QServe: W4A8KV4 Quantization and System Co-design for Efficient LLM Serving

Yujun Lin*,1, Haotian Tang*,1, Shang Yang*,1, Zhekai Zhang[1], Guangxuan Xiao[1], Chuang Gan[3,4], Song Han[1,2]
MIT[1], NVIDIA[2], UMass Amherst[3], MIT-IBM Watson AI Lab[4]
{yujunlin,kentang,shangy,songhan}@mit.edu
https://hanlab.mit.edu/projects/qserve

Abstract*—Quantization can accelerate large language model (LLM) inference. Going beyond INT8 quantization, the research community is actively exploring even lower precision, such as INT4. Nonetheless, state-of-the-art INT4 quantization techniques only accelerate low-batch, edge LLM inference, failing to deliver performance gains in large-batch, cloud-based LLM serving. We uncover a critical issue: existing INT4 quantization methods suffer from significant runtime overhead (20-90%) when dequantizing either weights or partial sums on GPUs. To address this challenge, we introduce QoQ, a W4A8KV4 quantization algorithm with 4-bit weight, 8-bit activation, and 4-bit KV cache. QoQ stands for *quattuor-octō-quattuor*, which represents 4-8-4 in Latin. QoQ is implemented by the QServe inference library that achieves measured speedup. The key insight driving QServe is that the efficiency of LLM serving on GPUs is critically influenced by operations on low-throughput CUDA cores. Building upon this insight, in QoQ algorithm, we introduce *progressive quantization* that can allow low dequantization overhead in W4A8 GEMM. Additionally, we develop *SmoothAttention* to effectively mitigate the accuracy degradation incurred by 4-bit KV quantization. In the QServe system, we perform *compute-aware weight reordering* and take advantage of *register-level parallelism* to reduce dequantization latency. We also make fused attention memory-bound, harnessing the performance gain brought by KV4 quantization. As a result, QServe improves the maximum achievable serving throughput of Llama-3-8B by 1.2× on A100, 1.4× on L40S; and Qwen1.5-72B by 2.4× on A100, 3.5× on L40S, compared to TensorRT-LLM. Remarkably, QServe on L40S GPU can achieve even higher throughput than TensorRT-LLM on A100. Thus, QServe effectively reduces the dollar cost of LLM serving by 3×. Code is released at https://github.com/mit-han-lab/qserve.

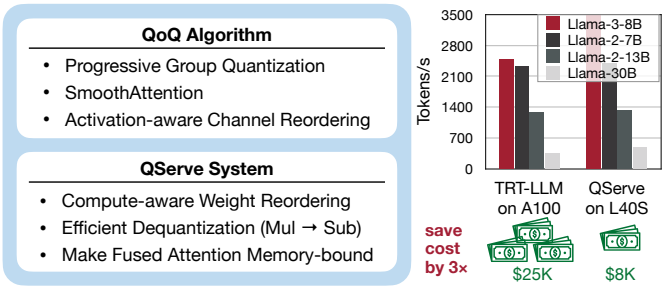

Fig. 1: QServe achieves higher throughput when running Llama models on L40S compared with TensorRT-LLM on A100, effectively saves the dollar cost for LLM serving by **3**× through system-algorithm codesign. See Table IV for absolute throughput numbers and precision choices in TensorRT-LLM.

I. Introduction

Large language models (LLMs) have demonstrated remarkable capability across a broad spectrum of tasks, exerting a profound influence on our daily lives.

However, the colossal size of LLMs makes their deployment extremely challenging, necessitating the adoption of quantization techniques for efficient inference. State-of-the-art integer quantization algorithms can be divided into three categories: 8-bit weight and 8-bit activation (**W8A8**), 4-bit weight and 16-bit activation (**W4A16**), 4-bit weight 4-bit activation (**W4A4**) quantization. The former two methods are considered nearly lossless in terms of accuracy. In contrast, **W4A4** quantization introduces a notable accuracy degradation, although it is anticipated to offer superior throughput in return by mapping its computations onto high-throughput 4-bit tensor cores. Unfortunately, this anticipated performance boost has not been consistently observed across current GPU platforms. For instance, the state-of-the-art **W4A4** serving system, Atom [44], exhibits 20-25% lower performance than its **W4A16** and **W8A8** counterpart in TensorRT-LLM when running the Llama-2-7B [34] model on A100 GPUs. That said, the research community has yet to find a precision combination superior to **W4A16** and **W8A8** for efficient cloud LLM serving.

In this paper, we reveal a critical observation: current 4-bit integer quantization methods experience significant overhead, ranging from 20% to 90%, during the dequantization of weights or partial sums on current-generation GPUs. For example, **W4A16** quantization performs computation on FP16 tensor cores while the weights are in INT4, so weight dequantization is required in the GEMM kernel. On the other hand, for **W4A4** quantization, to achieve reasonable accuracy, **W4A4** methods must apply **per-group** quantization to both weights and activation, sharing FP16 scaling factors on a sub-channel basis. For example, the state-of-the-art **W4A4** quantization method, QuaRot [2], reports a significant 0.2 perplexity degradation after switching from per-group quantization to per-channel quantization. This per-group quantization design requires an integer to floating-point dequantization for partial sums (since INT4 tensor cores produce INT32 partial sums), which operates on the slower CUDA cores within the sequential main loop of **W4A4** GEMM. On data center GPUs

*: The first three authors contribute equally to this project and are listed in the alphabetical order. Yujun Lin leads the quantization algorithm, Haotian Tang and Shang Yang lead the GPU kernels and the serving system.

like A100, a CUDA core operation is as expensive as **50** INT4 tensor core operations.

Therefore, reducing overhead on CUDA cores is crucial for achieving optimal throughput in LLM serving. Guided by this principle, we introduce QoQ (Quattuor-Octō-Quattuor, or 4-8-4 in Latin) algorithm which quantizes LLMs to **W4A8KV4** precision: 4-bit weights, 8-bit activations and 4-bit KV caches. Additionally, we present QServe, which provides efficient system support for **W4A8KV4** quantization.

In the QoQ algorithm, we introduce *progressive group quantization*. This method first quantizes weights to 8 bits using per-channel FP16 scales, then quantizes these 8-bit intermediates to 4 bits. This approach ensures that all GEMMs are performed on INT8 tensor cores. Additionally, we mitigate accuracy loss from KV4 quantization through *SmoothAttention*, which shifts the challenge of activation quantization from keys to queries, the latter of which are not quantized.

In the QServe system, the *protective range* in progressive group quantization enables full *register-level parallelism* during INT4 to INT8 dequantization, using a *subtraction after multiplication* computation order. Furthermore, we propose *compute-aware weight reordering* to minimize pointer arithmetic overhead on CUDA cores during W4A8 GEMM operations. Additionally, we delay the turning point of the CUDA core roofline and decrease the computational intensity of KV4 attention at the same time. This ensures that the attention operator remains within the memory-bound region, where low-bit quantization can effectively enhance throughput.

We evaluate seven widely-used LLMs using QServe on A100 and L40S GPUs, and compare their maximum achievable throughput against state-of-the-art systems, including TensorRT-LLM (in FP16, **W8A8**, and **W4A16** configurations), Atom [44] (in **W4A4**), and QuaRot [2] (in **W4A4**). On A100 GPUs, QServe achieves **1.2-2.4**× higher throughput over the best-performing configuration of TensorRT-LLM, and **2.5-2.9**× higher throughput compared to Atom and QuaRot. On L40S GPUs, QServe records an even more significant **1.5-3.5**× throughput improvement over TensorRT-LLM. Notably, we manage to accommodate the same batch size on the L40S while consistently achieving higher serving throughput than TensorRT-LLM on A100 for six of the eight models tested, thereby significantly reducing LLM serving cost by **3**×.

II. Background

A. Large Language Models

Large Language Models (LLMs) are a family of causal transformer models with multiple identically-structured layers. Each layer combines an attention block, a feed-forward network (FFN) and normalization layers. The input of each layer, $\mathbf{x}$, is an $N \times HD$ tensor, where N is the number of input tokens, H represents the number of attention heads, and D is the hidden dimension for each head. Serving LLMs involves two stages: the prefilling stage, where all prompt tokens are presented simultaneously ($N > 1$ for each request), and the decoding stage, where the model only processes one token at a time for each prompt ($N = 1$ for each request).

In attention blocks, $\mathbf{x}$ first undergoes linear projection to obtain $\mathbf{q} \in \mathbb{R}^{N \times HD}, \mathbf{k}, \mathbf{v} \in \mathbb{R}^{N \times H_{KV} D}$, where H_{KV} is the number of key/value heads. We have $H = H_{KV}$ in the standard multi-head attention (MHA), while recent methods [17], [18], [34] also employ grouped-query attention (GQA) [1] with $H = rH_{KV} (r \in \mathbb{Z})$. We concatenate $\mathbf{k}, \mathbf{v}$ with pre-computed *KV cache* features of S previous tokens to obtain $\mathbf{K}, \mathbf{V} \in \mathbb{R}^{(S+N) \times H_{KV} D}$ and compute attention using:

$$\mathbf{o}_h = \text{softmax}\left(\frac{\mathbf{q}_h \mathbf{K}_{h_{KV}}^T}{\sqrt{D}}\right) \mathbf{V}_{h_{KV}}, \quad h_{KV} = \left\lfloor \frac{h}{r} \right\rfloor . \tag{1}$$

The result $\mathbf{o}$ is multiplied with an output projection matrix $\mathbf{W}_O \in \mathbb{R}^{HD \times HD}$, and the product is added to $\mathbf{x}$ as the input of FFN. The FFN is composed of linear projection and activation layers and it does not mix features between tokens.

B. Integer Quantization

Integer quantization maps high-precision numbers to discrete levels. The process can be formulated as:

$$\mathbf{Q}_\mathbf{X} = \left\lceil \frac{\mathbf{X}}{s} + z \right\rfloor, s = \frac{\mathbf{X}_{\max} - \mathbf{X}_{\min}}{q_{\max} - q_{\min}}, z = \left\lceil q_{\min} - \frac{\mathbf{X}_{\min}}{s} \right\rfloor, \tag{2}$$

where $\mathbf{X}$ is the floating point tensor, $\mathbf{Q}_\mathbf{X}$ is its n-bit quantized counterpart, s is the scaling factor and z is the zero point. Thus, the dequantized tensor can be represented as,

$$\hat{\mathbf{X}} = Q(\mathbf{X}) = (\mathbf{Q}_\mathbf{X} - z) \cdot s \tag{3}$$

This is known as *asymmetric* quantization, where $\mathbf{X}_{\max} = \max(\mathbf{X})$, $\mathbf{X}_{\min} = \min(\mathbf{X})$, and $q_{\max} - q_{\min} = 2^n - 1$ for integer quantization. Equation 2 can be further simplied to *symmetric* quantization, where $z = \mathbf{0}$, $\mathbf{X}_{\max} = -\mathbf{X}_{\min} = \max|\mathbf{X}|$, and $q_{\max} - q_{\min} = 2^n - 2$.

In this paper, we denote x-bit weight, y-bit activation and z-bit KV cache quantization in LLMs as **WxAyKVz**, and use the abbreviated notation **WxAy** if y=z. Apart from bit precision, quantization can also be applied at various granularities. *Per-tensor* quantization shares s and z across the entire tensor. *Per-channel* quantization for weights or *per-token* quantization for activations means that s and z are shared within each *row* of tensor. *Per-group* quantization further reduces the degree of parameter sharing by using different s and z for every g columns within each row, where g is the group size.

III. Motivation

Weight and KV cache quantization (*e.g.* **W4**, **KV4**) can reduce the memory footprint in LLM serving. Quantizing both weight and activation (*e.g.* **W8A8**) can also improve the peak computation throughput. Choosing the right precision for LLM deployment is a difficult task. Existing solutions can be divided into three categories: **W4A16** (per-group), **W8A8** (per-channel weight + per-token activation), **W4A4** (per-group). We will demonstrate in this section why **W4A8KV4** is a superior choice.

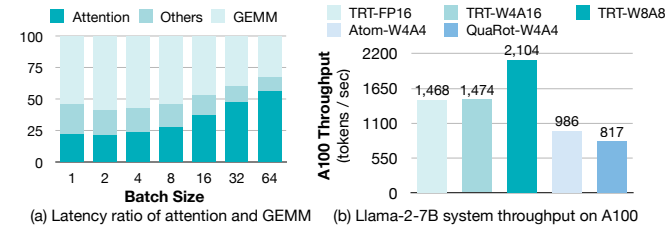

Fig. 2: **Left**: Both attention and GEMM are crucial for end-to-end LLM latency. **Right**: Despite 2× higher theoretical peak performance, **W4A4** systems significantly lag behind TRT-LLM-**W8A8** in efficiency.

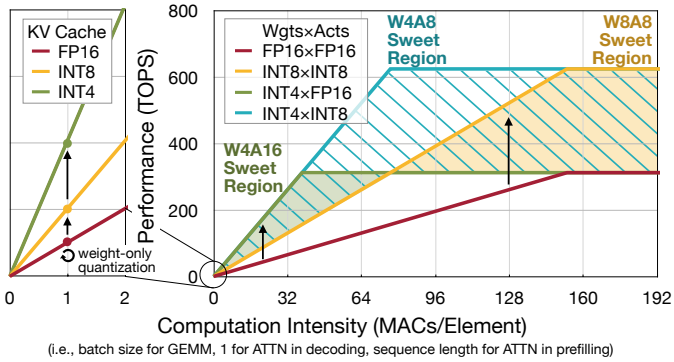

Fig. 3: **A100 roofline for LLM serving**: for GEMM layers, the **W4A8** roofline dominates both **W4A16** and **W8A8** across different batch sizes; for attention layers, 4-bit quantization improves theoretical peak performance.

A. *W4A8KV4 Has Superior Roofline Over W8A8, W4A16*

We begin our exploration through roofline analysis. As in Figure 2a, when considering real-world conversations with 1024 input tokens and 512 output tokens, attention and GEMM account for most of the runtime when deploying LLMs. Furthermore, the runtime of the decoding stage is approximately 6× that of the prefilling stage. Therefore, we focus our analysis on the attention and GEMM within the decoding stage.

For an $m \times n \times k$ GEMM problem, the computation intensity (defined as MACs/element) is approximately m when n, k are much larger than m. This situation applies to LLM decoding stage, since m is number of sequences and n, k are channel sizes. According to the A100 roofline[1] in Figure 3, **W4A16** has a higher theoretical throughput when $m < 78$, while **W8A8** performs better when $m > 78$. When the input batch size is small, GEMMs in LLMs are memory bound, and the memory bandwidth is dominated by weight traffic. Therefore, the smaller memory footprint of **W4A16** leads to better performance. However, when m is large, the problem is compute bound. Thus, **W8A8** has faster speed thanks to the higher throughput from INT8 tensor cores. Intuitively, one can expect **W4A8** to combine the best of both worlds across all batch sizes. This is clearly demonstrated in Figure 3, as long as we can perform all computation on INT8 tensor cores.

[1]A100 has a peak FP16/INT8/INT4 tensor core performance of 312/624/1248 TOPS and a DRAM bandwidth of 2 TB/s.

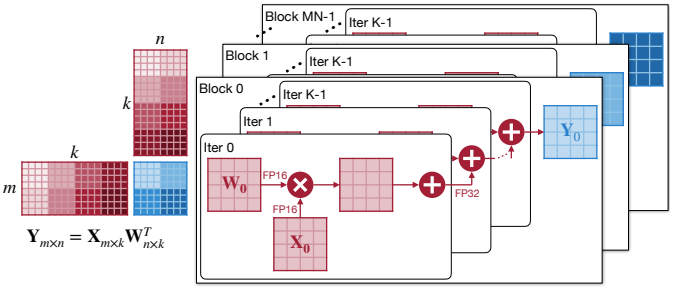

Fig. 4: **Illustration of $m \times n \times k$ GPU GEMM**: m, n are parallel dimensions and the reduction dimension k has a sequential main loop. In LLM serving, m is small and n, k are large. Thus, the main loop is long.

Why **KV4**: attention workloads in LLM decoding can be formulated as a sequence of batched GEMV operations, with a computation intensity of 1 MAC / element regardless of input batch sizes. As in Equation 1, the memory traffic is dominated by KV cache access, since $S \gg N = 1$ for each sequence. Quantizing the KV cache can be viewed as effectively increasing the memory bandwidth. Therefore, **KV4** offers 2× peak performance for attention over **KV8**. This improvement offers decent end-to-end speedup opportunities, since attention accounts for more than 50% of total runtime at batch=64 in Figure 2a.

B. *Why Not W4A4KV4: Main Loop Overhead in GEMM*

A natural follow-up question would be: "Why do we not choose the even more aggressive **W4A4**?" **W4A4** starts to achieve better theoretical GEMM performance when m, the number of input sequences, exceeds 78, as 4-bit tensor cores are twice as performant compared to their 8-bit counterparts. However, apart from the significant accuracy degradation, which will be discussed in Section VI, we demonstrate that such theoretical performance gains cannot be realized on existing GPU architectures (Ampere and Hopper). As in Figure 2b, existing **W4A4** serving systems Atom [44] and QuaRot [2] are even significantly slower than the **W16A16** solution from TensorRT-LLM.

While this performance gap can be partially explained by the inefficient runtime in these two systems, the inherent difficulty in mapping *per-group quantized* ***W4A4*** *GEMM* on GPUs has been overlooked in previous literature. State-of-the-art systems implement tensor core GEMM with an output stationary dataflow shown in Figure 4. For an $m \times n \times k$ problem, each thread block computes a $t_m \times t_n$ output tile by iterating sequentially through the reduction dimension k. This sequential loop is referred to as the main loop. The main loop comprises more than 100 iterations and dominates the runtime of the GEMM kernel. In both **FP16** and **W8A8** GEMM (Figure 5a), the main loop is executed entirely on tensor cores. TensorRT-LLM-**W4A16** (Figure 5b) and Atom-**W4A4** (Figure 5c) both require dequantization operations in the main

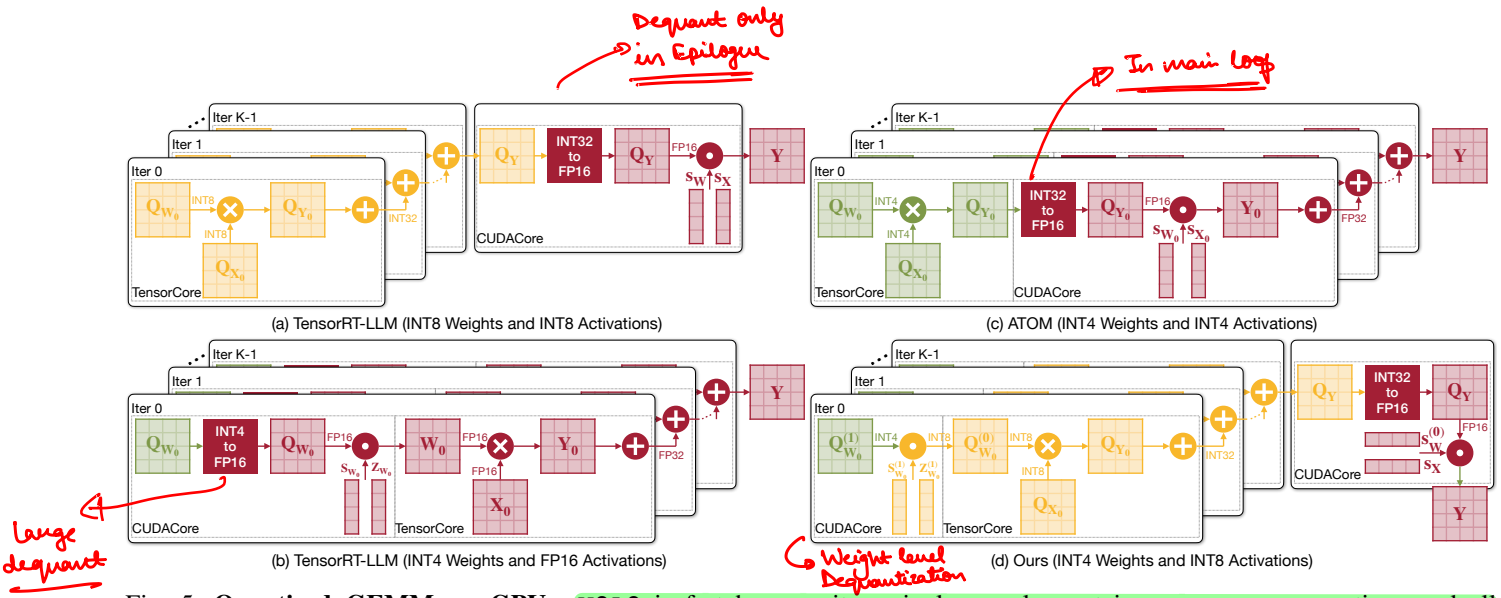

Fig. 5: **Quantized GEMM on GPUs:** **W8A8** is fast because its main loop only contains *tensor core* operations and all dequantization operations are present in the epilogue. Atom-**W4A4** and TensorRT-LLM-**W4A16** suffer from significant partial sum or weight dequantization overhead in the main loop. Thanks to the two-level progressive quantiation algorithm, QServe-**W4A8** reduces main loop dequantization overhead by introducing register-level parallelism.

loop, which is running on the CUDA cores. **W4A16** requires INT4 to FP16 weight conversion, while Atom-**W4A4** requires INT32 to FP32 partial sum conversion and accumulation.

The dequantization process in Atom's main loop leads to two substantial efficiency bottlenecks. Firstly, on modern data center GPUs like the A100 and H100, the peak performance of FP32 CUDA cores is merely **2%** of their INT4 tensor core counterparts. That said, de-quantizing one single partial sum in Atom is equivalent to **50** tensor core MACs. Therefore, the main loop is dominated by slow CUDA core operations rather than fast tensor core operations. Secondly, Atom creates two sets of registers (one for FP32 and one for INT32) to hold partial sums. Larger GEMM problems (e.g., prefilling stage) are typically register-bound on GPUs due to the nature of the output stationary dataflow, which results in high register consumption for storing *partial sums*. Consuming a large number of registers within each warp limits the number of warps that can be executed simultaneously on the streaming multiprocessor. It is important to note that GPUs rely on low-cost context switching between a large number of in-flight warps to hide latency. Consequently, a smaller number of concurrently executed warps limits the opportunity for latency hiding, further exacerbating the main loop overhead.

We preview our QServe's **W4A8** per-group quantized GEMM kernel design in Figure 5d. We employ a two-level *progressive group quantization* approach to ensure that all computations are performed on INT8 tensor cores. We opt for weight dequantization over partial sum dequantization due to its lower register pressure. Furthermore, we apply 4-way *register-level parallelism* to decode four INT4 weights simultaneously, further reducing the main loop overhead.

IV. QoQ Quantization

To this end, we have discussed why W4A8KV4 is a superior quantization precision choice. Yet, preserving model accuracy with such low-bit quantization remains a significant challenge. To unleash the full potential of W4A8KV4 without compromising the efficacy of large language models, we propose QoQ algorithm featuring *progressive group quantization*, *SmoothAttention*, and various general quantization optimizations.

A. Progressive Group Quantization

To enhance the accuracy of low-bit quantization, *group quantization* is commonly utilized [12], [23], [44]. However, as outlined in Section III-B, the dequantization overhead in the system implementation can negate these accuracy improvements. To tackle this issue, we introduce progressive group quantization, as depicted in Figure 6.

Given the weight tensor $\mathbf{W} \in \mathbb{R}^{k \times n}$, we first apply *per-channel symmetric* INT8 quantization:

$$\hat{\mathbf{W}} = \mathbf{Q}_{\mathbf{W}_{s8}}^{(0)} \cdot \mathbf{s}_{fp16}^{(0)}, \tag{4}$$

where $\mathbf{Q}_{\mathbf{W}_{s8}}^{(0)} \in \mathbb{N}^{n \times k}$ is the *intermediate* 8-bit quantized weight tensor, and $\mathbf{s}_{fp16}^{(0)} \in \mathbb{R}^{n \times 1}$ is the channel-wise quantization scales. We then further employ *per-group asymmetric* INT4 quantization on the intermediate weight tensor:

$$\mathbf{Q}_{\mathbf{W}_{s8}}^{(0)} = (\mathbf{Q}_{\mathbf{W}_{u4}} - \mathbf{z}_{u4}) \cdot \mathbf{s}_{u8}^{(1)}, \tag{5}$$

where $\mathbf{Q}_{\mathbf{W}_{u4}} \in \mathbb{N}^{n \times k}$ is the unsigned 4-bit quantized weight tensor, $\mathbf{z}_{u4} \in \mathbb{N}^{n \times k/g}$ is the unsigned 4-bit group-wise quantization zero points, and $\mathbf{s}_{u8}^{(1)} \in \mathbb{N}^{n \times k/g}$ is the unsigned 8-bit group-wise quantization scales.

For **W4A8** GEMM computation, the 4-bit quantized weight tensor $\mathbf{Q}_{\mathbf{W}_{u4}}$ will be first dequantized into intermediate 8-bit quantized weight tensor $\mathbf{Q}_{\mathbf{W}_{s8}}^{(0)}$ following Equation 5, and then perform INT8 matrix multiplication as if it was **W8A8** per-channel quantization.

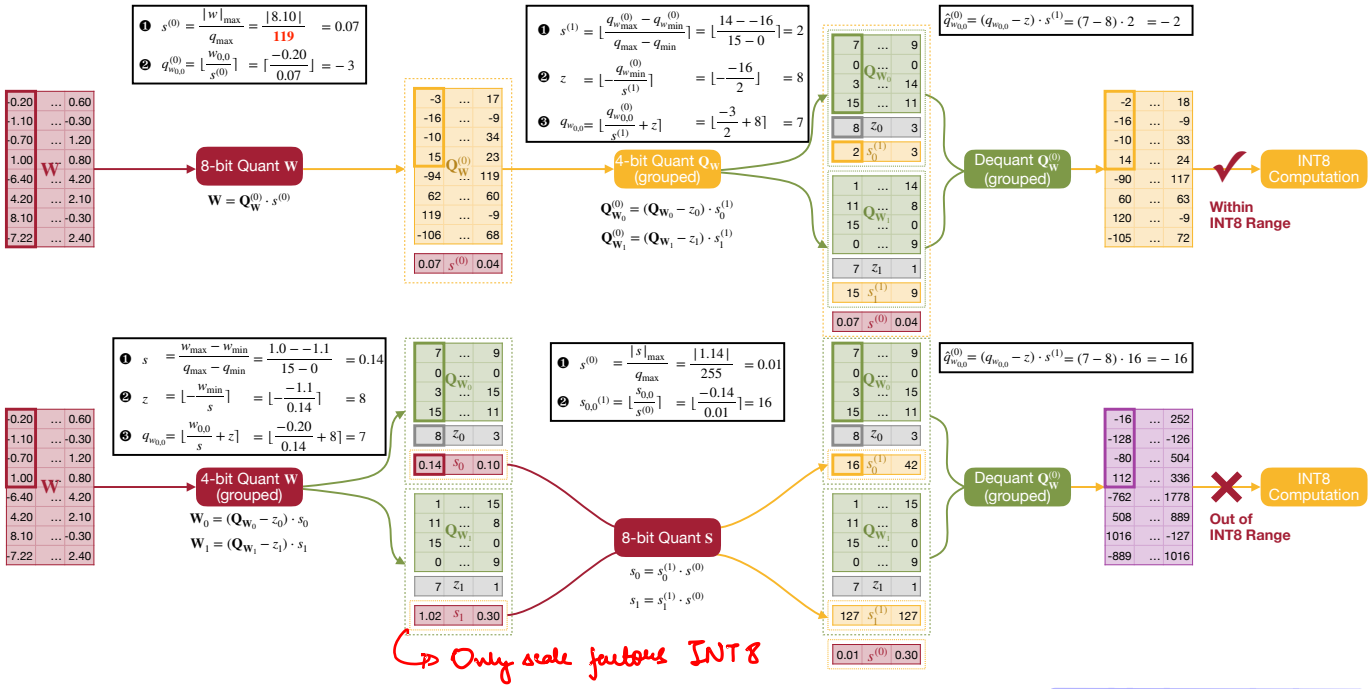

Fig. 6: **Progressive Group Quantization** first employs per-channel INT8 quantization with protective range [-119, 119], followed by per-group INT4 quantization, so that the dequantized intermediate values remain within the INT8 range for computation. **Bottom**: prior methods directly applies per-group INT4 quantization on weights, followed by per-channel INT8 quantization on scale factors. Thus the dequantized intermediate values may exceed the INT8 range, necessitating further dequantization to floating-point values for computation.

*a) **Protective Quantization Range**:* naïvely applying Equation 4 and 5 does not guarantee that the intermediate dequantized weights perfectly lie in the 8-bit integer representation range. For example, after INT8 quantization, a group of 8-bit weights lie in $[-113, 120]$. 4-bit asymmetric quantization will yield a scale factor of $\lceil (120 - -113)/(15 - 0) \rfloor = 16$ and a zero point of $\lceil 0 - -113/16 \rfloor = 7$. Thus value 120 is quantized into $\lceil 120/16 + 7 \rfloor = 15$. It will be dequantized into $(15 - 7) * 16 = 128$ which is beyond the max 8-bit integer 127. One straightforward solution is to turn on the saturation option in the arithmetic instructions during dequantization. However, simply applying saturation will severely damage the computation throughput, reducing speed by as much as 67%.

We reconsider the dequantization process. Take Equation 2 into Equation 5, we have,

$$\hat{q}_{s8} = \lfloor \frac{q_{s8}}{s_{u8}} \rceil \cdot s_{u8} \leq q_{s8} + \frac{1}{2} s_{u8}.$$

Since $s_{u8} = \frac{q_{s8\max} - q_{s8\min}}{q_{u4\max} - q_{u4\min}} \leq \frac{127-(-128)}{15-0} = 17$, we have,

$$\hat{q}_{s8} \leq 127 \rightarrow q_{s8} \leq 127 - \frac{1}{2} s_{u8} \rightarrow q_{s8} \leq 119.5$$

Therefore, we shrink the INT8 symmetric quantization range from [-127, 127] to a *protective range* [-119, 119] in order to avoid the dequantization overflow, as shown in the top of Figure 6.

*b) **Compared to previous two-level quantization**:* progressive group quantization introduces two levels of scales $\mathbf{s}^{(0)}_{\text{fp16}}$ and $\mathbf{s}^{(1)}_{\text{u8}}$. Prior studies such as VSQuant and DoubleQuant in QLoRA [9] also introduce two levels of scales to reduce the memory footprint of group-wise scaling factors. In contrast to our quantization flow, previous approaches directly apply group quantization with the target precision and then perform per-channel quantization on the group-wise floating-point scaling factors, as shown in the bottom of Figure 6:

$$\hat{\mathbf{W}} = \mathbf{Q}_{\mathbf{W}_{s4}} \cdot \mathbf{s}_{\text{fp16}}, \quad \hat{\mathbf{s}}_{\text{fp16}} = \mathbf{s}^{(1)}_{\text{u8}} \cdot \mathbf{s}^{(0)}_{\text{fp16}} \tag{6}$$

Therefore, using the group-wise scaling factors $\mathbf{s}^{(1)}_{\text{u8}}$ to dequantize $\mathbf{Q}_{\mathbf{W}_{s4}}$ cannot yield the 8-bit weight tensor. During the computation on GPUs, these approaches usually first dequantize the scales and, subsequently, the weights into floating-point values, which ultimately limits the peak throughput.

DGQ [43] also follows the quantization scheme of VSQuant and DoubleQuant, but enforces restrictions on scaling factors to make sure that all computation can be mapped onto INT8 tensor cores. However, the DGQ serving system separates dequantization kernel with the GEMM kernel. Consequently, the end-to-end latency of W4A8 GEMM in DGQ is even slower than the W8A8 GEMM in cuBLAS, failing to demonstrate the memory bandwidth advantage of 4-bit weight quantization. In contrast, our QoQ introduces a protective range, allowing us to fuse dequantization operations into the W4A8 GEMM kernel with full register-level parallelism, minimizing CUDA

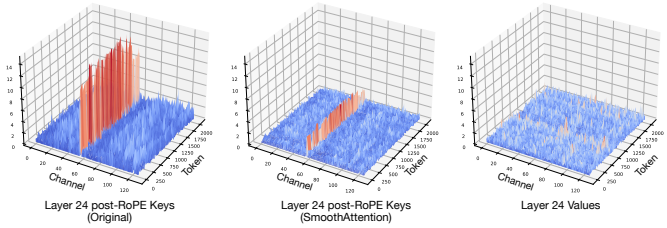

Fig. 7: **SmoothAttention effectively smooths the outliers in Keys. Values doesn't suffer from outliers.**

core overhead. Thus, our QServe's W4A8 per-group GEMM achieves 1.5× speedup over the W8A8 cuBLAS GEMM.

B. SmoothAttention

As illustrated in Figure 16, directly reducing the KV cache to 4 bits significantly degrades the LLM accuracy. We visualize the magnitude distributions of the sampled Key and Value cache activations in Figure 7. We observe that: **the Value matrices show no significant outlier pattern, whereas Key matrices tend to have fixed outlier channels in each head**. These outliers are ∼10× larger than most of activation values. Though they can be easily handled **KV8** quantization in prior works [38], it places challenging obstacle to **KV4** quantization due to less quantization levels.

Inspired by SmoothQuant [38], we propose SmoothAttention to scale down the outlier channels in Key cache by a per-channel factor λ:

$$\mathbf{Z} = (\mathbf{Q}\mathbf{\Lambda}) \cdot \left(\mathbf{K}\mathbf{\Lambda}^{-1}\right)^T, \quad \mathbf{\Lambda} = \mathrm{diag}\left(\lambda\right) \tag{7}$$

SmoothQuant migrates the quantization difficulty from activations to weights, and thus requires a dedicate balance between activation and weight quantization by searching the migration strength. In contrast, since we do not quantize Queries, we only need to concentrate on the Keys and simply choose the SmoothAttention scale factor as,

$$\lambda_i = \max\left(|\mathbf{K}_i|\right)^\alpha. \tag{8}$$

In practice, $\alpha = 0.5$ is good enough. As shown in Figure 7, after SmoothAttention, the outliers in Key cache have been greatly smoothed.

In order to eliminate the extra kernel call overhead for SmoothAttention scaling, fusing the scale into preceding linear layer's weights is preferred. However, modern LLMs employ the rotary positional embedding (RoPE) to both Keys and Queries, which needs extra handling. In practice, rotary positional embedding pairs channel i with channel $i + \frac{D}{2}$ within each head. Consequently, to make SmoothAttention scaling commutative in terms of RoPE, we add a hard constraint that $\lambda_i = \lambda_{i+\frac{D}{2}}$, and accordingly,

$$\lambda_i = \lambda_{i+\frac{D}{2}} = \max\left(\max\left(|\mathbf{K}_i|\right), \max\left(|\mathbf{K}_{i+\frac{D}{2}}|\right)\right)^\alpha \tag{9}$$

Afterwards, we can easily fuse the SmoothAttention scale $\mathbf{\Lambda}$ into previous layers' weights following $\mathbf{W}_Q = \mathbf{\Lambda}\mathbf{W}_Q$ and $\mathbf{W}_K = \mathbf{\Lambda}^{-1}\mathbf{W}_K$.

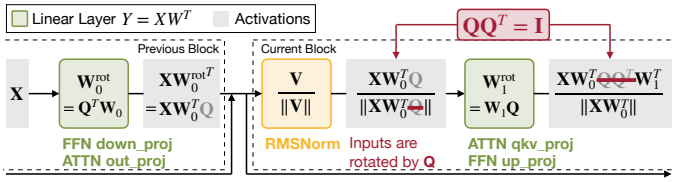

Fig. 8: **Rotate the block input activations to suppress the outliers**: since rotation is a unitary transformation, the rotation matrix $\mathbf{Q}$ can be absorbed by the weights of the output module in the previous block.

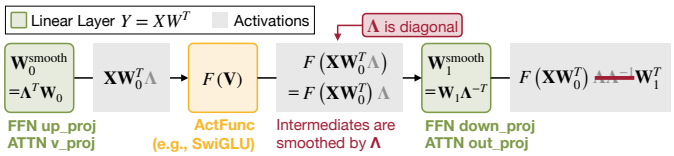

Fig. 9: **Smooth the block intermediate activations, migrating the quantization difficulty to weights**: since smoothing is channel-independent, the smooth matrix $\mathbf{\Lambda}$ is diagonal and can be absorbed by the weights of the previous modules.

C. General LLM Quantization Optimizations

One of the key challenges of low-bit LLM quantization is the activation outliers for every linear layers. We apply different optimizations for different types of linear layers as discussed below.

1) Block Input Module Rotation: In transformer blocks, we define the components that take in the block inputs as input modules, such as the QKV Projection Layer and the FFN 1st Layer. As shown in Figure 8, inspired by [2], [4], we rotate the block input activations by multiplying the rotation matrix. To keep mathematical equivalence of linear layers, we rotate the corresponding weights accordingly in the reversed direction. After rotation, each channel's activations are linear combinations of all other channels, and thus outlier channels are effectively suppressed. Furthermore, since rotation is a unitary transformation, we can fuse the rotation matrix with the previous linear layers' weights. We simply choose the scaled Hadamard matrix as the rotation matrix.

2) Block Output Module Smoothing: Output modules refer to those layers that generate block outputs, such as the Output Projection Layer and FFN 2nd Layer. As shown in Figure 9, inspired by [38], we smooth the block intermediate activations through dividing them by a per-channel smoothing factor. Original SmoothQuant does not smooth the block intermediate activations; moreover, if we directly smooth these modules with the same migration strength as input modules (*e.g.*, `q_proj`, `up_proj`), the evaluated Wikitext-2 perplexity of the Llama-2-7B model will drop by as much as 0.05. In practice, we find that the migration strength α should be near 0. That is, the smoothing factor λ is mostly determined by weights instead of activations, which is very different from the observations in SmoothQuant.

3) Activation-Aware Channel Reordering: Both AWQ [23] and Atom [44] have observed that maintaining the salient

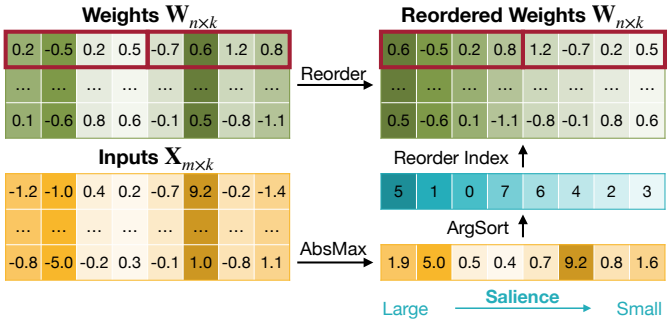

Fig. 10: Reorder weight input channels based on their salience in group quantization. Channel salience can be determined by the magnitude of input activations.

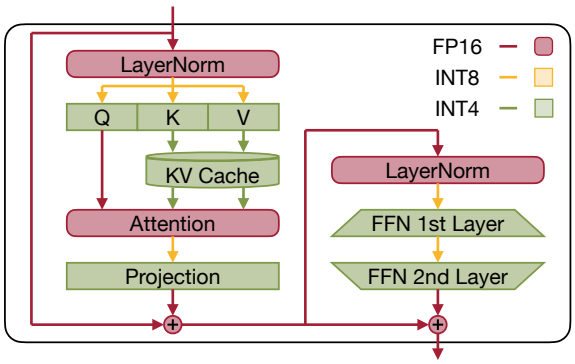

Fig. 11: **QServe's precision mapping** for an `FP16` in, `FP16` out LLM block. All GEMM operators take in **`W4A8`** inputs and produce `FP16` outputs. Activation quantization happens in normalization and activation layers.

weights in FP16 can significantly improve model accuracy. These salient weights can be identified by the activation distribution. Instead of introducing mixed-precision quantization used by Atom, we propose activation-aware channel reordering as shown in Figure 10. We use $\max(|\mathbf{X}|)$ to determine the channel salience, and then reorder channels so that channels with similar salience are in the same quantization group.

4) Weight Clipping: Weight clipping is another popular quantization optimization technique. It applies a clip ratio α to the dynamic range in Equation 2 by letting $\mathbf{W}_{\max} = \alpha \max(\mathbf{W})$ and $\mathbf{W}_{\min} = \alpha \min(\mathbf{W})$. Previous approaches [2], [12], [23], [44] grid search the clip ratio α to minimize either quantization error of tensor itself (*i.e.*, $\|\mathbf{W} - Q(\mathbf{W};\alpha)\|$) or output mean square error (*i.e.*, $\|\mathbf{X}\mathbf{W}^T - \mathbf{X}Q(\mathbf{W}^T;\alpha)\|$). In QServe, we minimize the layer output error for all linear layers, expect for `q_proj` and `k_proj`, for which we optimize block output mean square error:

$$\arg\min_{\alpha} \|\text{Block}(\mathbf{X};\mathbf{W}) - \text{Block}(\mathbf{X};Q(\mathbf{W};\alpha))\|. \quad (10)$$

V. QServe Serving System

To this end, we have presented the QoQ quantization algorithm, which aims to minimize accuracy loss incurred by **`W4A8KV4`** quantization. However, realizing the theoretical throughput benefits in Figure 3 remains challenging. Thus, in this section, we will delve into the QServe system design, which is guided by two important principles: **I.** Reducing main loop overhead in GEMM kernels; **II.** Making fused attention kernels memory bound.

A. QServe System Runtime

We start by introducing the QServe runtime in Figure 11. All GEMM layers in QServe operate on **`W4A8`** inputs, perform computation on `INT8` tensor cores, and generate `FP16` outputs. All attention layers perform computation in `FP16` on CUDA cores. Consequently, each LLM block in QServe has `FP16` inputs and `FP16` outputs.

Activation Quantization. To ensure that each GEMM takes in `INT8` activation, we fuse activation quantization into the preceding layernorm for the QKV projection and the first FFN layer, or into the preceding activation kernel for the second FFN layer. Furthermore, a separate quantization node is inserted before output projection in the attention block.

KV Cache Management. To avoid memory fragmentation, we follow vLLM [21] and TensorRT-LLM [25] to adopt paged KV caches. In contrast to these two frameworks, which perform *per-tensor*, *static* quantization (*i.e.*, scaling factors computed offline) on KV caches, QServe requires *per-head*, *dynamic* KV quantization to maintain competitive accuracy due to the lower bit precision (4 *vs.* 8). We therefore store `FP16` scaling factors and zero points for each head immediately following the quantized KV features in each KV cache page, allowing these values to be updated on-the-fly. QServe also supports in-flight batching, similar to vLLM and TensorRT-LLM.

B. `W4A8` GEMM in QServe

As discussed in Section III, the main loop overhead poses a significant obstacle in allowing quantized GEMMs to attain the theoretical performance gains projected by the roofline model (Figure 3). Therefore, the focus of QServe **`W4A8`** GEMM is to **reduce main loop overhead**. Specifically, we address the costs of pointer arithmetic operations through **compute-aware weight reorder**, and reduce dequantization overhead through a **subtraction after multiplication** computation order and **register-level parallelism**.

1) Compute-Aware Weight Reorder: Prior to dequantization and tensor core computation, the operands must be loaded from global memory into the L1 shared memory during each main loop iteration. This loading process is non-trivial since the tensor core GEMM intrisics require a strided layout for each thread in computation, as demonstrated in Figure 12a. For instance, instead of loading consecutive eight `INT8` weights, thread 0 first loads input channels 0-3, then skips ahead to input channels 16-19. That said, a naive weight loading implementation would require one address calculation per four channels, leading to two efficiency issues. First, pointer arithmetic operations are performed on CUDA cores, which have **32**× lower throughput than the `INT8` tensor core on the A100.

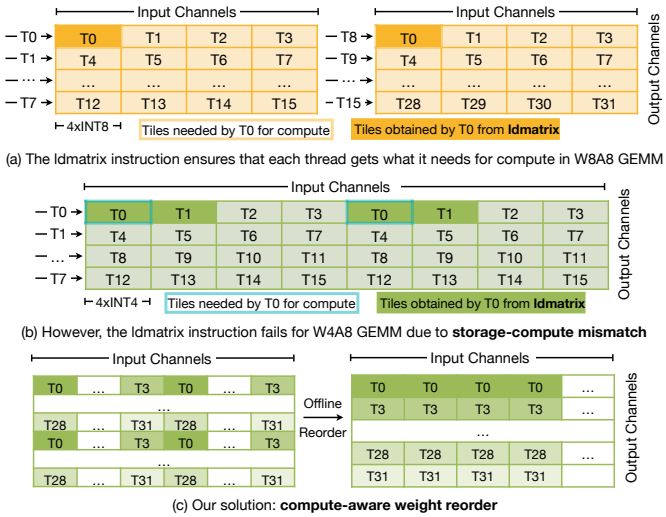

Fig. 12: QServe applies **compute-aware weight reoder** to minimize the pointer arithmetics in **W4A8** GEMM main loop.

Consequently, the address calculation overhead becomes non-negligible. Second, strided memory access prevents achieving the highest HBM bandwidth through packed 128-bit loading, further slowing down the memory pipeline. This issue is addressed by the `ldmatrix` instruction when the storage and compute data types are **the same**. As illustrated in Figure 12a, thread i loads a consecutive 128 bits in output channel $i\%8$, and the `ldmatrix` instruction automatically distributes the data in a strided manner, ensuring that each thread eventually obtains the required data for `INT8` tensor core computation.

Unfortunately, the `ldmatrix` instruction will **not** work when the data types used for storage and computation differ (like in **W4A8**). Specifically, in Figure 12b, `ldmatrix` ensures that each thread obtains the same number of **bytes**, not the same number of **elements**, after data permutation in the register file. Consequently, thread 0 obtains the tiles needed by both itself and thread 1, while thread 1 obtains the tiles needed by thread 2 and thread 3 in the subsequent `INT8` tensor core computation. This creates a mismatch between the data obtained by each thread and used in computation. That said, `ldmatrix` cannot be used for **W4A8** GEMM and the aforementioned pointer arithmetic overhead persists. Worse still, memory bandwidth utilization deteriorates further as we consecutively load only 16 bits for 4-bit weights.

We address this challenge through **compute-aware weight reordering** (Figure 12c). The key insight is to store the weights in the order they are used during computation. We divide the entire GEMM problem into multiple 32×32 tiles. Within each tile, thread 0 utilizes input channels 0-3 and 16-19 for output channels 0, 8, 16, and 24 (output channels 16-31 are omitted in Figure 12c). Consequently, we concatenate these 32 channels into a single 128-bit word. The 32 channels used by thread 1 are stored immediately following thread 0's 32 channels. Since weights are static, such reordering does not introduce any runtime overhead. Additionally, it not only

Fig. 13: QServe exploits **register-level parallelism** to significantly reduce the number of required logical operations in `UINT4` to `UINT8` weight unpacking.

reduces the pointer arithmetic overhead to the same level as `ldmatrix` but also guarantees high-bandwidth 128-bit/thread memory transactions. We apply this reordering to zero points and scales as well to mitigate dequantization overhead.

*2) Fast Dequantization in Per-Channel **W4A8** GEMM:* As illustrated in Figure 5d, dequantizing weights within the main loop becomes necessary when the bit precisions for weights and activations differ. In the case of per-channel **W4A8** quantization, second-level scaling factors are omitted, and first-level `FP16` scaling is efficiently fused into the GEMM epilogue. We therefore focus our discussion on the efficient conversion from `ZINT4` (i.e., unsigned 4-bit integers with zero points) to `SINT8` within the main loop. We further decompose this conversion into two steps: `UINT4` to `UINT8` (weight unpacking) and `UINT8` to `SINT8` (zero point subtraction). As depicted in Figure 13, we reorder every 32 `UINT4` weights $w_0, w_1, ..., w_{31}$ into $w_0, w_{16}, w_1, w_{17}, ...$ This allows us to exploit **register-level parallelism** and efficiently unpack them into `UINT8` numbers with only three logical operations.

For the conversion from `UINT8` to `SINT8`, the most intuitive approach is to introduce integer subtraction instructions within the main loop, which we refer to as subtraction before multiplication. Although straightforward, this approach inevitably introduces additional cost to the main loop, which is undesirable. Instead, we adopt a **subtraction after multiplication** approach to minimize the main loop overhead.

Specifically, a GEMM layer with per-channel quantized operands can be expressed as:

$$\mathbf{O} = \hat{\mathbf{X}}\hat{\mathbf{W}} = (\mathbf{Q_X} \odot \mathbf{S_X})((\mathbf{Q_W} - \mathbf{Z_W}) \odot \mathbf{S_W}), \tag{11}$$

where $\mathbf{Q_W}$ ($\mathbf{Q_X}$) is the quantized weight (activation), $\mathbf{Z_W}$ expands the zero point vector $\mathbf{z_W}$ of size n (output channels) to $k \times n$ (k is input channels) and $\mathbf{S_W}$, $\mathbf{S_X}$ are similarly obtained from scaling vectors $\mathbf{s_W}, \mathbf{s_X}$. We denote $\mathbf{Z_W} \odot \mathbf{S_W}$ as $\mathbf{ZS_W}$, then we rewrite Equation 11 as:

$$\begin{aligned}\mathbf{O} &= (\mathbf{Q_X} \odot \mathbf{S_X})(\mathbf{Q_W} \odot \mathbf{S_W} - \mathbf{ZS_W}) \\ &= (\mathbf{Q_X}\mathbf{Q_W}) \odot (\mathbf{s_W} \times \mathbf{s_X}) - (\mathbf{Q_X} \odot \mathbf{S_X})\mathbf{ZS_W}.\end{aligned} \tag{12}$$

The first term, $(\mathbf{Q_X}\mathbf{Q_W}) \odot (\mathbf{s_W} \times \mathbf{s_X})$, is analogous to the **W8A8** GEMM in TensorRT-LLM, where the $\mathbf{s_W} \times \mathbf{s_X}$ outer product scaling is performed in the epilogue. For the second term, we first replace $\mathbf{Q_X S_X}$ ($\hat{\mathbf{X}}$) with the unquantized $\mathbf{X}$. We then notice that:

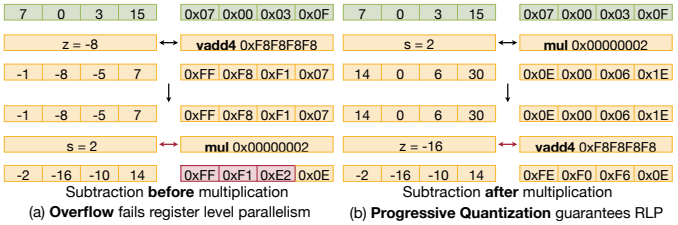

Fig. 14: Our progressive quantization algorithm ensures that all intermediate results in the **subtraction after multiplication** computation order will not overflow, thereby enabling **register-level parallelism** and reducing main loop overhead.

$$\mathbf{X}(\mathbf{Z}\mathbf{S}_\mathbf{W}) = \mathbf{t}_\mathbf{X} \times (\mathbf{z}_\mathbf{W} \odot \mathbf{s}_\mathbf{W}), \tag{13}$$

where $\mathbf{t}_\mathbf{X} = \mathbf{X}\mathbf{1}_k$, i.e., summing all input channels for each token. We observe that Equation 13 has a form similar to the outer product of scaling factors. Therefore, it can also be fused into the epilogue of the **W4A8** GEMM, analogous to the first term in Equation 12. To this end, we move the zero-point subtraction from the main loop to the epilogue, thereby largely eliminating its overhead in the GEMM kernel. This formulation of **subtraction after multiplication** necessitates precomputing $\mathbf{t}_\mathbf{X}$. Fortunately, each **W4A8** kernel is always preceded by a memory-bound kernel, allowing us to fuse the precomputation kernel into it with negligible latency overhead.

*3) Fast Dequantization in Per-Group **W4A8** GEMM:* The primary distinction between the per-group **W4A8** GEMM and its per-channel counterpart lies in the second-level dequantization process in Figure 5d. Firstly, since zero points are now defined on a per-group basis, it is no longer possible to merge zero point subtraction into the epilogue, as was done in the previous section. Secondly, due to the presence of level 2 scales, an additional INT8 multiplication is required for each weight. Akin to the previous section, we must determine whether to apply multiplication (scales) or subtraction (zeros) first during level 2 dequantization.

In this context, we contend that performing **subtraction after multiplication** remains the advantageous approach because it enables **register-level parallelism** (RLP). As shown in Figure 14, NVIDIA GPUs provide the vadd4 instruction that performs four INT8 additions with a single INT32 ALU operation. However, there is no instruction that realizes similar effect for 4-way INT8 multiplication. Consequently, in order to achieve RLP, one has to simulate this by padding 24 zeros to the most significant bits (MSBs) of the 8-bit scaling factor. However, this simulation is valid only when the result of each INT8 multiplication remains within the INT8 range. This condition is not met for the subtraction-before-multiplication computation order. As illustrated in Figure 14a, the result of the scale multiplication overflows, leading to an incorrect output. In the subtraction-before-multiplication approach, we can only perform multiplication one by one, which is extremely inefficient. On the other hand, with the subtraction-after-multiplication computation order, our progressive group quantization algorithm ensures that the result of the initial multiplication step never exceeds the INT8 range. This allows for fully leveraging the performance benefits of RLP in both multiplication and subtraction.

TABLE I: A naive KV4 attention implementation is 1.7× faster on L40S than TRT-LLM-KV8, but is 1.1-1.2× slower on A100 due to earlier CUDA core roofline turning point.

Seq_len	8-bit KV	4-bit KV (Naive)	4-bit KV (Ours)
128	0.09 ms	0.10 ms (0.87×)	0.07 ms (1.29×)
256	0.14 ms	0.16 ms (0.86×)	0.11 ms (1.32×)
512	0.23 ms	0.27 ms (0.87×)	0.16 ms (1.44×)
1024	0.42 ms	0.48 ms (0.88×)	0.28 ms (1.49×)
1536	0.62 ms	0.69 ms (0.90×)	0.41 ms (1.51×)

4) General Optimizations: In our **W4A8** kernel, we also employ general techniques for GEMM optimization. On the memory side, we apply multi-stage software pipelining and asynchronous memory copy to better overlap memory access with computation. Additionally, we swizzle the layout of the L1 shared memory to eliminate bank conflicts. To improve L2 cache utilization, we permute the computation partition across different thread blocks, allowing adjacent blocks to reuse the same weight. On the compute side, when the number of input tokens (m) is small, we found it beneficial to partition the reduction dimension k into multiple slices and reduce the partial sums across different warps in the L1 shared memory.

Split-K

C. *KV4 Attention in QServe*

Attention accounts for 30-50% of the total LLM runtime, as depicted in Figure 2a. Although the roofline model in Figure 5 suggests that quantizing the KV cache to INT4 should automatically yield a 2× speedup over the 8-bit KV baseline, this is not the case in real-world implementation.

We start with the **KV8**-attention decoding stage kernel from TensorRT-LLM as our baseline and replace all static, per-tensor quantized 8-bit KV cache accesses and conversions with their dynamic, per-head quantized 4-bit counterparts. This direct replacement immediately leads to 1.7× speedup on L40S, but results in **1.2× slowdown** on A100 (Table I), compared to the **KV8** baseline.

Once again, our analysis reveals that the devil is in the slow CUDA cores, which are responsible for executing the attention kernels during the decoding stage. While each individual batched GEMV has a computation intensity of 1 MAC / element, the computation intensity escalates significantly for a fused attention kernel that combines all the arithmetics and KV cache updates. As an illustration, naively dequantizing a single INT4 number from the KV cache necessitates 5 ALU Ops. This includes mask and shift operations to isolate the operand, type conversion from integer to floating-point representation, and floating point mul and sub to obtain the final results. It is crucial to note that the roofline turning point for A100 FP32 CUDA cores is merely **9.8 Ops/Byte**. That said, the dequantization of KV operands alone already saturates this bound, leading to the surprising observation that the fused **KV4**

attention kernel can become **compute-bound** on datacenter GPUs like A100. In fact, similar observations hold in other systems like QuaRot [2] and Atom [44]. Specifically, QuaRot introduces compute-intensive Hadamard transformation [4] in the attention operator, making it hard to achieve real speedup over TRT-LLM-KV8 with 4-bit quantized KV caches.

To mitigate the compute-bound bottleneck, it is important to shift the decoding stage **KV4** attention kernels away from the compute-bound region. We accomplish this objective through a bidirectional approach: Firstly, delaying the onset of the roofline turning point, and secondly, concurrently reducing the computation intensity within the fused kernel. For the first part, we replace all FP32 operations in the original TensorRT-LLM kernel with their FP16 counterpart, effectively doubling the computation roof. For the second part, we observe that the arithmetic intensity of dequantization can be significantly reduced to 2 operations per element by applying bit tricks proposed in [20]. Furthermore, we note that simplifying the control logic and prefetching the scaling factors and zero values, thereby simplifying address calculations, contribute to performance improvements. After incorporating these enhancements, we observe a **1.5**× speedup over TensorRT-LLM's **KV8** kernel on A100.

VI. Evaluation

A. Evaluation Setup

a) ***Algorithm:*** The QoQ quantization algorithm is implemented using HuggingFace [37] on top of PyTorch [26]. We use per-channel symmetric INT8 quantization on activations, and per-token asymmetric INT4 group quantization on KV cache. "**W4A8KV4** g128" refers to the case where QServe used progressive group quantization on weights: per-channel symmetric INT8 quantization followed by asymmetric INT4 quantization with a group size of 128, while "**W4A8KV4**" is the per-channel counterpart for weight quantization.

b) ***System:*** QServe serving system is implemented using CUDA and PTX assembly for high-performance GPU kernels. We also provide a purely PyTorch-based front-end framework for better flexibility. For the throughput benchmarking, we perform all experiments under PyTorch 2.2.0 with CUDA 12.2, unless otherwise specified. The throughput numbers reported are real measurements on NVIDIA GPUs. For baseline systems, we use TensorRT-LLM v0.9.0 and latest main branches from QuaRot and Atom as of April 18th, 2024. Paged attention is enabled for all systems except QuaRot, which does not offer corresponding support.

B. Accuracy Evaluation

a) Benchmarks: We evaluated QoQ on the Llama-1 [33], Llama-2 [34], Llama-3 families, Mistral-7B [17], Mixtral-8x7B [18] and Yi-34B [39] models. Following previous literature [2], [8], [12], [23], [38], [44], we evaluated QoQ-quantized models on language modeling and zero-shot tasks. Specifically, we evaluated on WikiText2 [24] for perplexity, and evaluated on PIQA [3] (PQ), ARC [5], HellaSwag [42] (HS) and WinoGrande [29] (WG) with `lm_eval` [13].

b) Baselines: We compared QoQ to widely used post-training LLM quantization techiniques, SmoothQuant [38], GPTQ [12], AWQ [23], and recently released state-of-the-art 4-bit weight-activation quantization frameworks, Atom [44] and QuaRot [2]. For SmoothQuant, we uses static per-tensor symmetric 8-bit quantization for KV cache following the settings in the TensorRT-LLM [25]. For GPTQ, we use their latest version with "reorder" trick, denoted as "GPTQ-R". For QuaRot and Atom, we mainly evaluated using Pile validation dataset as calibration dataset. We also report their results with WikiText2 as calibration dataset in gray color. For "**W4A8KV4** g128" setting, both QuaRot and Atom does not support progressive group quantization, and thus we evaluated them using ordinary group weight quantization (*i.e.*, each group has one FP16 scale factor). Unsupported models and quantization settings will be reported as NaN.

c) WikiText2 perplexity: Table II compares the Wikitext2 perplexity results between QoQ and other baselines. For Llama-2-7B, compared to **W8A8** SmoothQuant and **W4A16** AWQ, QoQ only increased perplexity by up to 0.16 QoQ consistently outperformed Atom with either **W4A4** or **W4A8KV4** quantization precision. QoQ also showed up to 0.49 perplexity improvement compared to **W4A4** Quarot.

d) Zero-shot accuracy: we report the zero-shot accuracy of five common sense tasks in Table III. QoQ significantly outperformed other 4-bit quantization methods. Especially on the Winogrande task, compared to Quarot, QoQ accuracy is 4.82% higher. Compared to `FP16`, QoQ only introduced 1.03%, 0.89% and 0.40% accuracy loss for Llama-2 at 7B, 13B and 70B size.

C. Efficiency Evaluation

We assessed the efficiency of QServe on A100-80G-SXM4 and L40S-48G GPUs by comparing it against TensorRT-LLM (using `FP16`, **W8A8**, and **W4A16** precisions), Atom (**W4A4**), and QuaRot (**W4A4**). The primary metric for system evaluation is the maximum achievable throughput within the same memory constraints, where we use an input sequence length of 1024 and output sequence length of 512. We notice that Atom only supports Llama-2-7B, and QuaRot does not support GQA. Therefore, we skip these unsupported models when measuring the performance of baseline systems.

We present relative performance comparisons in Figure 15 and absolute throughput values in Table IV. We use per-channel quantization for A100 and per-group quantization for L40S. This is because L40S has stronger CUDA cores for dequantization. Relative to the best-performing configuration of TensorRT-LLM, QServe demonstrates significant improvements on A100: it achieves **2**× higher throughput for Llama-1-30B, **1.2-1.4**× higher throughput for Llama-2 models, **1.2**× higher throughput for Mistral and Yi, and **2.4**× higher throughput for Qwen-1.5. The performance improvements are particularly notable on the L40S GPUs, where we observed a throughput improvement ranging from **1.47**× to **3.47**× across all seven models evaluated. Remarkably, despite the L40S's significantly smaller memory capacity compared to

TABLE II: WikiText2 perplexity with 2048 sequence length. The lower is the better.

WikiText2 Perplexity ↓		Llama-3	Llama-2			Llama			Mistral	Mixtral	Yi
Precision	Algorithm	8B	7B	13B	70B	7B	13B	30B	7B	8x7B	34B
FP16	-	6.14	5.47	4.88	3.32	5.68	5.09	4.10	5.25	3.84	4.60
W8A8	SmoothQuant	6.28	5.54	4.95	3.36	5.73	5.13	4.23	5.29	3.89	4.69
W4A16 g128	GPTQ-R	6.56	5.63	4.99	3.43	5.83	5.20	4.22	5.39	4.08	4.68
	AWQ	6.54	5.60	4.97	3.41	5.78	5.19	4.21	5.37	4.02	4.67
W4A4	QuaRot	8.20	6.10	5.40	3.79	6.26	5.55	4.60	5.71	NaN	NaN
		8.33	6.19	5.45	3.83	6.34	5.58	4.64	5.77	NaN	NaN
W4A4 g128	QuaRot†	7.32	5.93	5.26	3.61	6.06	5.40	4.44	5.54	NaN	NaN
		7.51	6.00	5.31	3.64	6.13	5.43	4.48	5.58	NaN	NaN
	Atom†	7.57	6.03	5.27	3.69	6.16	5.46	4.55	5.66	4.42	4.92
		7.76	6.12	5.31	3.73	6.25	5.52	4.61	5.76	4.48	4.97
W4A8KV4	RTN	9.50	6.51	5.40	3.90	6.51	5.71	4.91	6.18	5.02	6.52
	AWQ	7.90	6.28	5.25	3.68	6.33	5.59	4.61	5.92	4.58	5.26
	Quarot	**6.75**	**5.73**	**5.07**	**3.46**	**5.93**	5.29	**4.32**	**5.41**	NaN	NaN
	Atom	7.37	5.91	5.16	3.60	6.03	5.41	4.49	5.55	NaN	4.84
	QoQ	6.89	5.75	5.12	3.52	**5.93**	**5.28**	4.34	5.45	**4.18**	**4.74**
W4A8KV4 g128	RTN	7.25	5.99	5.19	3.70	6.23	5.46	4.56	5.59	4.39	5.49
	AWQ	6.94	5.83	5.12	3.51	5.93	5.36	4.39	5.50	4.23	4.78
	Quarot‡	**6.68**	5.71	**5.06**	**3.45**	5.91	5.26	4.30	**5.39**	NaN	NaN
	Atom‡	7.04	5.80	5.10	3.53	5.95	5.36	4.41	5.47	4.22	**4.75**
	QoQ	6.76	**5.70**	5.08	3.47	**5.89**	**5.25**	**4.28**	5.42	**4.14**	4.76

* Grayed results use Wikitext2 as calibaration dataset.
† QuaRot and Atom apply group quantization to activations as well.
‡ QuaRot and Atom use ordinary group quantization where each group has one FP16 scale factor.

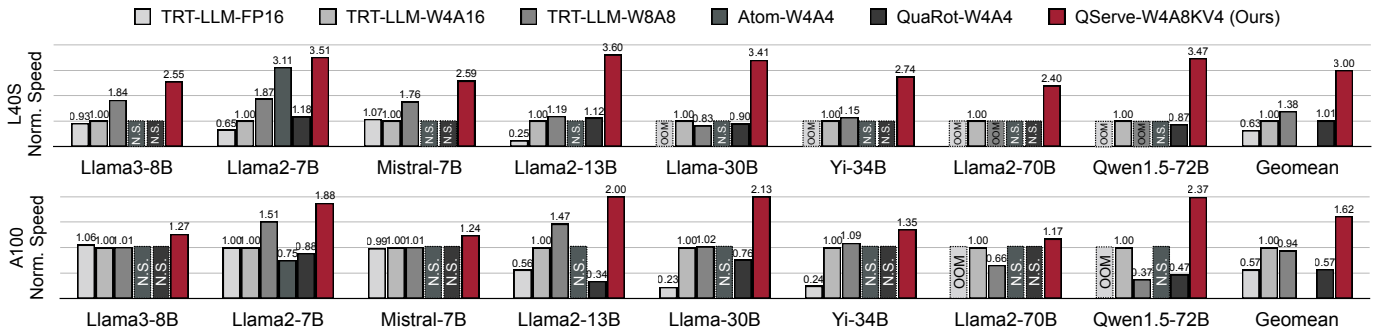

Fig. 15: QServe significantly outperforms existing large language model (LLM) serving frameworks in batched generation tasks across different LLMs, ranging from 7B to 72B models. It achieves an average speedup of 2.36× over the state-of-the-art LLM serving system, TensorRT-LLM v0.9.0, on the L40S GPU, and it is also 1.68× faster on the A100 GPU. All experiments were conducted under the same device memory budget (*i.e.* 80GB on A100 and 48GB on L40S). We omit the geometric mean speedup of Atom since it only supports Llama2-7B. For absolute values, see Table IV.

the A100, QServe effectively maintains the same batch size as TensorRT-LLM on the A100. This achievement is attributed to our aggressive 4-bit quantization applied to both weights and the KV cache. By examining Table IV, we clearly observe that serving five of seven models under 34B on L40S with QServe achieves even higher throughput than serving them on A100 using TensorRT-LLM. Our performance gain over Atom and QuaRot on A100 is even more prominent since these systems did not outperform TensorRT-LLM. On L40S, QServe still achieves 10% higher throughput than Atom when running Llama-2-7B, the only model supported by their system despite the fact that we use higher quantization precision. Besides, the accuracy achieved by QServe is much better than Atom, as indicated in Table III.

D. Analysis and Discussion.

a) Ablation study on quantization techniques: we examine the impact on accuracy of various quantization techniques implemented in QoQ. Our analysis begins with round-to-nearest (RTN) **W8A8** quantization on Llama-2-7B (per-channel + per-token). We then lower the quantization precision and apply different techniques step-by-step. For each step, we evaluated the WikiText2 perplexity and end-to-end inference performance on L40S with 64 requests of 1024 input tokens

TABLE III: Zero-shot accuracy on five common sense tasks with 2048 sequence length.

Llama-2	Precision	Method	Zero-shot Accuracy ↑					
			PQ	ARC-e	ARC-c	HS	WG	Avg.
7B	FP16	-	79.05	74.58	46.25	76.05	68.98	68.98
	W4A4	Quarot	76.77	69.87	40.87	72.16	63.77	64.69
	W4A4 g128	Atom	75.14	52.99	38.40	69.37	62.75	59.73
	W4A8KV4	QoQ	77.64	72.81	43.60	74.00	68.03	67.22
	W4A8KV4 g128	QoQ	**78.07**	**73.32**	**44.80**	**74.98**	**68.59**	**67.95**
13B	FP16	-	80.52	77.44	49.06	79.38	72.22	71.72
	W4A4	Quarot	78.89	72.98	46.59	76.37	70.24	69.01
	W4A4 g128	Atom	76.50	57.49	42.32	73.84	67.40	63.51
	W4A8KV4	QoQ	**79.71**	75.97	48.38	77.80	**70.96**	70.56
	W4A8KV4 g128	QoQ	79.43	**77.06**	**48.81**	**78.35**	70.48	**70.83**
70B	FP16	-	82.70	81.02	57.34	83.82	77.98	76.57
	W4A4	Quarot	82.43	80.43	56.23	81.82	76.24	75.43
	W4A4 g128	Atom	79.92	58.25	46.08	79.06	74.27	67.52
	W4A8KV4	QoQ	82.64	79.80	**56.83**	82.78	77.51	75.91
	W4A8KV4 g128	QoQ	**82.92**	**80.93**	56.40	**83.28**	**78.45**	**76.40**

* For reference, using MX-FP4 for W4A4 quantizing Llama-7B model will decrease the accuracy from 72.9 to 63.7 on ARC easy and from 44.7 to 35.5 on ARC challenge task. [28]

TABLE IV: The absolute token generation throughput of QServe and TensorRT-LLM in Fig. 15. *: we calculate the speedup over highest achieveable throughput from TensorRT-LLM across all three precision configurations. Our QServe system achieves competitive throughput on L40S GPU compared to TensorRT-LLM on A100, effectively reducing the dollar cost of LLM serving by 3×. Unit: tokens/second.

Device	System	Llama-3 8B	Llama-2 7B	Mistral 7B	LLama-2 13B	LLaMA 30B	Yi 34B	Llama-2 70B	Qwen1.5 72B
L40S	TRT-LLM-FP16	1326	444	1566	92	OOM	OOM	OOM	OOM
	TRT-LLM-W4A16	1431	681	1457	368	148	313	119	17
	TRT-LLM-W8A8	2634	1271	2569	440	123	364	OOM	OOM
	QServe (Ours)	**3656**	**2394**	**3774**	**1327**	**504**	**869**	**286**	**59**
	Speedup*	**1.39**×	**1.88**×	**1.47**×	**3.02**×	**3.41**×	**2.39**×	**2.40**×	**3.47**×
A100	TRT-LLM-FP16	2503	1549	2371	488	80	145	OOM	OOM
	TRT-LLM-W4A16	2370	1549	2403	871	352	569	358	143
	TRT-LLM-W8A8	2396	2334	2427	1277	361	649	234	53
	QServe (Ours)	**3005**	**2908**	**2970**	**1741**	**749**	**797**	**419**	**340**
	Speedup*	**1.20**×	**1.25**×	**1.22**×	**1.36**×	**2.07**×	**1.23**×	**1.17**×	**2.38**×

and 512 output tokens. The results are detailed in Figure 16. We see that reducing the weight precision to 4 bits significantly impaired the model performance, though it increased end-to-end processing speed by 1.12× and saved 3.5GB GPU memory. Rotating the block input modules helped suppress the activation outliers, resulting in 0.18 perplexity improvement. In addition, minimizing the block output MSE through weight clipping further decreased the perplexity by 0.16. Consequently, our **W4A8** configuration has achieved a perplexity comparable to that of **W4A16**. However, quantizing KV cache to 4 bits again deteriorated model performance by 0.14, although it substantially enhanced the end-to-end inference throughput by 1.47× and halved GPU memory usage. To solve this problem, SmoothAttention reduced perplexity by 0.05, without adding system overhead. Progressive group quantization further improved perplexity by an additional 0.02, with only a negligible increase in dequantization overhead. Lastly, activation-aware channel reordering enhanced perplexity by 0.03.

b) Ablation study on QServe system: **Dequantization overhead:** We measure the dequantization overhead of per-group QServe-**W4A8** GEMM and other baselines in Figure 18. Our dequantization overhead is comparable with TRT-LLM-**W4A16**, but since we perform computation on INT8 tensor cores, we enjoy 2× higher throughput.

Comparisons under the same batches: We demonstrate

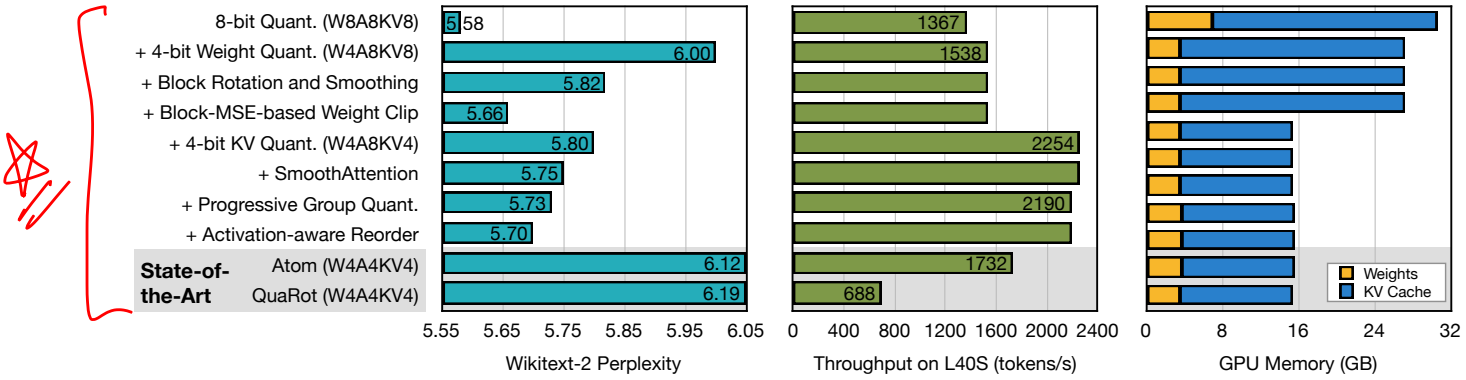

Fig. 16: Ablation study on quantization techniques used in QoQ and the impact of serving throughput, GPU memory consumption in QServe. The model used here is Llama-2-7B.

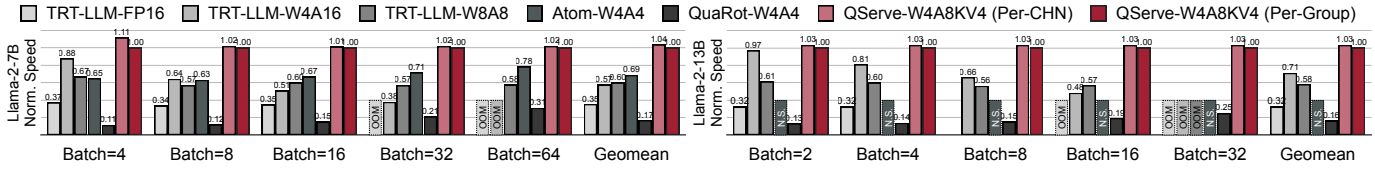

Fig. 17: Same-batch throughput comparison between QServe and baseline systems on L40S. We use an input sequence length of 1024 and output sequence length of 512.

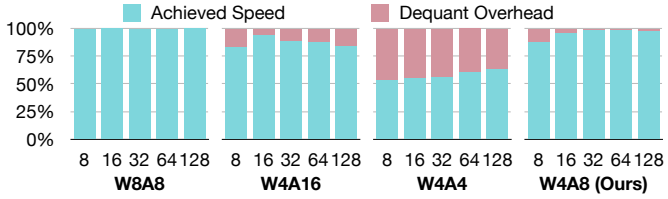

Fig. 18: The dequantization overhead in QServe is much smaller than that in Atom-**W4A4** (up to 90%).

speedup results under the same batch sizes in Figure 17. For Llama-2-7B, we show that the 1.88× speedup over TRT-LLM can be broken down to two parts: 1.45× from same batch speedup and 1.3× from the enlarged batch size. For larger models like Llama-2-13B, scaling up the batch size and single batch speedup are equally important (1.7× improvement).

Improvement breakdown for KV4 attention: We detail the enhancements from attention optimizations in Section Section V-C. Starting with the basic KV4 implementation, which exhibits an A100 latency of 0.48ms for a 64×1024 input, the application of bit tricks from [20] reduces the kernel latency to 0.44ms. Further improvements are achieved by simplifying the control flow, which reduces latency by an additional 0.05ms. Subsequently, converting the QK and SV products to FP16 each contributes a 0.03ms latency reduction. Asynchronous prefetching of dequantization parameters at the start of the attention kernel further enhances performance, ultimately reducing the latency to 0.28ms and achieving an end-to-end improvement of 1.7×.

VII. Related Work

Quantization of LLMs. Quantization reduces the size of LLMs and speedup inference. There are two primary quantization strategies: (1) **Weight-only quantization** [10], [12], [19], [23] benefits edge devices where the workload is memory-bound, improving weight-loading speed. However, for cloud services with high user traffic and required batch processing, this method falls short as it does not accelerate computation in compute-bound scenarios. (2) **Weight-activation quantization** accelerates computation in batch processing by quantizing both weights and activations [8], [36], [38]. OmniQuant [30] and Atom [44] exploring more aggressive quantizations (**W4A4, W4A8**) and mixed precision to enhance model quality and efficiency, though these can impact model accuracy and reduce serving throughput. QuaRot [2] further refines **W4A4** by rotating weights and activations at the cost of increased computational overhead due to additional transformations required during inference.

LLM serving systems. Numerous systems have been proposed for efficient LLM deployment. Orca [40] employs iteration-level scheduling and selective batching in distributed systems. vLLM [22] features virtual memory-inspired PagedAttention, optimizing KV cache management. SGLang [45] enhances LLM programming with advanced primitives and RadixAttention. LMDeploy [7] offers persistent batching and blocked KV cache features to improve deployment efficiency. LightLLM [6] manages GPU memory with token-wise KV cache control via Token Attention, increasing throughput. MLC-LLM [32] utilizes compiler acceleration for versatile LLM deployment across edge devices. TensorRT-LLM [25] is

the leading industry solution and the most important baseline in this paper.

LLM Accelerators. Transformers and LLMs have also generated considerable research interest in domain-specific accelerator design. Several works, such as A^3 [14], ELSA [15], and SpAtten [35], have applied pruning techniques to the attention module, while GOBO [41] and EdgeBERT [31] have investigated quantization approaches. Additionally, DOTA [27] introduces a lightweight, runtime detector for omitting weak attention connections, coupled with specialized accelerators for transformer inference. Apart from attention optimizations, STA [11] leverages $N{:}M$ sparsity and specialized softmax module to reduce off-chip communication. Moreover, DFX [16] exploits model parallelism and optimized dataflow for low-latency generation. However, these accelerators have yet to be scaled up to recent LLMs with billions of parameters.

VIII. Conclusion

We introduce QServe, an algorithm and system co-design framework tailored to quantize large language models (LLMs) to **W4A8KV4** precision, facilitating their efficient deployment on GPUs. On the algorithmic front, we design the QoQ quantization method that features progressive quantization, enabling **W4A8** GEMM operations to be executed on INT8 tensor cores, and SmoothAttention, which significantly reduces accuracy loss resulting from **KV4** quantization. Correspondingly, in the QServe system, we leverage the protective range established in the first level of progressive quantization to enable INT4 to INT8 dequantization. This process utilizes full register-level parallelism and employs a subtraction-after-multiplication computation sequence. Additionally, we implement compute-aware weight reordering to minimize the overhead associated with pointer arithmetic. As a result, when serving seven representative LLMs on A100 and L40S GPUs, QServe achieves up to **2.4-3.5**× higher throughput over the industrial standard for LLM serving, TensorRT-LLM.

Acknowledgements

We thank MIT-IBM Watson AI Lab, MIT AI Hardware Program, MIT Amazon Science Hub, and NSF for supporting this research. We also thank Julien Demouth, June Yang, and Dongxu Yang from NVIDIA for their helpful discussions.

References

[1] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai, "Gqa: Training generalized multi-query transformer models from multi-head checkpoints," *arXiv preprint arXiv:2305.13245*, 2023.

[2] S. Ashkboos, A. Mohtashami, M. L. Croci, B. Li, M. Jaggi, D. Alistarh, T. Hoefler, and J. Hensman, "Quarot: Outlier-free 4-bit inference in rotated llms," *arXiv preprint arXiv:2404.00456*, 2024.

[3] Y. Bisk, R. Zellers, R. L. Bras, J. Gao, and Y. Choi, "Piqa: Reasoning about physical commonsense in natural language," in *Thirty-Fourth AAAI Conference on Artificial Intelligence*, 2020.

[4] J. Chee, Y. Cai, V. Kuleshov, and C. D. Sa, "Quip: 2-bit quantization of large language models with guarantees," 2024.

[5] P. Clark, I. Cowhey, O. Etzioni, T. Khot, A. Sabharwal, C. Schoenick, and O. Tafjord, "Think you have solved question answering? try arc, the ai2 reasoning challenge," 2018.

[6] L. Contributors, "Lightllm: A light and fast inference service for llm," https://github.com/ModelTC/lightllm, 2023.

[7] L. Contributors, "Lmdeploy: A toolkit for compressing, deploying, and serving llm," https://github.com/InternLM/lmdeploy, 2023.

[8] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, "GPT3.int8(): 8-bit matrix multiplication for transformers at scale," in *Advances in Neural Information Processing Systems*, A. H. Oh, A. Agarwal, D. Belgrave, and K. Cho, Eds., 2022.

[9] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," *arXiv preprint arXiv:2305.14314*, 2023.

[10] T. Dettmers, R. Svirschevski, V. Egiazarian, D. Kuznedelev, E. Frantar, S. Ashkboos, A. Borzunov, T. Hoefler, and D. Alistarh, "Spqr: A sparse-quantized representation for near-lossless llm weight compression," 2023.

[11] C. Fang, A. Zhou, and Z. Wang, "An algorithm–hardware co-optimized framework for accelerating n: M sparse transformers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 30, no. 11, pp. 1573–1586, 2022.

[12] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "GPTQ: Accurate post-training compression for generative pretrained transformers," *arXiv preprint arXiv:2210.17323*, 2022.

[13] L. Gao, J. Tow, B. Abbasi, S. Biderman, S. Black, A. DiPofi, C. Foster, L. Golding, J. Hsu, A. Le Noac'h, H. Li, K. McDonell, N. Muennighoff, C. Ociepa, J. Phang, L. Reynolds, H. Schoelkopf, A. Skowron, L. Sutawika, E. Tang, A. Thite, B. Wang, K. Wang, and A. Zou, "A framework for few-shot language model evaluation," 12 2023. [Online]. Available: https://zenodo.org/records/10256836

[14] T. J. Ham, S. J. Jung, S. Kim, Y. H. Oh, Y. Park, Y. Song, J.-H. Park, S. Lee, K. Park, J. W. Lee *et al.*, "Aˆ 3: Accelerating attention mechanisms in neural networks with approximation," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 328–341.

[15] T. J. Ham, Y. Lee, S. H. Seo, S. Kim, H. Choi, S. J. Jung, and J. W. Lee, "Elsa: Hardware-software co-design for efficient, lightweight self-attention mechanism in neural networks," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 692–705.

[16] S. Hong, S. Moon, J. Kim, S. Lee, M. Kim, D. Lee, and J.-Y. Kim, "Dfx: A low-latency multi-fpga appliance for accelerating transformer-based text generation," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 616–630.

[17] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier *et al.*, "Mistral 7b," *arXiv preprint arXiv:2310.06825*, 2023.

[18] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. d. l. Casas, E. B. Hanna, F. Bressand *et al.*, "Mixtral of experts," *arXiv preprint arXiv:2401.04088*, 2024.

[19] S. Kim, C. Hooper, A. Gholami, Z. Dong, X. Li, S. Shen, M. W. Mahoney, and K. Keutzer, "Squeezellm: Dense-and-sparse quantization," 2024.

[20] Y. J. Kim, R. Henry, R. Fahim, and H. H. Awadalla, "Who says elephants can't run: Bringing large scale moe models into cloud scale production," *arXiv preprint arXiv:2211.10017*, 2022.

[21] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 611–626.

[22] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[23] J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, and S. Han, "Awq: Activation-aware weight quantization for llm compression and acceleration," in *MLSys*, 2024.

[24] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," 2016.

[25] NVIDIA, "TensorRT-LLM: A TensorRT Toolbox for Optimized Large Language Model Inference," 2023. [Online]. Available: https://github.com/NVIDIA/TensorRT-LLM

[26] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," 2019.

[27] Z. Qu, L. Liu, F. Tu, Z. Chen, Y. Ding, and Y. Xie, "Dota: detect and omit weak attentions for scalable transformer acceleration," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 14–26.
[28] B. D. Rouhani, R. Zhao, A. More, M. Hall, A. Khodamoradi, S. Deng, D. Choudhary, M. Cornea, E. Dellinger, K. Denolf *et al.*, "Microscaling data formats for deep learning," *arXiv preprint arXiv:2310.10537*, 2023.
[29] K. Sakaguchi, R. L. Bras, C. Bhagavatula, and Y. Choi, "Winogrande: An adversarial winograd schema challenge at scale," *arXiv preprint arXiv:1907.10641*, 2019.
[30] W. Shao, M. Chen, Z. Zhang, P. Xu, L. Zhao, Z. Li, K. Z. Zhang, P. Gao, Y. Qiao, and P. Luo, "Omniquant: Omnidirectionally calibrated quantization for large language models," *arXiv preprint arXiv:2308.13137*, 2023.
[31] T. Tambe, C. Hooper, L. Pentecost, T. Jia, E.-Y. Yang, M. Donato, V. Sanh, P. Whatmough, A. M. Rush, D. Brooks *et al.*, "Edgebert: Sentence-level energy optimizations for latency-aware multi-task nlp inference," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 830–844.
[32] M. team, "MLC-LLM," 2023. [Online]. Available: https://github.com/mlc-ai/mlc-llm
[33] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023.
[34] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.
[35] H. Wang, Z. Zhang, and S. Han, "Spatten: Efficient sparse attention architecture with cascade token and head pruning," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 97–110.
[36] X. Wei, Y. Zhang, X. Zhang, R. Gong, S. Zhang, Q. Zhang, F. Yu, and X. Liu, "Outlier suppression: Pushing the limit of low-bit transformer language models," *arXiv preprint arXiv:2209.13325*, 2022.
[37] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Huggingface's transformers: State-of-the-art natural language processing," 2020.
[38] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han, "SmoothQuant: Accurate and efficient post-training quantization for large language models," in *Proceedings of the 40th International Conference on Machine Learning*, 2023.
[39] A. Young, B. Chen, C. Li, C. Huang, G. Zhang, G. Zhang, H. Li, J. Zhu, J. Chen, J. Chang, K. Yu, P. Liu, Q. Liu, S. Yue, S. Yang, S. Yang, T. Yu, W. Xie, W. Huang, X. Hu, X. Ren, X. Niu, P. Nie, Y. Xu, Y. Liu, Y. Wang, Y. Cai, Z. Gu, Z. Liu, and Z. Dai, "Yi: Open foundation models by 01.ai," 2024.
[40] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A distributed serving system for Transformer-Based generative models," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 521–538. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/yu
[41] A. H. Zadeh, I. Edo, O. M. Awad, and A. Moshovos, "Gobo: Quantizing attention-based nlp models for low latency and energy efficient inference," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 811–824.
[42] R. Zellers, A. Holtzman, Y. Bisk, A. Farhadi, and Y. Choi, "Hellaswag: Can a machine really finish your sentence?" *CoRR*, vol. abs/1905.07830, 2019. [Online]. Available: http://arxiv.org/abs/1905.07830
[43] L. Zhang, W. Fei, W. Wu, Y. He, Z. Lou, and H. Zhou, "Dual grained quantization: Efficient fine-grained quantization for llm," *arXiv preprint arXiv:2310.04836*, 2023.
[44] Y. Zhao, C.-Y. Lin, K. Zhu, Z. Ye, L. Chen, S. Zheng, L. Ceze, A. Krishnamurthy, T. Chen, and B. Kasikci, "Atom: Low-bit quantization for efficient and accurate llm serving," in *MLSys*, 2023.
[45] L. Zheng, L. Yin, Z. Xie, J. Huang, C. Sun, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. Barrett, and Y. Sheng, "Efficiently programming large language models using sglang," 2023.