	$77.53_{0.08}$	$66.61_{0.12}$	$76.35_{0.04}$	$69.11_{0.13}$	$73.94_{0.40}$	$81.29_{0.59}$
	MatQuant	$51.13_{0.64}$	$75.45_{0.90}$	$68.82_{0.22}$	$75.24_{0.28}$	$65.30_{0.66}$	$77.04_{0.93}$	$66.96_{0.15}$	$74.53_{0.92}$	$81.20_{0.81}$
	MatGPTQ	$51.48_{1.91}$	$76.21_{1.73}$	$70.93_{0.21}$	$76.44_{0.42}$	$66.98_{0.91}$	$80.46_{1.55}$	$69.08_{0.47}$	$76.09_{0.38}$	$82.77_{0.18}$
	GPTQ sliced	$39.51_{1.02}$	$58.46_{1.08}$	$60.95_{0.22}$	$71.11_{0.75}$	$59.43_{0.35}$	$54.44_{0.58}$	$57.63_{0.40}$	$48.98_{0.53}$	$61.27_{0.70}$
6	GPTQ	$55.80_{0.24}$	$80.13_{0.22}$	$74.67_{0.08}$	$77.31_{0.26}$	$67.96_{0.23}$	$87.04_{0.17}$	$74.84_{0.11}$	$81.33_{0.71}$	$87.29_{0.47}$
	AP	$56.74_{0.45}$	$80.97_{0.36}$	$74.84_{0.11}$	$77.46_{0.19}$	$67.72_{0.54}$	$87.45_{0.22}$	$74.97_{0.13}$	$81.48_{0.51}$	$87.34_{0.36}$
	AB	$56.31_{0.52}$	$80.61_{0.50}$	$74.65_{0.18}$	$77.62_{0.22}$	$69.03_{0.66}$	$87.29_{0.68}$	$74.77_{0.08}$	$80.65_{0.11}$	$86.89_{0.18}$
	MatQuant	$55.78_{0.28}$	$80.28_{0.11}$	$74.74_{0.10}$	$76.97_{0.14}$	$67.73_{0.17}$	$87.55_{0.33}$	$74.95_{0.07}$	$81.55_{0.68}$	$87.22_{0.48}$
	MatGPTQ	$56.43_{0.64}$	$80.01_{0.08}$	$74.73_{0.02}$	$77.57_{0.26}$	$68.59_{0.49}$	$87.34_{0.20}$	$74.76_{0.11}$	$80.59_{0.18}$	$86.61_{0.30}$
	GPTQ sliced	$55.97_{0.28}$	$79.97_{0.19}$	$74.70_{0.05}$	$77.75_{0.40}$	$68.27_{0.27}$	$87.72_{0.17}$	$74.83_{0.08}$	$81.33_{0.74}$	$87.29_{0.52}$

Table 6: Evaluation results for MatGPTQ on asymmetric Llama3.1-8B-Instruct for different λ configurations, one triple applied to both the feedback and the scale-search channel. Weightings: $(x, y, z) \rightarrow (\lambda_3, \lambda_4, \lambda_8)$. IF-p and IF-i are the prompt- and instance-level strict IFEval accuracies; All weights four groups equally, the five common-sense tasks, MMLU, GSM8K and IFEval (↑). The mean over three calibration seeds is reported.

Bits	λ	ARC-c	ARC-e	HellaS	PIQA	WinoG	GSM8K	MMLU	IF-p	IF-i	All
8	(8, 1, 1)	54.27	79.17	77.31	79.71	73.01	79.30	67.28	69.50	78.54	73.32
	(4, 2, 1)	55.12	79.04	77.90	80.41	72.61	82.34	68.08	72.27	80.58	74.96
	(4, 4, 1)	54.27	80.18	78.19	80.69	73.32	83.85	68.28	74.12	80.82	75.73
	(1, 1, 1)	54.81	80.32	78.68	80.56	73.61	83.27	68.52	73.63	81.02	75.68
	(1, 2, 4)	54.93	79.83	78.87	81.02	74.11	84.69	69.07	74.23	81.73	76.37
	(1, 4, 4)	55.38	79.67	78.89	80.69	74.35	84.08	68.81	73.20	80.46	75.88
	(1, 1, 8)	55.12	79.76	79.05	80.85	73.95	85.60	69.36	75.79	82.49	76.96
4	(8, 1, 1)	54.01	79.25	76.55	79.98	72.77	77.86	66.55	66.91	76.14	72.11
	(4, 2, 1)	55.12	80.51	77.38	80.79	73.40	79.76	67.38	70.43	78.78	73.79
	(4, 4, 1)	53.84	79.50	77.62	80.52	73.88	80.29	67.92	70.98	79.14	74.09
	(1, 1, 1)	53.70	78.82	78.06	80.81	73.69	81.68	67.91	71.47	79.70	74.55
	(1, 2, 4)	53.96	78.65	78.42	80.76	73.62	82.84	68.33	73.42	81.03	75.37
	(1, 4, 4)	54.78	79.17	78.35	80.96	73.80	82.71	68.03	74.12	81.06	75.44
	(1, 1, 8)	54.78	79.38	78.00	81.45	74.11	81.43	68.05	72.46	80.70	74.90
3	(8, 1, 1)	52.05	77.40	75.76	79.54	72.06	71.34	63.87	67.28	76.74	69.65
	(4, 2, 1)	51.11	75.80	76.06	79.65	72.85	72.25	64.24	67.65	75.30	69.76
	(4, 4, 1)	51.62	77.69	75.81	79.33	71.51	71.87	63.59	68.58	77.10	69.87
	(1, 1, 1)	51.19	75.22	75.89	79.29	72.38	74.00	63.81	69.32	78.10	70.58
	(1, 2, 4)	50.24	75.88	75.26	78.82	72.04	68.28	62.96	69.54	77.70	68.83
	(1, 4, 4)	52.65	76.30	75.40	79.00	71.11	59.89	63.10	68.76	77.34	66.74
	(1, 1, 8)	48.98	76.81	75.19	78.67	72.45	69.07	62.69	67.28	76.14	68.47
6	(8, 1, 1)	54.86	79.67	77.38	79.87	72.69	79.00	67.02	70.61	79.02	73.43
	(4, 2, 1)	54.95	79.50	77.80	80.47	73.09	82.18	67.81	70.98	79.14	74.55
	(4, 4, 1)	54.69	79.84	78.18	80.96	74.03	83.40	68.29	71.72	79.74	75.24
	(1, 1, 1)	54.72	79.91	78.63	80.67	73.66	82.84	68.49	73.26	80.66	75.45
	(1, 2, 4)	54.88	79.60	78.82	81.01	74.14	84.82	68.93	73.94	81.61	76.30
	(1, 4, 4)	54.95	79.59	78.92	80.74	74.03	84.15	68.84	73.57	80.46	75.91
	(1, 1, 8)	55.03	79.46	78.90	81.18	73.72	84.69	69.04	73.38	80.70	76.10

D Detailed evaluations for the impact of loss weights λ_R

We compare a uniform weighting approach against configurations that highly weight either the lowest or the highest bit-width. Additionally, we compare increasing weights from low to high bit-widths and from high to low. Tables 6 and 7 present detailed evaluation results on the impact of the loss weights $(\lambda_3, \lambda_4, \lambda_8)$, for the asymmetric Llama3.1-8B-Instruct and Qwen3-8B models. For Llama, $(1, 2, 4)$ stays closest to the GPTQ baseline across all target bit-widths, with a worst-case gap of only 0.25 points. For Qwen, $(1, 1, 1)$ performs better at 3 bits but loses performance at the higher bit-widths, however averaged over all target bit-widths, the 3-bit improvement slightly outweighs these losses. However, since we aim for a model that is balanced across bit-widths, i.e., with a consistently small gap to GPTQ at every precision, we select $(1, 2, 4)$ as the default configuration for both models.

Generally, we can summarize the λ_r ablation as follows: the weights do matter. Focusing more strongly on higher bit-widths can worsen the lower-bit models, and vice versa. Therefore, depending on the target use case, λ_r should be tuned according to which bit-widths are most important.

E Detailed Kernel Performance Results

In Figures 5, 6, and 7, we report the absolute latency and end-to-end throughput (tokens/s) for our kernel evaluations. These results complement the relative speedups reported in the main paper and provide a more detailed view of the kernel performance across bit-widths and batch sizes.

Table 7: Evaluation results for MatGPTQ on asymmetric Qwen3-8B for different λ configurations, one triple applied to both the feedback and the scale-search channel. Weightings: $(x, y, z) \rightarrow (\lambda_3, \lambda_4, \lambda_8)$. IF-p and IF-i are the prompt- and instance-level strict IFEval accuracies; All weights four groups equally, the five common-sense tasks, MMLU, GSM8K and IFEval (↑). The mean over three calibration seeds is reported.

Bits	λ	ARC-c	ARC-e	HellaS	PIQA	WinoG	GSM8K	MMLU	IF-p	IF-i	All
8	(8, 1, 1)	56.83	79.12	72.81	77.26	68.51	86.66	72.94	80.41	86.21	78.45
	(4, 2, 1)	54.44	78.70	73.45	77.48	69.93	86.96	73.65	78.19	84.77	78.22
	(4, 4, 1)	56.46	80.06	73.66	77.71	69.17	86.53	74.14	80.47	86.41	78.88
	(1, 1, 1)	56.57	79.86	74.11	77.76	68.82	86.75	74.58	80.55	86.57	79.08
	(1, 2, 4)	56.03	80.13	74.68	77.53	68.14	87.36	74.73	81.76	87.53	79.51
	(1, 4, 4)	55.80	79.84	74.58	77.58	68.59	86.96	74.54	82.26	87.77	79.45
	(1, 1, 8)	56.23	80.43	74.81	77.75	67.64	87.41	75.15	81.15	87.53	79.57
4	(8, 1, 1)	54.18	77.82	72.18	76.44	69.06	84.91	71.96	80.22	85.85	77.46
	(4, 2, 1)	54.69	79.50	72.54	76.99	68.67	87.04	73.23	78.74	84.89	78.14
	(4, 4, 1)	55.97	79.29	73.17	76.70	68.30	86.13	73.28	80.71	86.13	78.38
	(1, 1, 1)	55.99	79.72	73.38	76.87	69.03	86.35	73.58	80.81	86.40	78.63
	(1, 2, 4)	55.97	80.18	73.93	77.69	68.64	87.14	73.64	81.33	86.85	79.04
	(1, 4, 4)	56.40	80.09	73.88	77.80	68.35	86.96	73.75	82.44	87.77	79.28
	(1, 1, 8)	55.46	80.01	73.71	77.91	68.27	86.28	73.10	82.26	87.65	78.85
3	(8, 1, 1)	53.41	77.02	72.28	76.28	67.25	83.62	71.34	76.52	83.45	76.05
	(4, 2, 1)	53.84	79.67	72.62	77.26	67.25	82.56	70.60	79.67	85.61	76.48
	(4, 4, 1)	53.07	78.17	72.49	76.82	67.64	82.79	71.00	75.54	82.97	75.67
	(1, 1, 1)	53.52	79.72	72.06	76.99	67.23	82.74	70.80	77.01	83.67	75.95
	(1, 2, 4)	51.48	76.21	70.93	76.44	66.98	80.46	69.08	76.09	82.77	74.35
	(1, 4, 4)	49.66	73.02	69.87	75.68	66.54	81.20	69.19	73.94	81.53	73.77
	(1, 1, 8)	50.68	75.21	70.60	77.58	65.67	78.17	68.52	75.42	82.85	73.44
6	(8, 1, 1)	56.83	78.87	73.00	77.37	68.67	86.13	72.76	79.67	85.61	78.12
	(4, 2, 1)	54.69	79.67	73.45	77.86	69.69	87.04	73.77	78.93	85.01	78.46
	(4, 4, 1)	56.29	80.20	73.58	77.78	69.06	86.58	73.92	80.16	86.13	78.76
	(1, 1, 1)	56.30	79.98	74.11	77.92	68.78	86.76	74.44	79.82	86.14	78.90
	(1, 2, 4)	56.43	80.01	74.73	77.57	68.59	87.34	74.76	80.59	86.61	79.29
	(1, 4, 4)	55.55	79.46	74.60	77.75	68.19	87.04	74.26	80.96	87.41	79.15
	(1, 1, 8)	56.57	80.47	74.74	77.75	68.35	88.32	75.02	80.96	87.05	79.73

Additionally, we measured end-to-end speedups for longer prefill and decode workloads using our Transformer's integrations. As shown in Figure 8, MatGPTQ maintains its performance advantage. This can be attributed to our kernel matching baseline cuBLAS performance for longer prompt sequences, thereby at least matching AnyBCQ's dequantization+cuBLAS approach during prefill while providing faster decode performance. Overall, our system therefore provides better end-to-end system integration.

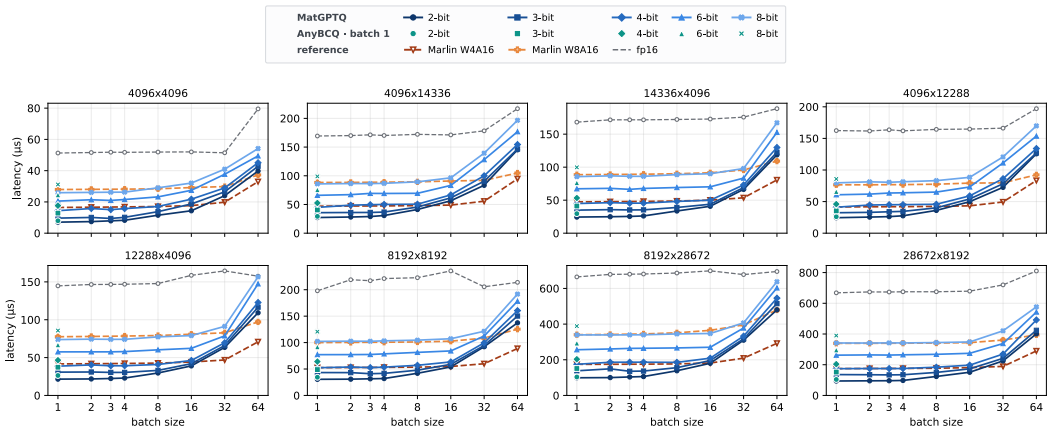


Figure 5: The figure shows the kernel latency compared to FP16 `torch.matmul` (cuBLAS). The kernels are evaluated at 2–8 bit precision for several representative matrix shapes across different batch sizes. AnyBCQ and Marlin W4A16/W8A16 are included as reference baselines. However, AnyBCQ results are only shown for batch size 1.

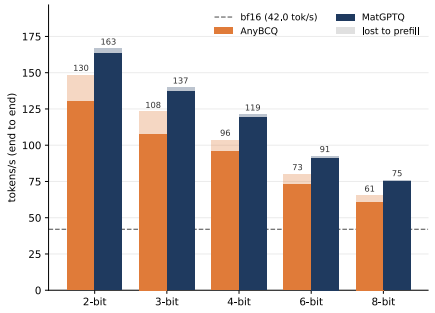


Figure 6: End-to-end throughput: Transformers – Llama3.1-8B-Instruct, Prompt/Decode 32/128.

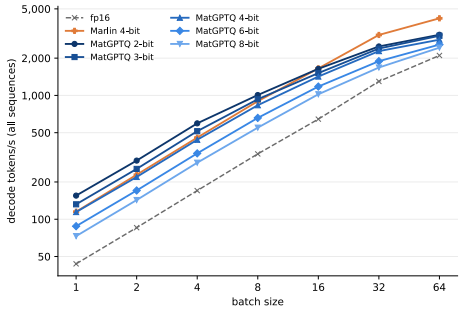


Figure 7: End-to-end throughput: vLLM – Llama3.1-8B-Instruct, Prompt/Decode 32/128.

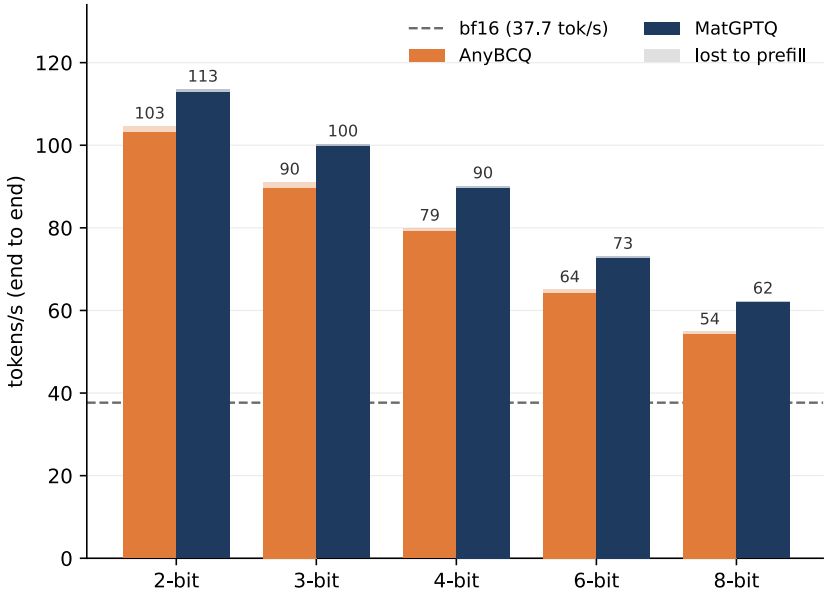


Figure 8: End-to-end throughput: Transformers – Llama3.1-8B-Instruct, Prompt/Decode 256/1024.