A Survey of Low-bit Large Language Models: Basics, Systems, and Algorithms

Ruihao Gong[1a], Yifu Ding[a], Zining Wang[a], Chengtao Lv[a], Xingyu Zheng[a], Jinyang Du[a], Haotong Qin[b], Jinyang Guo[a], Michele Magno[b], Xianglong Liu[2a]

[a]*Beihang University, 37 Xueyuan Road, Haidian District, Beijing, 100191, Beijing, China*
[b]*ETH Zurich, Rämistrasse 101, Zurich, 8092, , Switzerland*

Abstract

Large language models (LLMs) have achieved remarkable advancements in natural language processing, showcasing exceptional performance across various tasks. However, the expensive memory and computational requirements present significant challenges for their practical deployment. Low-bit quantization has emerged as a critical approach to mitigate these challenges by reducing the bit-width of model parameters, activations, and gradients, thus decreasing memory usage and computational demands. This paper presents a comprehensive survey of low-bit quantization methods tailored for LLMs, covering the fundamental principles, system implementations, and algorithmic strategies. An overview of basic concepts and new data formats specific to low-bit LLMs is first introduced, followed by a review of frameworks and systems that facilitate low-bit LLMs across various hardware platforms. Then, we categorize and analyze techniques and toolkits for efficient low-bit training and inference of LLMs. Finally, we conclude with a discussion of future trends and potential advancements of low-bit LLMs. Our systematic overview from basic, system, and algorithm perspectives can offer valuable insights and guidelines for future works to enhance the efficiency and appli-

[1]Ruihao Gong leads the overall organization of the survey, with Yifu Ding and Jinyang Du contributing to Sections 2 and 3. Xingyu Zheng is responsible for authoring Section 4, while Chengtao Lv and Zining Wang collaborate on Section 5. Haotong Qin, Jinyang Guo, Michele Magno, and Xianglong Liu provide guidance during the whole process and assist in refining the final manuscript.

[2]Corresponding author. Email: xlliu@buaa.edu.cn

cability of LLMs through low-bit quantization.

Keywords:
Large Language Model, Quantization, Low-bit, System, Algorithm

1. Introduction

Large language models (LLMs) (OpenAI et al., 2024; Touvron et al., 2023a,b; Dubey et al., 2024; Lozhkov et al., 2024; Liu et al., 2024a) have revolutionized natural language processing by delivering unprecedented performance across a range of tasks, from text generation to language understanding. However, their remarkable capabilities come with significant computational and memory demands. This has raised considerable challenges when deploying these models in scenarios with limited resources or high concurrency. To address these challenges, low-bit quantization has emerged as a pivotal approach for enhancing the efficiency and deployability of LLMs.

Low-bit quantization involves the process of reducing the bit-width of tensors, which effectively decreases the memory footprint and computational requirements of LLMs. By compressing weights, activations, and gradients of LLMs with low-bit integer/binary representation, quantization can significantly accelerate inference and training and reduce storage requirements with acceptable accuracy. This efficiency is crucial for enabling advanced LLMs to be accessible on devices with constrained resources, thereby broadening their applicability.

In this paper, we aim to provide a survey with a comprehensive overview of low-bit quantization for large language models (LLMs), encompassing the fundamental concepts, system implementations, and algorithmic approaches related to low-bit LLMs. Compared with the traditional models, LLMs, as the representative paradigm of the foundation model, always feature a vast number of parameters, which presents unique challenges for effective quantization. As depicted in Figure 1, Section 2 introduces the fundamentals of low-bit quantization of LLMs, including new low-bit data formats and quantization granularities specific to LLMs. Section 3 reviews the systems and frameworks supporting low-bit LLMs across various hardware platforms. We then categorize low-bit quantization techniques for efficient training and inference in Sections 4 and 5, respectively. For training, we discuss methods for low-bit training and fine-tuning of LLMs. For inference, we differentiate LLM quantization methods by quantization-aware training and post-training

quantization. Quantization-aware training is often used for low-bit settings (such as binary quantization). Post-training quantization is more commonly applied in existing research since it is a resource-efficient pipeline. For a clear understanding, we first cover the widely used techniques of equivalent transformation for reducing outlier influence and weight compensation for mitigating quantization errors. Then the mixed precision, techniques that combine quantization with other compression methods, as well as methods for new quantization forms are discussed. Additionally, we summarize toolkits that integrate these algorithms to support the development of accurate low-bit LLMs. Finally, Section 6 explores future trends and directions, discussing emerging research areas, potential breakthroughs, and the impact of new technologies on LLM quantization. Our survey provides a detailed description of the fundamentals of low-bit LLMs and gives a comprehensive view of the system implementations for accelerating training and inference through low-bit quantization and algorithms and strategies to maintain and enhance quantized accuracy. We believe this survey can provide valuable insights and advance the development of LLM quantization.

2. Basics of Low-bit LLMs

In this section, we introduce the basic fundaments of quantization and low-bit LLMs from three aspects: (1) Low-bit number formats. To deal with the outliers in LLMs, low-bit floating-points are first used in quantization. And lots of custom data formats are designed to tackle the outliers. However, integers are still the mainstream. (2) Quantization granularity. To improve the performance of quantized LLMs, finer-grained quantization retains more information and generates better results. But course-grained ones occupy less storage and are more efficient in inference. (3) Dynamic or static quantization. Dynamic quantization does not require calibration, as the quantization parameters are calculated on the fly, making the preparation of a quantized model simpler. In contrast, static quantization requires pre-calibration of quantization parameters, but it offers faster inference performance.

2.1. Low-bit Number Formats

We start with the low-bit number formats at the beginning of the introduction. First, we demonstrate the standard formats that are well-recognized, but focus on the differences in LLMs. Second, we introduce some typical custom formats that are designed for LLMs.

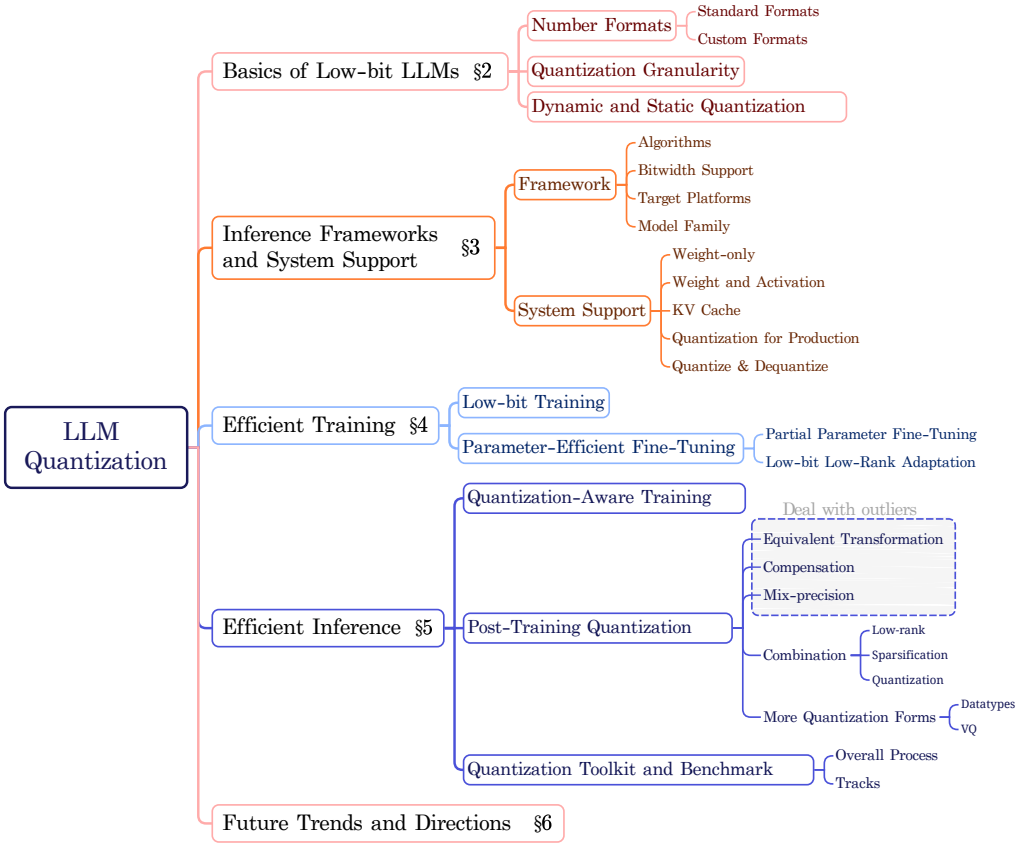


Figure 1: The skeleton of the LLM Quantization methods. The diagram illustrates the main areas in the survey.

2.1.1. Standard Formats

Floating-point Numbers. The floating-point data type is comprehensively defined in the IEEE 754 (iee, 2019) standard, which is also the most prevailing number format in computer systems. Let us denote them as FPk, where k represents the number of bits that the value occupies in memory, usually $32, 16, 8$, etc. A floating-point number can be uniformly expressed as

$$X_{\text{FP}k} = (-1)^s 2^{p-bias}(1.\texttt{mantissa}) = (-1)^s 2^{p-bias}(1 + \frac{d_1}{2} + \frac{d_2}{2^2} + ... + \frac{d_m}{2^m}), \quad (1)$$

where s is the sign bit, p is the exponent integer, $bias$ is applied to the exponent, m is the total number of mantissa bits in the significand, and $d_1, d_2, \ldots, d_m$ represent the digits of the mantissa part in the binary format.

The bits of s, p and m should be accumulated to k for an FPk value.

Since LLMs occupy more memory, lower-bit formats become popularly adopted in both training and inference. We omit the 32-bit number format here since the 16 and lower bitwidth has become the mainstream practice in application. We can further categorize each FPk according to its bit allocations for the exponent (E) and mantissa (M) parts. We use EeMm to denote the subcategories. As for FP16, IEEE 754 defines float16 (also known as half-precision or FP16) and bfloat16 (brain floating point or BF16), which can be represented as E5M10 and E8M7, respectively. Therefore, bfloat16 can represent larger magnitudes with more exponent bits (identical to that of FP32) while more sparse than float16 with less mantissa in the significand, which may exert unprecedented potential in LLMs (Henry et al., 2019). As well as E4M3 and E5M2 for FP8, both are standard formats that are already supported by several mainstream deep learning inference engines, such as MLC-LLM, Quanto, and so on (see Sec. 3.1.2 for details).

NormalFloat (NF) (Dettmers et al., 2024) is a fixed floating-point method used in weight-only quantization strategies for LLMs. The data representing format follows the floating-points, but the 2^k values $X_i^{\mathrm{NF}}, i \in [0, 2^k - 1]$ are estimated to be

$$X_i^{\mathrm{NF}} = \frac{1}{2}\left(quantile\left(N(0,1), \frac{i}{2^k+1}\right) + quantile\left(N(0,1), \frac{i+1}{2^k+1}\right)\right), \tag{2}$$

where $quantile(\cdot, q)$ is the q-th quantiles of the input. $N(0,1)$ means the standard normal distribution. For a tensor that does not fall within the range of -1 to 1, we must first scale it using its maximum absolute value. To ensure the exact representation for zero, it asymmetrically divides the data into the positive and negative parts by estimating 2^{k-1} of X_i^{NF} for the negative and $2^{k-1} - 1$ for the positive, then removes one of the zeros in both sets. NF is estimated to have an almost equal expected number of values in each quantization bin to keep the most information in the quantized formats.

Micro Scaling FP (Rouhani et al., 2023). It was proposed and developed in collaboration with industry alliance members, including AMD, Arm, Intel, Meta, Microsoft, NVIDIA, and Qualcomm, which aims to establish a unified standard for fine-grained sub-blocks of tensor format. It applies E8M0 scaling factors on a block of data with various original formats (i.e., FP8, FP6, FP4, INT8). The scaling block size indicates the number of elements that each scaling applies. It keeps high precision for the value

Format	Max (normal)	Min (normal)
INT4	7	-8
INT8	127	-128
FP8 (E4M3)	128	-128
FP8 (E5M2)	32768	-32768
FP16 (E5M10)	65504	-65504
BF16 (E8M7)	3.39e38	-3.39e38

Table 1: Min and Max values for different number formats (iee, 2019).

representation but is significantly efficient on hardware due to the shared scalings.

Integer Numbers. Integer quantization is the most widely studied quantized data format since the quantization technique has emerged. It divides the floating-points into 2^k equally spaced discrete integers. The formula is:

$$X_{\mathrm{INT}_k} = (-1)^s(d_1 2^m + d_2 2^{m-1} + \cdots + d_m 2^0), \quad x \in \mathbb{N}^+, \tag{3}$$

where $m = k - 1$ and $s \in \{0, 1\}$ for signed integers. $m = k$ for unsigned integers while we regard $s = 0$. Therefore, the signed integers range from $[-2^{k-1}, 2^{k-1} - 1]$, and the unsigned one $[0, 2^k - 1]$.

Binarized Numbers. Binarization is the most aggressive quantization technique, which directly abstracts the sign of value (Liu et al., 2018; Qin et al., 2022; Li et al., 2024d). It will lose most information, but bring significant acceleration and parameter compression in inference. The hardware takes $0, 1$ for each bit originally, but developers define different logic rules and accumulation algorithms to achieve various binarized computations. Therefore, floating-point numbers can be binarized to $\{-1, 1\}$ or $\{0, 1\}$, depending on what value we expect the single bit to represent in our algorithms.

Table 1 shows the representation ranges of various standard formats. It shows that even with the same bit-width, different numerical representation formats can have significantly different value ranges. The floating-point numbers with larger E have larger representation ranges but sparser points. Therefore, there is a tradeoff between finer data intervals or larger data ranges when determining data formats for a specific model and task.

2.1.2. Custom Formats

For faster computation and better fitting the numerical distributions of LLMs, many studies propose custom number formats besides the standard formats de-

scribed above. Here we introduce three typical customized formats. We omit the works before the advent of LLMs (Tambe et al., 2019) because their performance has not been validated on LLMs.

Floating-point Integer (Flint) (Guo et al., 2022) combines the advantages of floating-point and integer representations, which is $X_{\text{Flint}} = 2^{p-bias} \times (1.\texttt{mantissa})$. We take the 4-bit Flint on float-based MAC units as an example:

$$p = \begin{cases} 3 - \texttt{LZD}(b_2b_1b_0), & b_3 = 0 \\ 4 + \texttt{LZD}(b_2b_1b_0), & b_3 = 1 \end{cases}, \quad \texttt{mantissa} = b_2b_1b_0\texttt{<<}(\texttt{LZD}(b_2b_1b_0)+1), \tag{4}$$

where the LZD denotes the Leading Zero Detector (Oklobdzija, 1994) which accumulates the leading zeros on the left of the bitstring, << is the left shift operation, and $bias = 1$ for float-based Flint4. It expands the range by integrating exponents into the integers, therefore. Compared to pure integers, Flint can represent a larger range with a limited number of bits, which better fits the distribution of LLM parameters.

Adaptive Biased Float (Abfloat) is first proposed in Outlier-Victim Pair Quantization (OVP) (Guo et al., 2023a) to deal with outliers. The difference to Flint is that Abfloat applies a bigger *bias* to the exponent, and left shifts m-bit to enlarge the 1 before mantissa, making the magnitude even larger to cover the outliers. The EeMm Abfloat value can be expressed as:

$$X_{\text{Abfloat}} = (-1)^s \times 2^{p+bias} \times (2^m + \texttt{mantissa}). \tag{5}$$

When bias = 0, the range is similar to Flint4. With bias = 2 for E2M1, the range changes to $\{12, \ldots, 96\}$. With bias = 3, the range further extends to $\{24, \ldots, 192\}$. The other difference to Flint is that Abfloat is only adopted on outliers, but the normal values are stored in INT4/8 or Flint4. Both data formats require custom system support to define the behavior of the base operations (such as addition, multiplication, and so on).

Student Float (SF) (Dotzel et al., 2024a) follows the floating-point format but has specific fixed points for quantization, which is different from the above two types. SF is an improvement of NF in Sec. 2.1.1 and holds the view that the parameters obey Student's t-distribution $S(t;\nu)$, of which the probability density function is:

$$S(t;\nu) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu\pi}\Gamma\left(\frac{\nu}{2}\right)} \left(1 + \frac{t^2}{\nu}\right)^{-\frac{\nu+1}{2}}, \tag{6}$$

where t and ν are the independent variable and degrees of freedom, respectively, and Γ is generalized factorial.

$$\tilde{X}_i^{\text{SF}} = quantile\left(S(t;\nu), q_i\right), \quad q_i = \begin{cases} \delta + (\frac{1}{2} - \delta)\frac{i-1}{7} & i \in \{1, \ldots, 8\} \\ \frac{1}{2} + (\frac{1}{2} - \delta)\frac{i-8}{8} & i \in \{9, \ldots, 16\} \end{cases}, \tag{7}$$

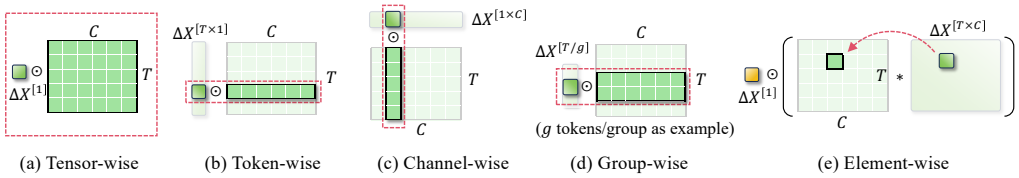


Figure 2: Illustrations for different quantization granularity.

where $\delta = \frac{1}{2}(\frac{1}{32} + \frac{1}{30})$, $\{q_1, \ldots, q_8\}$ and $\{q_9, \ldots, q_{16}\}$ are two groups of evenly spaced quantiles. Then we normalize $\tilde{X}^{\text{SF}}$ to $[-1, 1]$ by $X_i^{\text{SF}} = \frac{\tilde{X}_i^{\text{SF}}}{\max_i |\tilde{X}_i^{\text{SF}}|}$. As ν increases, the peaks of the t-distribution become shorter and wider, and SF4 spreads out more. It converges to the standard normal distribution (NF) as $\nu \to \infty$. Same as NF, SF is used in weight-only quantization (which we introduce in Sec. 3.2.1). Therefore, it does not need the low-level definition of base operations but requires a custom dequantization from SF to standard formats.

2.2. Quantization Granularity

Quantization granularity refers to the different weight/activation partitions corresponding to each element of the scaling factor and zeropoint. It determines how finely the scale recovers and the zero point shifts. Figure 2 showcases five fundamental types of quantization granularity: tensor-wise, token-wise, channel-wise, group-wise, and element-wise.

Tensor-wise is the simplest and coarsest granularity, which takes a single scaling factor and zero point to the entire tensor (Zhang et al., 2024c). It can be the fastest but may lead to the most performance degradation because it is incapable of handling the values with a wide variation. Therefore, it is unsuitable for cases where accuracy is important or the task/model is sensitive to quantization.

Token-wise is used in LLMs only, which means that each token (word or subword) has a scaling (Yao et al., 2022). It captures the fine-grained variations in different tokens. Usually, we adopt dynamic token-wise quantization for activation to reduce the quantization error and ensure diversity in generative models.

Channel-wise means each channel in weight within a tensor uses one scale and can be merged into quantized weight (Kim et al., 2024). Token-wise activation and channel-wise weight are usually used together. Because for i-th token in activation and j-th channel in weight, the corresponding $s_{X_i} \in s_X \in \mathbb{R}^{T\times 1}$ and $s_{W_j} \in s_W \in \mathbb{R}^{1\times C}$ can be calculated first as $s \in \mathbb{R}^{[1]}$ and multiplied to the coordinate $[i, j]$ in output matrix X_O. In this way, we preserve the generation performance with little computation overheads.

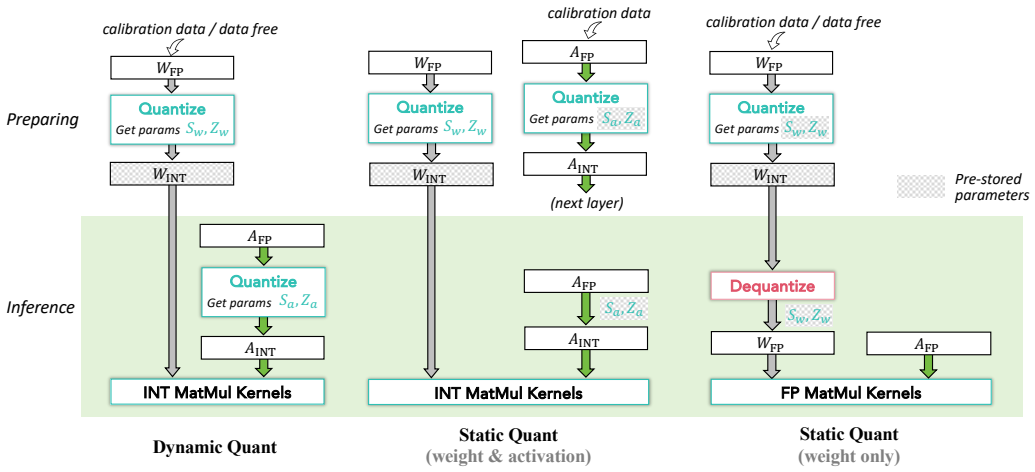


Figure 3: Dynamic and static quantization. Operations in the green block mean the inference process, while outside the block is the production and preparation process.

Group-wise balances the computational complexity and the quantization error by grouping tensors or channels with the same scaling factor. It also reduces the storage of scaling factors by g if there are g tokens/channels per group (Heo et al., 2023; Yao et al., 2022).

Element-wise is only applied while training the weight, which is always used together with another quantization granularity, such as tensor-wise (see Figure 2(e)). Before inference, the element-wise scaling is merged into the quantized weight. Therefore, only the tensor-wise scale is needed to be computed in inference (Lee et al., 2023) to recover the value magnitude.

Different quantization granularity are always combined and adopted together. For example, Lee et al. (2023) uses a channel-wise scale for the Key matrix but a token-wise scale for the Value matrix based on the distribution of the data. More algorithms can be found in Sec. 5.2.3.

2.3. Dynamic and Static Quantization

Dynamic and static quantization mainly refers to the strategies in PTQ, which are illustrated in Figure 3. We take integer quantization as an example, and other low-bit quantization methods have a similar process.

Dynamic Quantization (Krishnamoorthi, 2018; Liu et al., 2022) calibrates and stores quantized weight. Usually, it does not need input data, but searches for the optimal scaling factors s_W and zero-points Z_W by minimizing the quantization error for each tensor of weight. During inference, the activation will be input into the quantization module to compute the optimal scaling factors s_X and Z_X, and

then quantized to INT8 by the dynamically computed factors before conducting integer GEMM with quantized weight. The scaling and zero point of activation are obtained in real-time based on the current batch of input data. Therefore, the scaling factor flexibly adapts the input data distribution, bringing the smallest quantization error. While it takes extra computational complexity to get the scale during inference. It is suitable for scenarios that require rapid deployment because it does not require calibration.

Static Quantization (Bai et al., 2021) takes calibration data consisting of a small fraction of the training dataset. By inputting the samples into the model, we find the optimal scaling factors for both weight and activation (the middle one in Figure 3) or weight only (the right one), and are fixed during inference. It allows for the evaluation of the quantized model during preparation, ensuring that quantization does not significantly harm the model's performance. As for inference, the middle one in Figure 3 quantizes the activation to low-bit and computes low-bit GEMM (Dettmers et al., 2022b) with quantized weight. For the right one in Figure 3, the weight will be dequantized to floating-point numbers, and the activation will not be quantized before conducting floating-point GEMM (Lin et al., 2024a), thus we name it weight-only quantization.

3. Frameworks and System Support

In the few short years since the large language model emerged, there have arisen many frameworks to support the easy usage of LLMs. We have selected some well-known representative frameworks and tools related to quantization, summarized and introduced them in this section according to the following categories: (1) **Inference framework for quantization**, which provides comprehensive libraries and APIs for the rapid development and deployment of LLM applications, (2) **System support for quantization**, which supports the underlying core functionality for quantization methods. In the following, our emphasis is on the quantization of LLMs across various frameworks and libraries.

3.1. Inference Framework for Quantization

We list the representative inference frameworks in Table 2. Currently, no single inference framework dominates in terms of performance or usage. However, some classic deep learning frameworks, such as TensorRT-LLM[1], ONNX-runtime[2],

[1]https://github.com/NVIDIA/TensorRT-LLM
[2]https://github.com/microsoft/onnxruntime

Framework	Algorithms	Model family	Device	W&A$_{bit}$	Weight-only$_{bit}$	KV Cache$_{bit}$	Organ.
TensorRT-LLM	AWQ, GPTQ, LoRA, SmoothQuant	Transformer-like (Llama)	NVIDIA GPU, Intel CPU	$W_{\text{fp8}}A_{\text{fp8}}$, $W_{\text{int4}}A_{\text{int4}}$, $W_{\text{int8}}A_{\text{int8}}$	int4, int8	fp8, int4, int8	NVIDIA
Transformers	AQLM, AWQ, AutoGPTQ, EETQ, HQQ, Exllama, QAT, Quanto, SmoothQuant	Mixture-of-Expert, Transformer-like (Llama)	Intel CPU, MPS, Numba, NVIDIA GPU, Triton	$W_{\text{fp8}}A_{\text{fp8}}$, $W_{\text{int4}}A_{\text{int4}}$, $W_{\text{int8}}A_{\text{fp8}}$, $W_{\text{int8}}A_{\text{int8}}$	int2, int4, int8, fp8		HuggingFace
OpenVINO	AWQ, GPTQ, SmoothQuant	Multi-modal	CPU, NPU, NVIDIA GPU	$W_{\text{int8}}A_{\text{int8}}$	int4, int8		Intel
ONNX-Runtime	GPTQ, HQQ, LoRA, RTN	Mixture-of-Expert, Transformer-like (Llama), falcon	Android, Intel CPU, Ascend NPU, Mac, NVIDIA GPU, iOS/iPadOS	fp16 only	fp4, fp8, int4, int8		Microsoft
PowerInfer	LoRA	Mixture-of-Expert, Transformer-like (Llama), falcon	AMD GPU, CPU, NVIDIA GPU, Intel CPU	fp16 only	int4		Shanghai Jiao Tong University
PPLNN		Mixture-of-Expert, Transformer-like (Llama)	CPU, CUDA, NVIDIA GPU, x86_64 CPU	$W_{\text{int8}}A_{\text{int8}}$	int8	int8	OpenMMLab & SenseTime
Xorbits Inference	AWQ, GPTQ, LoRA	Multi-modal, Transformer-like (Llama)	CPU, Metal	$W_{\text{int4}}A_{\text{int4}}$	int3, int4, int8	int8	Xorbits
bitsandbytes	LLM.int8(), LoRA	Mixture-of-Expert, Transformer-like (Llama)	AMD GPU, Intel CPU	$W_{\text{int8}}A_{\text{int8}}$	int2, int4, int8, fp8		HuggingFace
DeepSpeed-MII	FP6-LLM, ZeroQuant	Mixture-of-Expert, Multi-modal LLMs (e.g. LLaVA), Transformer-like (Llama), falcon	NVIDIA GPU	$W_{\text{int4}}A_{\text{int4}}$, $W_{\text{int8}}A_{\text{fp8}}$, $W_{\text{int8}}A_{\text{int8}}$	fp6, int8		Microsoft
vLLM	AQLM, AWQ, GPTQ, LoRA, SmoothQuant, SqueezeLLM	Mixture-of-Expert, Multi-modal LLMs (e.g. LLaVA), Transformer-like (Llama)	AMD GPU, Neuron devices, NVIDIA GPU, TPU, XPU, x86_64 CPU	$W_{\text{fp8}}A_{\text{fp8}}$, $W_{\text{int4}}A_{\text{int4}}$, $W_{\text{int4}}A_{\text{int8}}$, $W_{\text{int8}}A_{\text{fp8}}$, $W_{\text{int8}}A_{\text{int8}}$	int4	fp8	University of California, Berkeley
ctransformers	Exllama, GPTQ	Transformer-like (Llama)	AMD GPU, Mac, NVIDIA GPU	fp16 only	fp8, int2, int3, int4, int5, int6		Ravindra Marella
MLC-LLM	AWQ	Mixture-of-Expert	AMD GPU, Android, Mac, NVIDIA GPU, Vulkan, iOS/iPadOS	fp16 only	fp8, int3, int4, int8		MLC
LMDeploy	AWQ	Mixture-of-Expert, Multi-modal LLMs (e.g. LLaVA), Transformer-like (Llama), falcon	Jetson, NVIDIA GPU	$W_{\text{int8}}A_{\text{int8}}$	int4	int4, int8	Shanghai AI Lab & SenseTime
LightLLM	AWQ, GPTQ	Mixture-of-Expert, Transformer-like (Llama)	NVIDIA GPU, Triton	$W_{\text{int8}}A_{\text{int8}}$	int4, int6, int8	int8	SenseTime
QServe	AWQ, Atom, GPTQ, QoQ, QuaRot, SmoothQuant	Mixture-of-Expert, Transformer-like (Llama)	NVIDIA GPU	$W_{\text{int4}}A_{\text{int4}}$, $W_{\text{int4}}A_{\text{int8}}$, $W_{\text{int8}}A_{\text{int8}}$	int4, int8		MIT EECS
llama.cpp	AWQ	Mixture-of-Expert, Multi-modal LLMs (e.g. LLaVA), Transformer-like (Llama), falcon,	AMD GPU, NVIDIA GPU	$W_{\text{int4}}A_{\text{int4}}$	fp8, int2, int3, int4, int5, int6		ggml
llama2.c	LoRA	Transformer-like (Llama)	Mac, NVIDIA GPU, Intel CPU	fp16 only	int8		Andrej Karpathy
inferflow	AWQ, LoRA	Mixture-of-Expert, Transformer-like (Llama), falcon,	Mac, NVIDIA GPU, Intel CPU	fp16 only	int2, int3, int4, int5, int6, int8		InferFlow
ScaleLLM	AWQ, AutoGPTQ, Exllama, GPTQ	Mixture-of-Expert, Transformer-like (Llama)	CPU, NVIDIA GPU	$W_{\text{int4}}A_{\text{int4}}$	int2, int4, int8		Vectorch
SGLang	AWQ, GPTQ	Mixture-of-Expert, Transformer-like (Llama), Multi-modal	NVIDIA GPU	$W_{\text{int8}}A_{\text{int8}}$	int4, int8	fp8	LMSYS.org

Table 2: Inference frameworks for quantized large language models.

Transformers[3] (Huggingface), OpenVINO[4], PowerInfer[5], PPLNN[6], and Xorbits

[3]https://huggingface.co/docs/transformers/en/index
[4]https://github.com/openvinotoolkit/nncf
[5]https://github.com/SJTU-IPADS/PowerInfer
[6]https://github.com/openppl-public/ppl.nn

Inference [7] have integrated the support for efficient inference of large models. In addition, other inference frameworks emerged after the advent of large models that are specifically proposed for LLMs, such as bitsandbytes[8], ctransformers[9], MLC-LLM[10], DeepSpeed-MII[11], vLLM[12], LMDeploy[13], LightLLM[14], QServe[15], llama.cpp[16], llama2.c[17], inferflow[18], ScaleLLM [19], SGLang [20] and so on. These frameworks are lightweight and have integrated many specialized optimization techniques for large models.

3.1.1. Ready-to-use Algorithms

With the emergence of quantization algorithms for LLMs, some typical methods have already been integrated into most frameworks, while some methods may be developed and published originally on a specific framework. We list the most ready-to-use algorithms in each mainstream framework in Table 2. Some methods are included by the most frameworks, such as GPTQ (Frantar et al., 2022), AWQ (Lin et al., 2024a), SmoothQuant (Xiao et al., 2023), LoRA (Hu et al., 2021) and so on. These methods share several advantages: high accuracy and efficient performance after quantization, seamless integration into existing implementation procedures, and user-friendliness.

In addition, some algorithms are supported by several frameworks. For example, LLM.int8() (Dettmers et al., 2022b) was well supported by bitsandbytes (by HuggingFace), which allows to store and load 8-bit weights directly from the HuggingFace Hub and quantize weight in linear layers to 8-bit. FP6-LLM (Xia et al., 2024) is integrated in DeepSpeed-FastGen[21] (Holmes et al., 2024) to implement the runtime quantization for 6-bit floating-point weight-only quantization. It allows efficient quantization and dequantization of 6-bit weight LLMs through

[7] https://github.com/xorbitsai/inference
[8] https://github.com/bitsandbytes-foundation/bitsandbytes
[9] https://github.com/marella/ctransformers
[10] https://github.com/mlc-ai/mlc-llm
[11] https://github.com/microsoft/DeepSpeed-MII
[12] https://github.com/vllm-project/vllm
[13] https://github.com/InternLM/lmdeploy
[14] https://github.com/ModelTC/lightllm
[15] https://github.com/mit-han-lab/qserve
[16] https://github.com/ggerganov/llama.cpp
[17] https://github.com/karpathy/llama2.c
[18] https://github.com/inferflow/inferflow
[19] https://github.com/vectorch-ai/ScaleLLM
[20] https://github.com/sgl-project/sglang
[21] https://github.com/microsoft/DeepSpeed/tree/master/blogs/deepspeed-fastgen

a unified configuration option. It is noteworthy that Transformers (by HuggingFace) and QServe (by MIT EECS Lin et al. (2024b)) integrate most algorithms with comprehensive user manuals and detailed examples, enabling a quick start for deep learning researchers and developers.

3.1.2. Bitwidth Support

The support for bitwidth always reflects how comprehensive the quantization system implementation is for an inference framework or engine. It can be categorized into three types according to its position and function in accelerating LLMs:

Weight-only$_{bit}$ means only quantizing the weight while keeping FP16 activation (Lin et al., 2024a). The quantized weight will be dequantized back to FP16 using pre-obtained scaling factors, and then conduct FP16 `mma` with FP16 activation. Therefore, it theoretically supports non-uniform quantization with arbitrary bitwidth. The speedup is achieved by reducing the latency of data transmission between the computing device and storage host with smaller amounts of weight data, but the dequantizing of weight costs extra time. The detailed speedup will be discussed in Sec. 3.2.1.

W&A$_{bit}$ means that the algorithm quantizes both the weight and activation, and conducts low-bit matrix multiplication (MatMul) in low-level (for example, in PTX ISA 8.5[22] for NVIDIA GPUs, instruction `mma.sync.aligned.shape.row.col .s32.u4.u4.s32` means the data type of the multipliers is the 4-bit unsigned integer). All the frameworks support the INT8 and FP16 MatMul. However, limited by the computing capabilities of the hardware and the supported operations in the instruction set, only part of them have INT4 and FP8 MatMul. Few supports different bitwidth of weight and activation (like $W_{\text{int4}}A_{\text{int8}}$), which requires customized computation kernels with assembled GEMV instructions[23] (Egiazarian et al., 2024a). It should be mentioned that if you want to use low-bit MatMul, your hardware architecture must support the specific low-bit computing, and it is necessary to upgrade/downgrade the driver to the corresponding version to reproduce the real low-bit computation and get the desired speedup ratio.

KV Cache$_{bit}$ lists the bitwidth of Key-Value Cache. As a caching technology, memory consumption of the KV cache increases rapidly as batch size and sequence length continue to grow, potentially surpassing the model size. Therefore, quantizing the KV cache significantly reduces memory usage during model inference. There are several works devoted to quantizing the KV cache (Hooper et al., 2024; Yue et al., 2024; Liu et al., 2024c). Similar to weight-only algorithms, the quantized

[22]https://docs.nvidia.com/cuda/parallel-thread-execution/index.html

[23]https://huggingface.co/docs/transformers/main/en/quantization/eetq

key-value pairs usually need to be dequantized to floating-point before MatMul, otherwise, the specific system support of multiplying low-bit to floating-point is required. Except for the listed bitwidth, all frameworks support the FP16 KV cache, which means directly storing the activation.

3.1.3. Target Platforms

Numerous vendors are competing fiercely in the deep learning hardware. As one of the pioneers in the field of deep learning GPUs today, NVIDIA GPUs are supported by most frameworks. Meanwhile, vLLM, bitsandbytes, llama.cpp, ctransformers, MLC-LLM, and PowerInfer also have the support for AMD GPUs. For some other processing units, such as TPU, XPU, Vulkan, Metal, and other hardware, the system support is relatively limited. Some frameworks that are devoted to generalizing LLMs to edge devices are more likely to extend the support for those platforms, such as MLC-LLM, ONNX-Runtime, and llama.cpp. However, it should be noted that the frameworks with support for both low-bit quantization and hardware deployment in Table 5 cannot guarantee the deployment of any quantized model on each listed hardware. Users should carefully refer to the manual for guidance. However, the table we compiled may help reduce the time it takes to find a suitable framework that may meet your deployment desire.

3.1.4. Model Family

All the frameworks support custom model definition and seamlessly integrate external model zoos, such as HuggingFace Hub. To help users quickly get started, the frameworks provide predefined specification files for commonly used models. We can roughly classify the large models into three categories: Transformer-like LLMs (e.g., Llama, Orion, Baichuan, ChatGLM, Falcon), Mixture-of-Expert(e.g., Mixtral, Mistral, DeepSeek), Multi-modal LLMs (e.g., LLaVA). However, not all large models included in external model zoos can be smoothly supported, because the frameworks integrate new algorithms with a lag. Therefore, users should refer to the model zoo provided by the framework, and make sure that the target model has no additional underlying system requirements before importing a new model from the external model zoo that is beyond the supported model list.

3.2. System Support for Quantization

In practical implementations, it is perplexing that some quantization algorithms, although reducing the bitwidth of weight or activation, do not lead to faster inference. Therefore, a critical question comes into mind: *How do quantization actually achieve real acceleration and storage saving?* To answer this question, we first need to clarify the data transmission process involved in model inference.

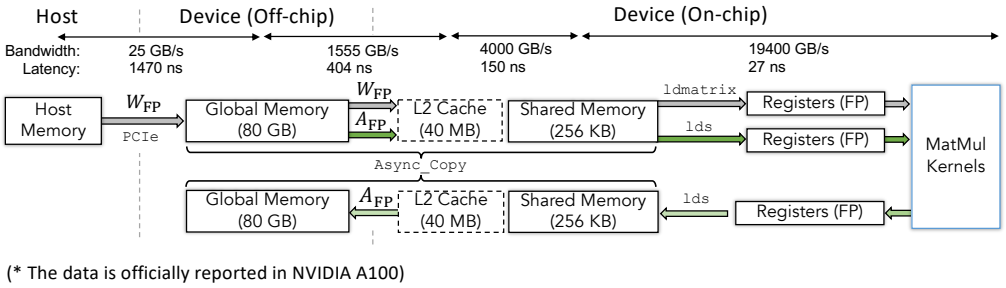


Figure 4: Data transmission of weight and activation in the caching system during inference. The bandwidth and latency are officially reported by NVIDIA A100 as an example.

The data transmission process of weight and activation in the multi-level caching system is outlined in Figure 4, which shows the general dataflow of quantized LLMs. GPUs typically use a hierarchical cache structure with multiple levels, each has different sizes and IO speeds. On-chip caches (L2 cache, shared memory, and registers) provide faster access but have limited capacity, while off-chip caches (device memory or global memory, host memory) offer more capacity but have slower access speed. Therefore, in today's LLMs inference frameworks, we need to load and compute data in segments with highly parallel single instruction, multiple threads (SIMT) paradigm to ensure an acceptable inference speed.

Host memory → Device memory. For weight, we load one layer's weight from the host memory to the device's global memory. The bandwidth is relatively low, which is 25 GB/s per direction (taking NVIDIA A100 as an example (Smith, 2020)). If quantized, it is always in a compact format, thus the time can be saved. The activation is originally generated on the device during inference, which does not need to be copied from the host.

Off-chip memory → On-chip memory. We copy a chunk of weight and activation ready to compute matrix multiplication from the off-chip global memory to the on-chip L2 cache and shared memory. The amount of data copied at a time is basically determined by the design of matrix multiplication (MatMul) kernels, which is always multiple of the number of elements computed in once kernel execution by SIMT. The bandwidth is 1555 GB/s in A100.

Shared memory → Registers. For faster computation, the quantize/dequantize operations and MatMul are always conducted in registers. Therefore, we need to copy the weight and activation from the shared memory to the registers with small pieces. The bandwidth is 19400 GB/s, which requires more than 10 times threads and 1/780 compute intensity of `PCIe`.

Offloading (Registers → shared memory → off-chip memory). The computation results are copied or accumulated to the corresponding elements on shared

memory. After finishing the computation for the chunk of data, the results on shared memory are offloaded to the off-chip memory. The memory that stores the weight and activation of the last chunk can be freed before moving to the next.

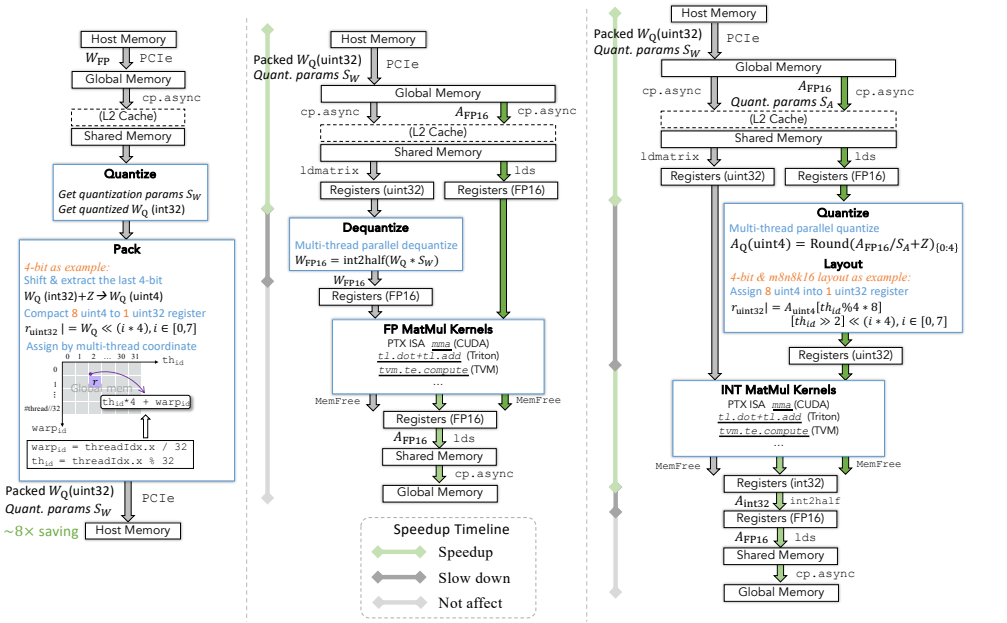
Above, we have clarified the data transmission process by taking the MatMul of a linear layer as an example. Only after then can we answer the question: ***How do quantization reduce the latency and storage?*** To achieve the actual inference acceleration and storage saving, we need comprehensive system support for quantization from the bottom up.

In the following sections, we demonstrate the system supports for quantization according to the action scopes: **Weight-only, Weight & Activation, KV Cache, and Quantization & Dequantization**. We first provide the common and general practices in most frameworks. While these practices may not be the most efficient, they offer high scalability and generalization, allowing new algorithms and implementations to be quickly and easily integrated. Then, we introduce several custom designs. These studies investigate the speedup and generation quality bottlenecks and propose faster solutions for a certain scope. Figure 5 shows how the quantization of weight or activation reduces inference time (4-bit integer quantization is taken as an example, which can also be any other low-bit data format). Figure 6 illustrates how quantized KV Cache affects the inference. Speedup Timelines in both figures clearly divide the whole process into three types based on the time consumption compared to the FP16 counterpart: *Speedup* (green line), *Slow down* (dark grey line), and *Not affect* (light grey line).

3.2.1. Weight-Only Quantization

The fundamental bottleneck in model inference before and after the advent of large models is the data transmission and storage costs, which are always neglected in ordinary small models. Due to the huge amount of data, the transmission latency can not be overlooked, which even surpasses the computation latency and becomes the major challenge in LLM inference. Therefore, weight-only quantization emerges, which compacts the weight and reduces the data copy burden among levels of caches (Lin et al., 2024a; Frantar et al., 2022).

The processes related to weight-only quantization are illustrated in Figure 5 (a) and (b). Both weight-only and weight & activation quantization require packing weight to lower bitwidth beforehand. The weight packing is only conducted once before inference, and it costs little computation resources and time. The weight data are distributed to multi-threads, with each thread tiles a chunk of data according to the following steps: (1) quantizing the weight to lower bitwidth by pre-obtained scaling factors, (2) densely packing them into `uint32` units without idle bits, (3) offloading and storing into host memory. Therefore, the packed weight has a significant reduction in storage compared to the floating-point one.



(a) Prepare (pack weight) (b) Process of weight-only quantization (c) Process of weight & activation quantization

Figure 5: The data transmission process of quantization for (a) Quantized weight preparation (weight pack), (b) Weight-only quantization, and (c) Weight & Activation quantization.

See speedup timeline in (b), weight-only quantization alleviates the burden of data transmission from host memory to on-chip memory by reducing the data amounts. However, it introduces additional dequantization of weight before conducting the MatMul because the general kernels only receive the same datatype of inputs. As long as the time spent on dequantization is shorter than the time saved on data transmission, the weight-only quantization brings benefits in acceleration, which indeed is the case. It is the overload of parameter transmission in LLMs that makes weight-only quantization valuable in practice. Therefore, even using floating-point MatMul kernels, weight-only quantization can still accelerate the inference of LLMs.

As for custom designs, since weight-only quantization dequantizes the weight back to FP16, it is possible to pack the weight with arbitrary bitwidths during quantized weight preparation. Many works propose 3-bit, 5-bit, 6-bit weight quantization (Shi et al., 2024; Frantar et al., 2022; Xia et al., 2024). Furthermore, since the quantized weight must be dequantized to higher bitwidths before MatMul, it is not necessary to design a linear surjection from low bitwidth numbers to real values. In other words, we can map the integers to arbitrary floating-point numbers,

and adopt lookup tables for dequantization (Dotzel et al., 2024a; Dettmers et al., 2024). To make full use of storage and reduce the time of dequantizing weight during inference, researchers design customized backends on specific platforms to support efficient inference. For example, FP6-LLM (Xia et al., 2024) designs a complete GPU kernel to support faster FP6→FP16 dequantization and the dense storage of weight. SpQR (Dettmers et al., 2023) has an efficient decoding backend based on GPUs to deal with the outliers by sparse quantization and achieves load balancing.

3.2.2. Weight & Activation Quantization

Following the traditional practice of quantization, both weight and activation are quantized to low bitwidth, and the MatMul kernels are also implemented by low-bit instructions. We illustrate the speedup timeline in Figure 5(c) that the accelerated processes are weight transmission in the caching system as well as the low-bit MatMul. The extra operations are the quantization for activation from FP16 to low-bit integer before MatMul, and the datatype casting for the results from INT32 to FP16 after MatMul. Weight & activation quantization yields greater acceleration compared to the weight-only quantization because the computationally intensive MatMul usually can be accelerated by lower bitwidth kernels, which use more efficient instructions and a better degree of parallelism. Meanwhile, it is recommended to simplify the complexity of activation quantization to minimize the time spent on runtime quantization. However, the actual speedup ratio highly depends on the hardware design, such as the number of floating-point and integer processing units.

As for custom designs, there are two categories of techniques: (1) Faster Quantization and Dequantization (or datatype conversion). For example, QQQ (Zhang et al., 2024d) proposes faster FP16→INT8 for quantizing activation, INT4→INT8 for dequantizing weight, and INT32→FP16 for casting the MatMul results to accelerate the data format conversion during inference. This work is based on Kim et al. (2022) which firstly introduces a faster INT4→FP16 datatype conversion. Besides speeding up, other approaches turn to remove the process. Tender (Lee et al., 2024b) proposes a decomposed quantization technique to eliminate runtime dequantization/quantization during inference. (2) Faster MatMul Kernel. GEMV can be more flexible and efficient in fitting various bitwidths than GEMM, and even receives input matrices with two bitwidths, such as INT1*INT8 and INT3*INT8 (Wang et al., 2023). By assembling several products of a matrix and a vector, we can get the desired results without padding or idle bits. For example,

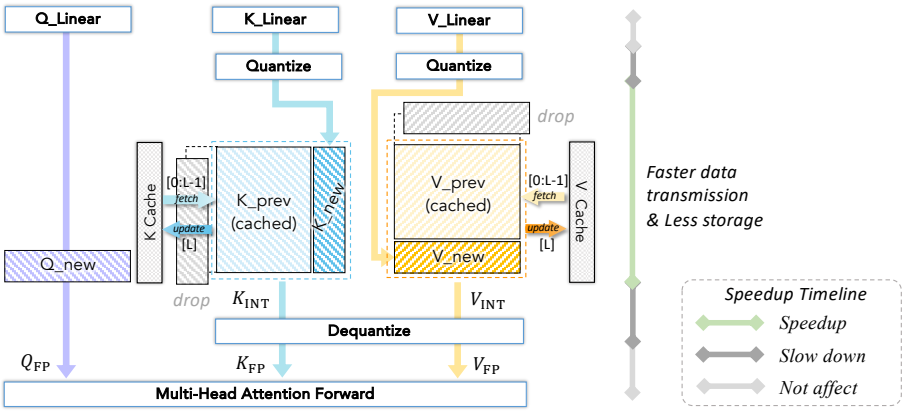


Figure 6: Illustration of KV Cache quantization.

EETQ[24] introduces GEMV operators which are 13-27% faster than GEMM kernel. SqueezeLLM (Kim et al., 2023) proposes LUT-based MatMul by GEMV, which supports highly efficient 4-bit MatMul kernel on hardware architectures that do not support integer MatMul instruction. AQLM (Egiazarian et al., 2024a) designs W1A16 and W2A8 MatMul kernels to receive input matrices with extremely low bitwidth and calculate them directly without dequantizing or datatype conversion.

3.2.3. KV Cache Quantization

KV Cache, or key-value cache, is to optimize the generative models that predict text token by token. Although the model generates only one token at a time, each token depends on the previous context. To avoid repeated calculation, the KV cache acts as a memory bank storing previous key-value results to reuse in the following generations. However, the storage highly depends on the sequence length, hidden size, attention head numbers, and so on. Quantization is an efficient approach to compress the storage. The overall process is illustrated in Figure 6.

The KV cache is generated and updated in runtime along with the serialized input data. During inference, the K_{new} and V_{new} from linear layers are first quantized, then concatenate to the end of the stored key and value lists, which are also quantized, to form new lists. While the earliest key and value will be dropped if the cache size is exceeded. Then we dequantize the matrices to FP16 before conducting multi-head attention forward propagation with the newly generated query Q_{new}. We illustrate how KV cache quantization affects the inference in the

[24]https://github.com/NetEase-FuXi/EETQ

Speedup Timeline. Compared to the FP KV cache, the quantized one occupies less storage in device memory, and spares less time in KV data transmission in the caching system due to the smaller data bytes.

There are mainly three techniques for quantizing KV cache: (1) Quantizing to lower bitwidth. QoQ (Lin et al., 2024b) compresses KV to 4-bit and proposes SmoothAttention to prevent the accuracy drop due to the lower bitwidth. KIVI (Zirui Liu et al., 2023) even developed a tuning-free 2bit KV cache quantization algorithm. Yang et al. (2024) proposes a mixed-precision strategy that quantizes the earliest KV to lower bitwidth, while keeping the new KV with more bits. (2) Quantizing window. Many studies postpone the quantization of KV pairs, but only quantize them in a batch when the length of the full-precision KV list exceeds the window size. (3) Skipping the dequantization of K_{new}. Methods like WKVQuant (Yue et al., 2024) concatenates the FP K_{new} and V_{new} to the dequantized K_{prev} and V_{prev}, which preserves more information of current token in K and V matrices, then quantizing the K_{new} and V_{new} and store into the KV cache (when meets the condition). (4) Optimizing outliers. There are token-wise outliers in KV matrices, so methods such as storing the outliers with higher bitwidth or mitigating the outlier magnitudes can improve the performance (Dong et al., 2024; Liu et al., 2024c; Kang et al., 2024; Lin et al., 2024b). We omit the details of this category, as the general practices are similar to the quantization methods used for the entire model.

3.2.4. Quantization for Production

Besides accelerating the inference, quantization techniques are also used for accelerating the model production of quantized models. Before LLMs emerge, the PTQ methods are regarded as a time-efficient approach. However, some PTQ methods require times of forward propagation to get the parameters, which is neglectable in ordinary models but becomes burdensome in LLMs due to the huge parameter amount. Therefore, accelerating the PTQ calibration becomes a new issue worth exploring. Methods like HQQ (Badri and Shaji, 2023) simplify the calibration by improving the sparsity by L1-norm and enhancing the awareness of the weight distribution. Meanwhile, it applies the closed-form solutions to avoid calculating the gradients. Without the calibration data, HQQ quantizes the model in just a few minutes, which significantly reduces the production time of quantized LLMs. The HQQ+ (Badri and Shaji, 2024) even pushes the bitwidth to the aggressive 1/2-bit.

3.2.5. Quantization and Dequantization

In this section, we roughly categorize the quantization into three types: (1) *Floating-point Quantization*, casting the high-bit floating-points into low-bit ones.

(2) *Integer Quantization*, which mainly refers to dividing the floating-points into evenly spaced integers. We omit requantizing higher bitwidth integers to lower bitwidth ones here, because it is seldom used in real practices, and few studies propose faster implementations to convert integers. (3) *Binarization*, including `sign` and `bool` functions.

Floating-point Quantization. Quantizing higher bitwidth floating-point to lower is actually the clip of mantissa bits. That is because the source value with higher bitwidth usually has more or equal bits for both exponent and mantissa parts compared to the target value with lower bitwidth. Algorithm 3.2.5 provides an example of quantizing FP32 to FP8. And we follow Micikevicius et al. (2022) to summarize the general process as follows:

(1) *Scale.* Since the target value occupies less bitwidth, the representation range may shrink drastically, and not be able to convey most of the data. Scaling the source value to a suitable range can best preserve the information after quantized to FP8. The scaling is pre-obtained by learning or calibration.

(2) *Check Overflow/Underflow.* Check whether the source value overflows FP8 from the upper or lower bound. If so, return the maximum or minimum directly. If it is not overflow, check if the exponent part underflows from the smallest positive normal number that the FP8 can present. If so, we divide the value by the smallest subnormal number in FPx, round to the nearest integer, and then multiply the smallest subnormal number. The integer determines the value of mantissa bits and the exponent bits are all set to zero.

(3) *Copy and Round.* If the value is neither overflows nor underflows of FP8, we copy the lower e bits from the source FP32 value to the target FP8 value. Then we clip the mantissa to m bits by rounding to the nearest. It is notable that rounding and overflow/underflow handling are both crucial for maintaining numerical stability and precision in real applications. However, since the reduction of mantissa bits, precision degradation is inevitable while converting to lower bitwidth.

Floating-point Dequantization. Dequantizing floating-point numbers to higher bitwidth is straightforward. In the FP format system, the bitwidth of both the exponent and mantissa bits in lower bitwidth values will not exceed that in higher bitwidth. Therefore, we can directly extract and copy the sign bit, exponent and mantissa from the original value (with fewer bitwidth) to the most significant bits in the corresponding parts of the target value (with more bitwidth). And then we conduct zero filling on the rest bits for the exponent and mantissa parts [25].

[25] https://github.com/pytorch/pytorch/blob/main/c10/util/Float8_fnuz_cvt.h

Algorithm 1 Quantization to lower-bit floating-point values.

Input: X_{FP32}, $s \in \mathbb{R}^+$, $X_0 \in \mathbb{R}$, $e, m \in \mathbb{Z}^+$, $clip^{min}$, $clip^{max}$
Output: X_{FP8}

$X_{\text{FP32}}^{unscaled} = X_{\text{FP32}}/s$
$e^{min} =$ `-(1 << (`e `- 1)) + 1`
$e^{max} =$ `(1 << (`e `- 1))`
$m = x - e - 1$
// Theoretical maximum of exponent part for FP8
$X_{\text{FP8}}^{e} = e^{max} + 2^{8-1} << 23$
// Theoretical maximum of mantissa part for FP8
$X_{\text{FP8}}^{m} =$ `~(0x007FFFFF >>` m`) & 0x007FFFFF`
$X_{\text{FP8}}^{theomax} = X_{\text{FP8}}^{e} + X_{\text{FP8}}^{m}$
// Check exponent overflow
if $X_{\text{FP32}}^{unscaled} > \min(clip^{max}, X_{\text{FP8}}^{theomax})$ **then**
  $X_{\text{FP8}} = \min(clip^{max}, X_{\text{FP8}}^{theomax})$
else if $X_{\text{FP8}}^{theomax} < \max(clip^{min}, -X_{\text{FP8}}^{theomax})$ **then**
  $X_{\text{FP8}} = \max(clip^{min}, -X_{\text{FP8}}^{theomax})$
else
  $X_{\text{FP8}}^{sign} = X_{\text{FP32}}^{unscaled}$ `& 0x80000000`
  $X_{\text{FP8}}^{e} = X_{\text{FP32}}^{unscaled}$ `& 0x7F800000`
  $X_{\text{FP8}}^{m} = X_{\text{FP32}}^{unscaled}$ `& 0x007FFFFF`
  // Check exponent underflow
  if $(X_{\text{FPx}}^{e} >> 23) - 2^{x-1} < e^{min} + 1$ **then**
    $X_{\text{FP8}.subnorm}^{min} =$ `1 / (1 <<` $((1 << (e - 1)) + m - 2)$`)`
    $X_{\text{FPx}} =$ `round2int`$(X_{\text{FP32}}^{unscaled}/X_{\text{FP8}.subnorm}^{min})X_{\text{FP8}.subnorm}^{min}$
  end if
  // Round mantissa
  $R_m = (X_{\text{FP8}}^{m}$`<<`$m)$ `& 0x007FFFFF + 0x3F800000`
  $R_m =$ `round2int`$(R_m - 1)$
  // Process mantissa
  $X_{\text{FP8}}^{m} = (X_{\text{FP8}}^{m}$`>>`$(23 - m) + R_m)$`<<`$(23 - m)$
  $X_{\text{FP8}} = X_{\text{FP8}}^{sign} + X_{\text{FP8}}^{e} + X_{\text{FP8}}^{m}$
end if
return X_{FP8}

Integer Quantization. We first scale the floating-point numbers to the representation span of INTk by dividing the scaling factor $s \in \mathbb{R}^+$, and adding a zero-point $Z \in \mathbb{Z}$ to shift the clamped range (Wu et al., 2020). round$(\cdot)$ is the round-to-the-nearest function, and clamp$(\cdot, q^{\min}, q^{\max})$ restricts values to be within the representation span of k-bit with $q^{\min} = -2^{k-1}, q^{\max} = 2^{k-1} - 1$ in symmetric quantization and $q^{\min} = 0, q^{\max} = 2^k - 1$ in asymmetric quantization. Therefore, the overall quantization formulation can be written as

$$X_{\text{INT}_k} = \text{clamp}\left(\text{round}\left(\frac{X_{\text{FP}}}{s}\right) + Z, q^{\min}, q^{\max}\right), \tag{8}$$

where the scaling factor s can be initialized as $s_0 = (q^{\max} - q^{\min})/(X_{\text{FP}}^{\max} - X_{\text{FP}}^{\min})$, where $X_{\text{FP}}^{\max}$ and $X_{\text{FP}}^{\min}$ are the maximum and minimum values.

For system support, many frameworks apply the Marlin quantization [26] (Frantar and Alistarh, 2024) as the standard process. The pseudocode Algorithm 2 outlines the steps involved in Marlin quantization, and uses 4-bit integer quantization as an example. The values are quantized and stored as unsigned integers with the desired bitwidth. Extra pre/post-shift will be conducted to get the signed values. Therefore, we first scale the X_{FP32} values by s and round it to integers. Then, adding 2^{k-1} to shift the values to non-negative integers within the span of uINT4 (4-bit unsigned integer). We omit the detail of C++ built-in datatype casting function `float2uint`. To be understood, we explain the packing process in lines 4 to 8 by double **for** loops, which is actually implemented as lines 9 to 11. In 4-bit quantization, every 8 values are packed as a single uINT32, and the quantized matrix size is a quarter of the original. By using $i::8$, we abstract every 8 values along C starting with i, incrementing by 8, and ending by default (till the end of dimension C). And then left shifting the values by $4 * i$ to place the 4-bit value to the corresponding bit range, and leave $4 * i$ zeros on the right, allowing previously-stored quantized values to be preserved after `OR` operation.

There are several custom algorithms that introduce faster data type conversions. QQQ (Zhang et al., 2024d) designs a faster FP16 to INT8 conversion, named `FastFP16toINT8`. It starts by shifting the FP16 values to the representation span of uINT8 by adding 128. Next, adding an additional 1024 which effectively converts and places the 8 bits of uINT8 into the lower segment of the FP16 mantissa. Finally, the lower 8 bits from FP16 are extracted and applied with an `XOR` operation with `0x80` to obtain the desired INT8 format. The overall process can be further simplified to `FMA`, `PRMT`, and `XOR` operations in practice.

[26] https://github.com/IST-DASLab/marlin

Algorithm 2 Marlin quantization from FP32 to INT4.

Input: $X_{\text{FP32}} \in \mathbb{R}^{T,C}$, $s \in \mathbb{R}^{+}$
Output: X_{uINT4}

1: $X_{\text{FP32}}^{\text{round}} \leftarrow \texttt{round}(X_{\text{FP32}}/scale)$
// Shift to span of uint4
2: $X_{\text{FP32}}^{\text{clamp}} \leftarrow \texttt{clamp}(X_{\text{FP32}}^{\text{round}} + 2^3, 0, 2^4 - 1)$
3: $X_{\text{uint32}} \leftarrow \texttt{float2uint}\ (X_{\text{FP32}}^{\text{clamp}})$
// Pack every 8 X_{uint32} to a single uINT32
4: **for** $k \leftarrow 0$ **to** C//8 **do**
5: **for** $i \leftarrow 0$ **to** 7 **do**
6: $X_{\text{INT4}}[:, k]_{(4*i+3:4*i)} \leftarrow X_{uint32}[:, i + 8 * k] << (4 * i)$
7: **end for**
8: **end for**
// Line 4∼8 can also simplify as line 9∼11
// `i::8` creates a sequence starts at i, increments by 8, and ends by default
9: **for** $i \leftarrow 0$ **to** 7 **do**
10: $X_{\text{uINT4}}[:, :]_{(:4*i)} \leftarrow X_{\text{uint32}}[:, i::8] << (4 * i)$
11: **end for**
12: **return** X_{uINT4}

Integer Dequantization. It means projecting the integers back to the real numbers by multiplying the scaling factors, which can be expressed as

$$\hat{X}_{\text{FP}} = s \cdot (X_{\text{INT}x} - Z) \approx X_{\text{FP}}. \tag{9}$$

Therefore, in many works s can also be initialized by searching from candidates to find an optimal (Wei et al., 2023b):

$$s_{\text{candidate}} = \frac{i}{num_i} s_0, \quad i \in \mathbb{Z}^{+}, i \in (0, num_i) \tag{10}$$

$$s.t.\ \min ||X_{\text{FP}} - \hat{X}_{\text{FP}}||_p. \tag{11}$$

where num_i means the number of candidates, which is always set as 50, 100 and so on (Yuan et al., 2024; Wei et al., 2023b). s can also be a learnable parameter (Wei et al., 2023b; Shao et al., 2023). The way to find a better s has been widely studied before LLMs emerged (Ding et al., 2024; Wei et al., 2023a).

For system support, we first unpack the element according to the way we pack them, and then multiply to the corresponding scaling factor, which can be

tensor-wise, channel-wise, token-wise, and other granularities described in Sec. 2.2. Custom implementations are also proposed, `SINT4toS8` Li et al. (2023a) designs a faster conversion from INT4 to INT8 by multiplied by 16.

Binarization. It takes the `sign` or `bool` function to abstract the sign:

$$X_{\texttt{sign}} = \begin{cases} 1, & X_{\text{FP}} \geq 0, \\ -1, & X_{\text{FP}} < 0, \end{cases} \quad X_{\texttt{bool}} = \begin{cases} 1, & X_{\text{FP}} \geq 0, \\ 0, & X_{\text{FP}} < 0. \end{cases} \tag{12}$$

Using `sign` or `bool` depends on the algorithm design, i.e. what value we expect the bits to represent. For example, binarized transformers always use `bool` function on attention scores and the post-ReLU activation. While the weight and activation in linear functions are taken `sign` function. Since the hardware always regards the bits as 0 or 1, we can assemble instructions to achieve any desired matrix multiplication[27]. For example, on NVIDIA GPUs, the `mma` instruction takes 0/1 bit matrices and regards them as 0s and 1s while conducting bitwise accumulation operation `popcount`. Therefore, to obtain the correct accumulation, the `popcount` function is designed with different arithmetic rules, i.e., if it substracts 1 for each 0, we can have the result of `sign` function. It has many accelerated implementations, such as lookup table [28], nifty popcnt (Wilkes et al., 1958), hacker popcnt (Warren, 2012), hakmem popcnt [29] and so on.

Binarization Dequantization. It is simply by multiplying a scaling factor s, i.e. $\hat{X}_{\text{FP}} = s \cdot X_{\texttt{sign/bool}}$ to preserve the magnitude of the original values. It is easy to understand that large amounts of information will be lost in binarization. Therefore, few studies are devoted to binarizing LLMs due to the sharp performance degradation. Due to the significant speedup and storage reduction, it is valuable to dig deeper into binarizing LLMs, but may require a new formulation beyond `sign` and `bool` functions. DB-LLM (Chen et al., 2024a) propose 2-bit weight quantization by decomposing to two 1-bit weight matrices, which can be efficient in MatMul theoretically.

4. Quantization Strategies for Efficient LLM Training

4.1. Low-bit Training

There are different strategies to accelerate the training of Large Language Models (LLMs) using low-bit. The common-used techniques contain BF16, FP16, FP8, and INT8 training.

[27] https://github.com/yifu-ding/BGEMM-CUDA
[28] https://github.com/WojciechMula
[29] https://en.wikipedia.org/w/index.php?title=HAKMEM&oldid=1228234783

FP16 training: Among all the data formats, BF16 training is widely used for LLMs since they are usually stable during training. However, they require hardware (e.g., A100, 4090, H100) support with Ampere or Hopper architectures. For some older hardware like Volta or Turing architectures (e.g., V100, T4), the data format is not available. In these cases, FP16 is often adopted to speed up the training, even for some small computer vision models. Since they have smaller exponent bits, FP16 faces a higher risk of encountering underflow or overflow issues. Therefore, a loss scaling strategy is proposed to preserve small or large gradient magnitudes.

Algorithm 3 Algorithm for Weight Update with FP16 Precision

```
Maintain a primary copy of weights in FP32
while not converged do
    Make an FP16 copy of the weights
    Forward propagation (FP16 weights and activations)
    Multiply the resulting loss with the scaling factor S
    Backward propagation (FP16 weights, activations, and their gradients)

    Multiply the weight gradient with 1/S
    Complete the weight update (including gradient clipping, etc.)
end while
```

FP8 training. Since some hardware vendors like NVIDIA or AMD have designated new architectures supporting FP8 or FP4 formats. To achieve satisfactory acceleration with little modification, we can utilize the library Transformer Engine provided by vendors. While the dynamic range provided by the FP8 types is sufficient to store any particular activation or gradient, it is not sufficient for all of them at the same time. This makes the single loss scaling factor strategy, which worked for FP16, infeasible for FP8 training and instead requires using distinct scaling factors for each FP8 tensor. The scaling process can be formulated as:

$$FP8\_MAX = maximum\_representable\_value(fp8\_format), \tag{13}$$

$$exp = get\_exponent(FP8\_MAX/amax), \tag{14}$$

$$new\_scaling\_factor = 2.0^{exp}. \tag{15}$$

$fp8\_format$ indicates the formats like E4M3 or E5M2. $FP8\_MAX$ is the relevant max value under that format. $amax$ is the maximal absolute value of the tensor. Then we can calculate the $new\_scaling\_factor$ with exp. However, the calculation of $new\_scaling\_factor$ can not be online since it will introduce much more memory

access. The best practice is to employ delayed scaling. This strategy chooses the scaling factor based on the maximums of absolute values seen in some number of previous iterations. This enables the full performance of FP8 computation but requires storing the history of maximums as additional parameters of the FP8 operators.

Institution	Format	Framework	Engine	Hardware
NVIDIA	BF16	Deepspeed, Megatron-LM	AMP	Ampere, Hopper GPUs
NVIDIA	FP16	Deepspeed, Megatron-LM	AMP	Ampere, Hopper, Volta, Turing GPUs
NVIDIA	FP8	Deepspeed, Megatron-LM	Transformer Engine (TE)	Ampere, Hopper GPUs
Intel	FP8	Deepspeed, Megatron-LM	Transformer Engine (TE)	Intel Gaudi 2 AI accelerator
GraphCore	FP8	PyTorch	UnitScaling	Graphcore C600 IPU-Processor PCIe Card

Table 3: Systems for low-bit training.

INT8 training: During training, in addition to the model's weight parameters, it is also necessary to save the gradients required by the optimizer and the backup information of the weights or gradients.

This makes the massive parameter scale of LLMs a more pronounced memory bottleneck during fine-tuning, hindering their deployment in broader application scenarios. INT8 Training (Zhu et al., 2020) is considered a direct method to reduce the memory usage of gradients during training. However, the instability of quantization in backpropagation makes the training of LLMs more unstable and can even lead to crashes. QST (Zhang et al., 2024g) proposes optimizing three key sources of memory usage simultaneously: model weights, optimizer states, and intermediate activations. In addition to quantizing the LLM model weights to 4 bits and introducing a separate side network that uses the LLM's hidden states for task-specific predictions, QST also uses several low-rank adapters and gradient-free downsampling modules to significantly reduce the number of trainable parameters, thereby saving memory on optimizer states. Q-GaLore (Zhang et al., 2024f) points out that GaLore's memory-saving strategy of projecting gradients using SVD incurs significant time costs. To address this, Q-GaLore adaptively updates the gradient subspace based on gradient convergence statistics and keeps the projection matrix in INT4 format and the weights in INT8 format, allowing llama-7b to be trained from scratch on a single 16GB GPU. Jetfire (Xi et al., 2024) features an INT8 data flow to optimize memory access and a per-block quantization method to maintain the accuracy of pretrained transformers. 4-bit Optimizer (Li et al., 2024a) uses a smaller block size and proposes to utilize both row-wise and column-wise information for better quantization, and further identify a zero point problem of quantizing the second moment, solving it with a linear quantizer.

4.2. Quantization Strategies for PEFT

Well-pretrained LLMs possess excellent generalization and exhibit good transferability and adaptability during fine-tuning, making them potentially useful for a variety of downstream tasks. However, the massive parameter scale of LLMs creates a significant memory bottleneck during fine-tuning, hindering their broader application. Thus, the concept of Parameter-Efficient Fine-Tuning(PEFT) is introduced to address the issue of LLM fine-tuning under resource constraints (Ding et al., 2023; Han et al., 2024).

As the demand for fine-tuning LLMs arises, it has been discovered that quantization can reduce memory usage during the fine-tuning process; some improve traditional QAT training, significantly reducing the parameter load during each update, while others class of methods combines quantization with the LoRA fine-tuning approach.

4.2.1. Partial Parameter Fine-Tuning with Quantization

Previous QAT methods require almost the same resources as full parameter training, making them infeasible in resource-constrained fine-tuning scenarios. Therefore, partial parameter fine-tuning strategies have been proposed. PEQA (Kim et al., 2024) follows the naive QAT training approach. But after quantizing the weights W, it obtains scaling factors s_0 and fixed-point numbers $\overline{W}_0$, then it keeps W and only trains s_0. OWQ (Lee et al., 2024a) only updates the high-precision "weak columns" after mixed-precision quantization.

4.2.2. Low-bit Low-Rank Adaptation

Low-Rank Adaptation (LoRA) (Hu et al., 2021) freezes the pre-trained weights and only trains low-rank matrices. Although it reduces the trainable parameters by 10,000 times, it does not decrease the size of the pre-trained model weight itself, thus only reducing the memory requirements for fine-tuning by 3 times.

Methods like QLoRA (Dettmers et al., 2024) represent a typical paradigm that fine-tune the LoRA for quantized LLMs. They first quantize the pre-trained LLM to low bits using PTQ methods:

$$\mathbf{W}_q \leftarrow quant(\mathbf{W}), \tag{16}$$

Where $\mathbf{W}$ is the weight of each layer.

Then, they freeze all weight parameters and update only the LoRA during fine-tuning, with the forward pass as follows:

$$\mathbf{Y} = \mathbf{X} \cdot dequant(\mathbf{W}) + \mathbf{X} \cdot \mathbf{AB}, \tag{17}$$

Where $\mathbf{X}$ is the input of each layer.

In these methods, matrix $\mathbf{A}$ is typically initialized with random Gaussian values, while $\mathbf{B}$ is initialized to all zeros. This approach not only significantly reduces the memory footprint of the model's weight parameters but also ensures that the optimizer only needs to store the gradients of LoRA during fine-tuning, greatly decreasing memory usage. QLoRA (Dettmers et al., 2024) introduces the use of Normal Float for double quantization of $\mathbf{W}$, achieving both good accuracy retention and memory savings, allowing fine-tuning of a 65B pre-trained model using a single 48GB GPU. IR-QLoRA (Qin et al., 2024) incorporates information theory into the QLoRA paradigm, enhancing fine-tuning performance through information calibration and connection. LoRA+ (Hayou et al., 2024) demonstrates that setting different learning rates for matrices A and B in LoRA enables efficient feature learning. QDyLoRA (Rajabzadeh et al., 2024) and Bayesian-LoRA (Meo et al., 2024) employs more flexible rank allocation within LoRA.

Some methods aim to directly obtain a deployable quantized model after fine-tuning. QA-LoRA (Xu et al., 2023) uses INT format to quantize $\mathbf{W}$ and adjusts $\mathbf{X} \cdot \mathbf{A}^{i \times r} \mathbf{B}^{r \times o}$ to $\text{mean}(\mathbf{X}) \cdot \mathbf{A}^{\frac{i}{L} \times r} \mathbf{B}^{r \times o}$, allowing the fine-tuned $\mathbf{AB}$ to be losslessly merged into the INT format $\mathbf{W}_q$, without extra computation when deployment. L4Q (Jeon et al., 2024), on the other hand, maintains the dimension $\mathbf{A} \in \mathbb{R}^{i \times r}$ and directly uses the full QAT forward propagation method, simultaneously updating the quantizer parameters s and b for $\mathbf{A}, \mathbf{B}$, and $\mathbf{W} + \mathbf{AB}$. While L4Q does not reduce the memory footprint of the weights through quantization during pre-training, the optimizer still does not need to retain the gradients of the weights, resulting in a fine-tuned quantized model that can be deployed directly with higher accuracy.

Many methods have recognized that the initialization of LoRA significantly impacts the effectiveness of these quantization-based efficient parameter fine-tuning methods. As a result, they aim to minimize $\|\mathbf{W} - (\mathbf{W}_q + \mathbf{AB})\|_F$ before fine-tuning. LoftQ (Li et al., 2023c) and LQ-LoRA (Guo et al., 2023b) both achieve this through iterative computation: $Q_t \leftarrow quant(\mathbf{W} - \mathbf{A}_{t-1}\mathbf{B}_{t-1}^\top)$ and $\mathbf{A}_t, \mathbf{B}_t \leftarrow SVD(\mathbf{W} - Q_t)$. LQ-LoRA also suggests incorporating calibration data, adjusting the minimization objective to $\left\| \sqrt{F} \odot (\mathbf{W} - (\mathbf{W}_q + \mathbf{AB})) \right\|_F^2$, where F is the Fisher information matrix for $\mathbf{W}$, and $\odot$ represents the Hadamard product. Additionally, LQ-LoRA introduces dynamic quantization configurations to better adapt to resource constraints.

Figure 7 is an illustration of different LoRA structures. Figure 7(a) represents methods like QLoRA that do not alter any part of the LLM during the fine-tuning stage and keep the complete original LoRA structure (Dettmers et al., 2024; Qin et al., 2024; Hayou et al., 2024; Li et al., 2023c). Figure 7(b) represents methods like QA-LoRA that also do not change any part of the LLM during fine-tuning

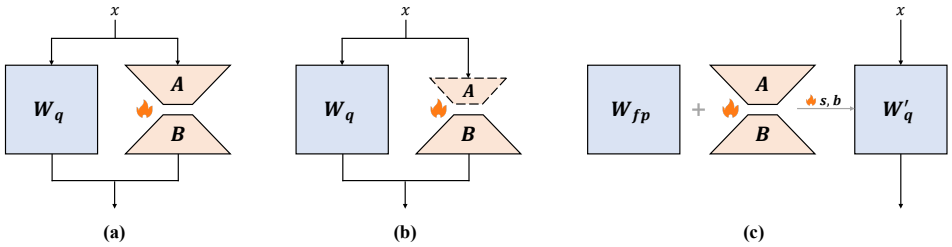


Figure 7: Illustrations for different LoRA structures.

but modify the original LoRA structure (Xu et al., 2023). Figure 7(c) represents methods like L4Q that modify the original LoRA structure and use a training process similar to QAT (Jeon et al., 2024). Both (a) and (b) require only the quantized LLM weights W_q during fine-tuning, while (c) needs to store the pre-trained full-precision weights W_{fp}. (a) is solely intended to reduce training costs and cannot directly produce a quantized model after fine-tuning, while both (b) and (c) can directly integrate the LoRA module after fine-tuning to produce a deployable quantized model.

5. Quantization Algorithms for Efficient LLM Inference

This section navigates through the algorithms of LLM quantization. Quantization algorithms can be broadly divided into two primary approaches: Quantization-Aware Training (QAT) and Post-Training Quantization (PTQ). QAT integrates quantization into the training/fine-tuning process, enabling the model to learn and adapt to the quantization constraints, thereby minimizing the accuracy loss associated with lower precision. In contrast, in the scenario of PTQ, we are given a pre-trained floating-point model along with a small amount of calibration data, aiming to generate an accurate quantized model without an end-to-end training process. We will delve into these quantization algorithms in detail. By the end of this section, we hope our survey can serve as a thorough and systematic collection of the various quantization algorithms applicable to LLMs, their implementation strategies, and their implications for model performance and efficiency.

5.1. Quantization-Aware Training

Table 4 summarizes the different QAT methods for LLMs. LLM-QAT (Liu et al., 2023b) is the pioneering work that investigates the QAT for LLMs. To overcome the training data limits, it proposes data-free knowledge distillation which aligns the teacher logits of full-precision models and student logits of quantized

Algorithms	Category	Target bits	Dataset	Train. time	Affiliation
LLM-QAT	W-A-KV Quant.	4/8-bit W-A-KV	Date-free	medium	Meta
BitDistiller	W-only Quant.	2/3-bit W	Alpaca, Evol-Instruct-Code, WikiText-2, MetaMathQA	fast	HKUST, SJTU, Microsoft
EfficientQAT	W-only Quant.	2/3/4-bit W	RedPajama	fast	OpenGVLab, HKU
BitNet	W-A Quant.	1-bit W, 8/16-bit A	Pile, Common Crawl snapshots RealNews, CC-Stories	slow	Microsoft, UCAS, THU
BitNet b1.58	W-A Quant.	Ternary W, 8/16-bit A	RedPajama	slow	Microsoft, UCAS

Table 4: Comparison of different QAT methods.

models. Following LLM-QAT, BitDistiller (Du et al., 2024) employs the asymmetric clipping strategy for asymmetric quantization during the self-distillation stage. EfficientQAT (Chen et al., 2024b) significantly reduces the training cost by splitting the QAT into two consecutive phases. The first phase optimizes all parameters for each block and then the second phase merely optimizes quantization parameters for the entire network. To pave the way for a new era of extreme quantization level, BitNet (Wang et al., 2023) replaces the BitLinears with original Linears and trains from scratch. Its variant, BitNet b1.58 (Ma et al., 2024a), leverages ternary weight for each parameter which achieves near-lossless performance.

5.2. Post-Training Quantization

Post-Training Quantization (PTQ) is a technique that applies quantization to a pre-trained model. Unlike QAT, PTQ does not require the model to be trained with quantization modules. This makes PTQ a highly practical approach for deploying models that were originally trained with high precision. PTQ is particularly useful when access to the training data is limited or when retraining is computationally expensive. Therefore, with the development of the LLMs, the past few years have witnessed a remarkable surge in PTQ algorithms because of their small training cost.

To have a better introduction, we systematically divide PTQ algorithms into several categories, as described in Figure 8.

5.2.1. Equivalent Transformation

Many studies (Luo et al., 2020; Bondarenko et al., 2021; Wei et al., 2023b; Xiao et al., 2023) have highlighted the presence of significant outliers in LLMs. These outliers pose substantial challenges for quantization, as they force a large number of normal values to be represented with a limited number of bits, which leads to large quantization errors and accuracy degradation. Therefore, a multitude of algorithms have emerged in recent years, aiming to mitigate the issue of outliers in LLMs.

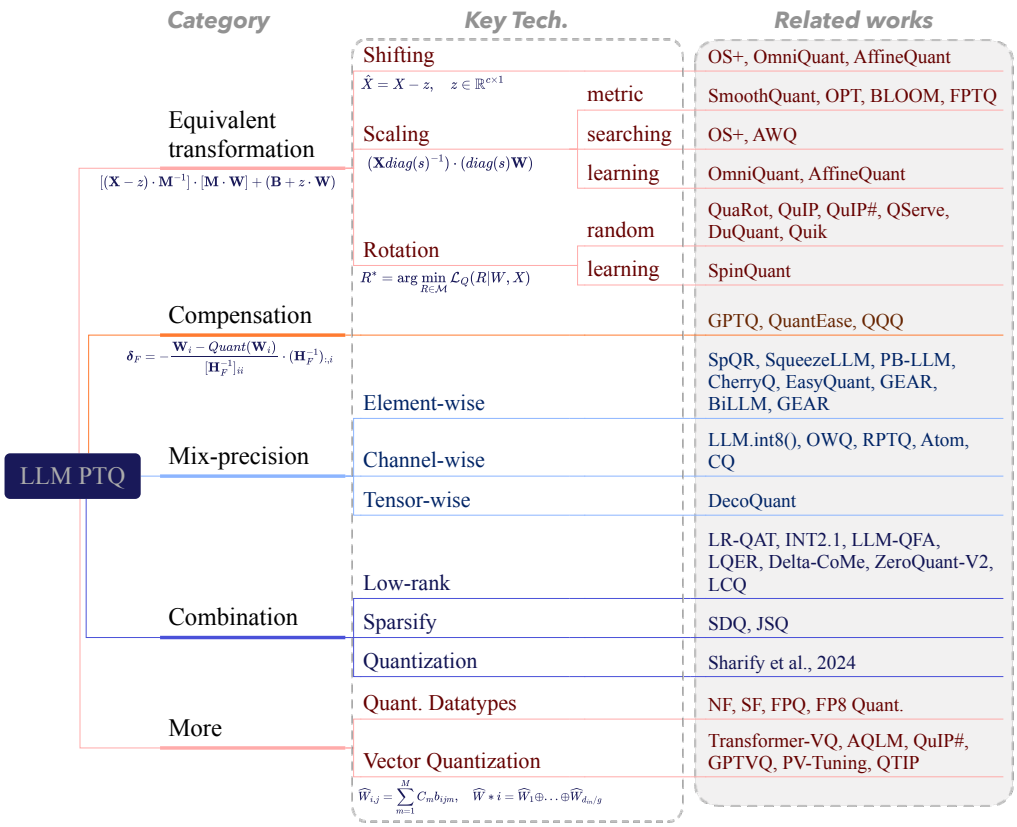


Figure 8: An overview of the PTQ algorithms.

Among all the algorithms addressing the outlier problem, equivalent transformation is one of the most representative and effective methods. Most equivalent transformation methods alleviate the impact of outliers on quantization by making the outliers in weights or activations more symmetrical and smooth, which can be formulated as follows:

$$\begin{aligned}\mathbf{Y}&=\mathbf{X}\mathbf{W}+\mathbf{B}\\&=[(\mathbf{X}-z)\cdot\mathbf{M}^{-1}]\cdot[\mathbf{M}\cdot\mathbf{W}]+(\mathbf{B}+z\cdot\mathbf{W}),\end{aligned}\tag{18}$$

where z is a shifting factor used to make the distribution of outliers in the input symmetrical, and $\mathbf{M}$ is a matrix used to make the distribution smoother. By adopting the aforementioned equivalent transformation, many existing quantization methods have achieved state-of-the-art (SOTA) performance under various quantization settings and scenarios.

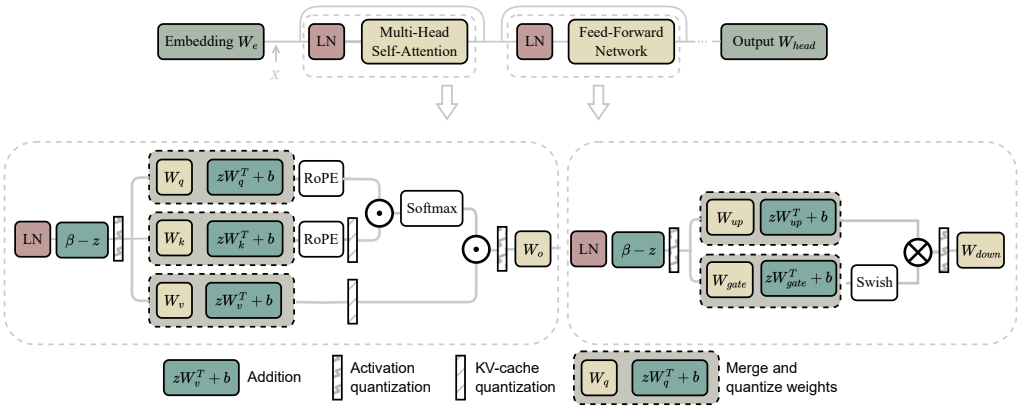


Figure 9: Overall diagram of shifting transformation. z can be merged into the parameter β in Layernorm and the weight metrices.

Based on the implementation, equivalent transformation can be further subdivided into three types: shifting transformation, scaling transformation, and rotation transformation. We then independently provide a detailed introduction for each type.

Shifting Transformation. Outliers in LLMs are asymmetrically distributed across different channels. This asymmetrical representation can cause a tensor composed of channels with small ranges to exhibit a very large overall range, resulting in difficulty in the quantization process. To address this issue, OS+ (Wei et al., 2023b) first proposes the channel-wise shifting transformation, which adjusts activations across channels to mitigate the impact of asymmetry as the following equation:

$$\hat{X} = X - z, \tag{19}$$

where $z \in \mathbb{R}^{c \times 1}$ serves as a row vector and shifts each channel of the activations. Note that this operation is not the conventional shifting operation used in symmetric quantization. Instead, it operates on a channel-wise basis and provides a better distribution for per-tensor quantization. In detail, OS+ defined z in a handicraft-way:

$$z_j = \frac{max(\mathbf{X}_{:,j}) + min(\mathbf{X}_{:,j})}{2}. \tag{20}$$

With the channel-wise shifting in place, the tensor range is reduced to the largest channel range, eliminating the influence of asymmetric outliers. However,

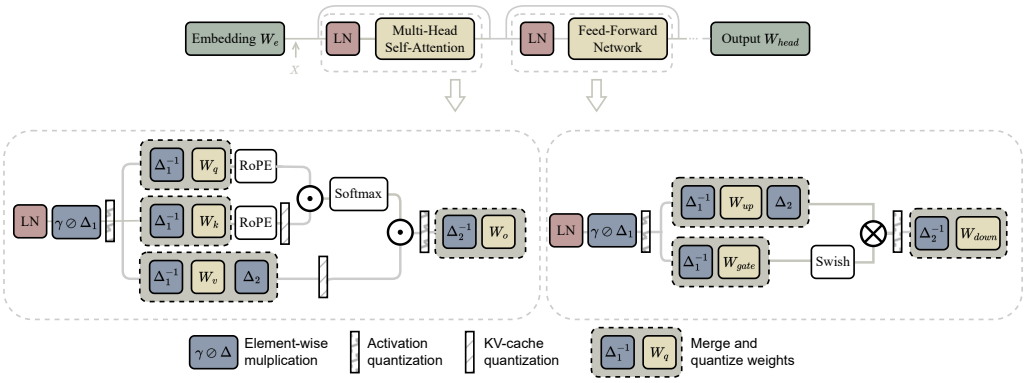


Figure 10: Overall diagram of scaling transformation. Δ can be merged into the parameter γ in Layernorm and the weight metrices.

handcrafting the equivalent parameters leads to sub-optimal results. Hence OmniQuant (Shao et al., 2023) is proposed to determine the optimal shifting parameters in a differentiable way by including the block-wise quantization error minimization:

$$\arg\min_{\Theta} ||\mathcal{F}(\mathbf{W}, \mathbf{X}) - \mathcal{F}\big(Q_w\left(\mathbf{W}; \Theta\right), Q_a\left(\mathbf{X}; \Theta\right)\big)||, \tag{21}$$

where $\mathcal{F}$ represents the mapping function for a transformer block in the LLM, $Q_w(\cdot)$ and $Q_a(\cdot)$ denote the weight and activation quantizer respectively, Θ is the shifting parameter. Block-wise minimization is easy to optimize with minimal resource requirements. Therefore, by optimizing the objective function block by block, a more effective shifting vector can be obtained compared to the direct computation used in OS+ in an efficient and resource-saving way. However, OmniQuant requires fine-tuning of the learnable parameters; otherwise, issues such as gradient explosion can easily occur. Similar to OmniQuant, AffineQuant (Ma et al., 2024b) also adopts a learning-based shifting operation.

We illustrate the diagram of shifting transformation as shown in Fig. 9. The shifting factor z can be fused in LayerNorm and weight matrix so that no further overhead is needed.

Scaling Transformation. Shifting transformation effectively addresses the issue of asymmetrical distribution of outliers in activations, reducing the large range caused by the asymmetry. However, this only aids per-tensor quantization and does not reduce the difficulty of per-channel quantization, as it does not fundamentally eliminate the outliers distributed across channels in the activations. To further reduce the impact of outliers on quantization, SmoothQuant (Xiao et al., 2023) initially proposes to use a scaling transformation. It relies on a key ob-

servation: although activations are much difficult to quantize than weights due to the presence of outliers, different tokens exhibit similar variations across their channels (Dettmers et al., 2022a). Based on this observation, SmoothQuant migrates the quantization difficulty from activations to weights offline by introducing a mathematically equivalent per-channel scaling transformation that significantly smooths the magnitudes across channels:

$$\mathbf{Y} = (\mathbf{X}diag(s)^{-1}) \cdot (diag(s)\mathbf{W}) = \hat{\mathbf{X}}\hat{\mathbf{W}}, \tag{22}$$

where s is a smoothing factor. Note that $diag(s)$ corresponds to the matrix M in Equation 18, but it is a diagonal matrix used to achieve per-channel smoothing. SmoothQuant introduces a hyper-parameter α as the migration strength to control how much difficulty to migrate from activation to weights, using the following equation:

$$s_j = \frac{max(|\mathbf{X}_j|)^{\alpha}}{max(|\mathbf{W}_j|)^{1-\alpha}}. \tag{23}$$

However, this method requires multiple trials to determine the optimal migration strength for different models, i.e., $\alpha = 0.5$ is a well-balanced point for all OPT (Zhang et al., 2022) and BLOOM (Le Scao et al., 2023) models.

Inspired by SmoothQuant, FPTQ (Li et al., 2023b) argues that it is unnecessary to consider weights for computing the activation smoothing scale while it is crucial to retain all the activation values with a non-linear lossless mapping. This mapping needs to fit two criteria: (1) touching gently with the inliers and (2) harshly suppressing the outliers. Based on this, they adopt a logarithmic function to improve the calculation of the smooth matrix s:

$$s_j = \frac{max(|\mathbf{X}_j|)}{log_2(2 + max(\mathbf{X}_j))}. \tag{24}$$

In addition to FPTQ, many other works have followed the approach of SmoothQuant. Both OS+ and AWQ (Lin et al., 2024a) use searching-based methods to find the smooth scale. However, the optimization objectives and search spaces of the two methods differ. The optimization objective of OS+ is:

$$s^* = \arg\min_{s} \mathbb{E}||Q\big((\mathbf{X}-\Delta)\cdot diag(s)^{-1}\big)Q\big(diag(s)\cdot\mathbf{W}^{\mathsf{T}}\big) + \hat{\mathbf{b}} - (\mathbf{X}\mathbf{W}^{\mathsf{T}} + \mathbf{b})||_F^2. \tag{25}$$

To simplify the search space, OS+ optimizes the outlier threshold t, compressing channels with an activation range over $t$into $(-t, t)$ and leaving others unchanged. This reduces the problem to a single variable. A grid search is then

used for t to minimize the objective. After finding the optimal t, the scaling vector is calculated as follows:

$$s_j = max(1.0, \frac{max(\mathbf{X}_{:,j} - \Delta_j)}{t}). \tag{26}$$

AWQ finds that the saliency of weight channels is actually determined by the activation scale. To this end, it adopts an activation-awareness optimization objective and uses a very simple search space:

$$s = {s_x}^{\alpha}, \quad s^* = \arg\min_{\alpha} ||Q\big(\mathbf{W} \cdot diag(s)\big)(diag(s)^{-1} \cdot \mathbf{X}) - \mathbf{WX}||, \tag{27}$$

where s_x is the average magnitude of activation (per channel), and use a single hyper-parameter α to balance between the protection of salient and non-salient channels.

In addition to searching-based methods, some approaches use learning-based techniques to find the optimal scaling matrix. OmniQuant and AffineQuant also learn the scaling matrix. In Equation 21, OmniQuant learns both the shifting factor Δ and the scaling matrix $diag(s)$. However, OmniQuant optimizes only within the range of a diagonal matrix. AffineQuant (Ma et al., 2024b) argues that this limited search range can lead to significant quantization errors, reducing the generalizability of the quantization method in low-bit scenarios. It proposes learning a general invertible matrix to perform equivalent affine transformations on weights and activations, achieving better results.

We also illustrate the diagram of scaling transformation. The same as shifting transformation, scaling factor Δ can be merged into layernorm and weight matrix.

Rotation Transformation. Rotation transformation was first introduced by QuIP (Chee et al., 2024). QuIP is based on the insight that quantization works better when the weight and Hessian matrices are incoherent. This means that the weights should have similar magnitudes and the directions that require precise rounding should not align with the coordinate axes. To make it straight, a weight matrix is μ-incoherent if:

$$max(\mathbf{W}) \leq \mu ||\mathbf{W}||_F / \sqrt{mn}, \tag{28}$$

where mn is the number of the matrix elements. QuIP shows that multiplying a weight matrix on the left and right by an orthogonal matrix can reduce incoherence, which is equal to performing a rotation transformation on the weight matrix. QuIP utilizes Kronecker-structured orthogonal matrices, allowing for rapid additional computations. Building on this, QuIP# (Tseng et al., 2024a) replaces these with Hadamard matrices, enhancing quantization through better incoherence and

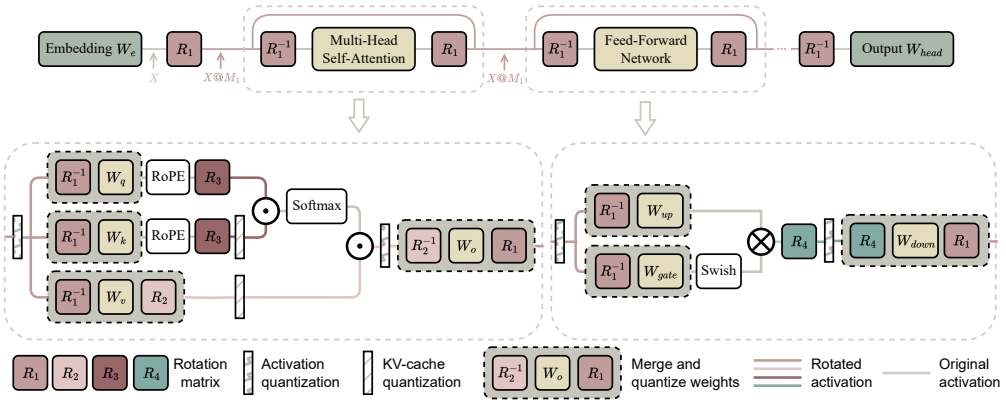


Figure 11: Overall diagram of the rotation transformation. The rotated activations exhibit fewer outliers and are easier to quantize. R_1 and R_2 are randomized matrices which can be merged into the weights matrices. R_3 and R_4 can not be merged and are usually Hadamard matrices.

speeding up the forward pass, as the Hadamard transform can be computed in $\mathcal{O}(nlogn)$ addition operations.

Both of these two methods target weight-only quantization. Following these approaches, QuaRot (Ashkboos et al., 2024) introduces a weight&activation quantization method that also quantizes the KV cache. QuaRot operates in two stages. First, the model weights are manipulated in full precision, and two Hadamard operations are added to the model's forward pass. In the second stage, the weights are quantized using an existing method, and quantization operations are integrated into the forward pass for online activation quantization.

However, both the orthogonal matrices in QuIP and the Hadamard matrices in QuIP# and QuaRot are randomly generated. Although these works have shown that these randomly generated matrices can alleviate the outlier problem to some extent, they are not optimal. SpinQuant (Liu et al., 2024e) finds that the performance of a quantized network can vary significantly with different rotation matrices. For example, the average accuracy on downstream zero-shot reasoning tasks can fluctuate by up to 13 points depending on the rotation used on the MMLU benchmark. Therefore SpinQuant proposes a learning-based rotation transformation. The rotation matrix is learned using the Cayley SGD method, with the following optimization objective:

$$R^* = \arg\min_{R \in \mathcal{M}} \mathcal{L}_Q(R|W, X). \tag{29}$$

Here, $\mathcal{M}$ presents the Stiefel manifold, i.e., the set of all orthogonal matrices. $\mathcal{L}_Q(\cdot)$ denotes the task loss. By employing the learned matrix, the performance

is improved significantly and the variance becomes much smaller compared with randomized matrices. The diagram in SpinQuant (Liu et al., 2024e) effectively illustrates the overall process of the rotation transformation, so we have borrowed it for our purposes as shown in Figure 11.

We can observe that scaling transformation and rotation transformation can be utilized for the different parts of LLM quantization. Qserve (Lin et al., 2024b) is a co-designed quantization system for efficient LLM serving, combining scaling and rotation transformations. Where additional overhead is required for the online computation of rotation matrices, Qserve uses scale transformation as a substitute for rotation operations, thereby avoiding the extra overhead.

5.2.2. Compensation

The weight compensation technique, originally stemming from Optimal Brain Damage (OBD) (LeCun et al., 1989), involves a Taylor series expansion of the objective function. This method assumes that upon the removal of any given parameter, the influence of the remaining parameters on the objective function remains unchanged. Based on OBD, OBS (Hassibi et al., 1993) and OBQ (Frantar and Alistarh, 2022) calculate the impact of each parameter weight on the objective function by solving the inverse Hessian matrix. Concurrently, they compute a compensation term applied to the remaining weights to offset the error introduced by each weight adjustment.

Although one-by-one weight quantization methods have achieved satisfactory performance on smaller models, the computational overhead becomes prohibitive when scaling to larger models. To accelerate quantization, GPTQ (Frantar et al., 2022) quantizes the weights column-by-column, and the rounding errors are compensated using second-order information. Specifically, this algorithm compensates for the quantization error induced by the quantized weights $Quant(\mathbf{W}_i)$ by adjusting the subset F of full-precision weights F with an update $\boldsymbol{\delta}_F$:

$$\mathbf{W}_i = \underset{\mathbf{W}_i}{\operatorname{argmin}} \frac{(Quant(\mathbf{W}_i) - \mathbf{W}_i)^2}{[\mathbf{H}_F^{-1}]_{ii}}, \tag{30}$$

$$\boldsymbol{\delta}_F = -\frac{\mathbf{W}_i - Quant(\mathbf{W}_i)}{[\mathbf{H}_F^{-1}]_{ii}} \cdot (\mathbf{H}_F^{-1})_{:,i} \tag{31}$$

where the Hessian matrix is $\mathbf{H}_F = 2\mathbf{X}_F\mathbf{X}_F^\top$. Based on GPTQ, several works have been successively proposed. QuantEase (Behdin et al., 2023) utilizes the Coordinate Descent to compute more precise compensation for the unquantized weights. QQQ (Zhang et al., 2024e) adopts the GPTQ for the transferred weights by OS+ (Wei et al., 2023b).

5.2.3. Mix-precision

As aforementioned, the presence of outliers is widely found in the activations and weights of large language models, which poses a significant challenge for quantization. Consequently, the motivation of numerous mixed-precision methods for LLMs is to represent a small number of outlier values in higher precision and other values in lower precision separately. Similarly, depending on the granularity of mixed precision, methods can be categorized into element-wise, channel-wise, and tensor-wise approaches, as described in 2.2.

Element-wise. SpQR (Dettmers et al., 2023) was the first to demonstrate that outliers also exist in weights. It identifies and isolates these outlier weights based on their sensitivity, saving them as a highly sparse, higher-precision matrix. SqueezeLLM (Kim et al., 2023) adopts non-uniform quantization for non-salient weights, which achieves near-lossless performance. Similarly, CherryQ (Cui and Wang, 2024) defines heterogeneity to identify the critical cherry parameters. To explore extreme compression rate, PB-LLM (Shang et al., 2023) is the first to binarize the non-salient weights in LLMs. Since PB-LLM still allocates high precision to 10%-30% of salient weights, BiLLM (Huang et al., 2024) employs residual approximation for salient weights and group quantization for non-salient weights, reducing the quantization bit-width of LLM weights to 1.08 bits. GEAR (Kang et al., 2024) extends the concept of mixed precision to the KV cache compression and utilizes low-rank matrices to approximate the quantization residuals.

Channel-wise. LLM.int8() (Dettmers et al., 2022a) splits the weights and activations into two independent parts according to the outlier channels to minimize output quantization errors in activations, which effectively reduces the GPU memory usage during inference. OWQ (Lee et al., 2024a) proposes a sensitivity-aware mixed-precision scheme to identify the weak columns by Hessian metric. Furthermore, OWQ also provides weak column tuning (WCT) to enable accurate parameter-efficient fine-tuning for task-specific adaptation. RPTQ (Yuan et al., 2023) observe the varying ranges across channels in activations pose challenges for quantization. Therefore, RPTQ reorders the channels into different clusters with respective quantization. Atom (Zhao et al., 2024) employs dynamic reorder for activations and static reorder for weights to remain aligned with the corresponding activation channels. Atom further quantizes the KV cache to 4-bit which significantly boosts serving throughput. Inspired by information theory, CQ (Zhang et al., 2024b) couples multiple key/value channels together and jointly quantize them.

Tensor-wise. DecoQuant (Liu et al., 2024b) aims to migrate the quantization difficulties by performing the tensor decomposition to local tensors. (Li et al., 2024b) investigates the weight bits among different blocks, experts, and linear layers, which reveals the varying numbers of weight bits are effective.

5.2.4. Combination

Although current quantization methods for large models have achieved relatively good results, their performance under extremely high compression rates is still unsatisfactory due to the limited representation capacity of low-bit quantization. Currently, commonly used compression methods include low-rank decomposition, model sparsification, and model distillation.

Low-rank. Although QAT is generally considered to offer the best accuracy, its high memory cost makes it difficult to apply to LLMs. Therefore, some methods consider introducing LoRA or other matrix decomposition methods as a trade-off between PTQ and QAT. Unlike PEFT discussed in Section 3.3, these methods aim to reduce quantization error using techniques like LoRA or SVD to achieve a quantized model closer to the full-precision model, rather than enhancing learning ability on fine-tuning datasets. Some works have used LoRA to achieve parameter-efficient QAT. LR-QAT (Bondarenko et al., 2024) computes $s \cdot \text{clamp}(\mathbf{W}_q + \mathbf{A}^{i \times r}\mathbf{B}^{r \times o})$ during the forward pass and does not update $\mathbf{W}$ during the backward pass, allowing a 7B LLM to be trained on a single consumer-grade GPU with 24GB of memory. This approach results in a quantization-friendly model after fine-tuning. LLM-QFA aims to produce models with various bit widths through a single supernet training, significantly reducing the resource overhead of this production method by leveraging the low resource cost of LoRA. INT2.1 (Chai et al., 2023) utilizes LoRA to shift the optimization target from minimizing per-layer or per-block quantization error to minimizing the overall output error of the model. Through end-to-end fine-tuning, it reduces the distance between the output distribution and its corresponding original full-precision counterpart. Other works have reduced quantization errors through matrix decomposition. LQER (Zhang et al., 2024a) applies SVD to quantization errors and uses an activation-induced scaling matrix to guide the singular value distribution toward the desired pattern. Delta-CoMe (Ping et al., 2024) discovers that the singular values of delta weights exhibit a long-tailed distribution after applying SVD, and proposed a mixed-precision delta quantization method that uses high-bit representations for the singular vectors corresponding to these singular values. ZeroQuant-V2 (Yao et al., 2023) introduced an optimized low-rank compensation method that enhances model quality recovery by leveraging a low-rank matrix obtained through SVD of the quantization errors. LCQ (Cai and Li, 2024) uses low-rank codebooks with a rank greater than one for quantization, addressing the issue of accuracy loss when using rank-one codebooks under high compression ratios.

Sparsification. Model sparsification aims to remove unimportant weights to accelerate the model, while quantization further reduces the remaining weights using lower-bit representations. Therefore, the two methods can be effectively used in

a complementary manner. SDQ (Jeong et al., 2024) first sparsifies the weights of LLMs based on the magnitude. as much as possible until the quality of the LLM is significantly impacted (e.g., a 1% increase in perplexity). Then it utilizes a mix-precision quantization method to deal with the outliers. However, this method does not take into account the coupling of the two approaches. Sparsification and quantization often conflict with each other. Sparsification tends to preserve parameters with large absolute values in LLMs (Han et al., 2015; Sun et al., 2023), while quantization prefers a smaller range of parameter values (Wei et al., 2023b). As a result, the parameters preserved during sparsification may degrade the performance of quantization. JSQ (Guo et al., 2024) design a new sparsity metric to address this issue:

$$\begin{aligned} \mathbf{I}_{ij} &= ||\mathbf{X}||_2 \cdot ||\mathbf{W}||, \\ \mathbf{A}_{ij} &= max(\hat{\mathbf{Y}}_{:i}) - min(\hat{\mathbf{Y}}_{:i}), \\ &\quad where \quad \hat{\mathbf{Y}} = \mathbf{X} \cdot (\Phi(\mathbf{W}; i; j))^{\mathsf{T}}, \\ \mathbf{S}_{ij} &= \mathbf{I}_{ij} + \lambda \mathbf{A}_{ij}. \end{aligned} \tag{32}$$

Here, $\Phi(\mathbf{W}; i; j)$ denotes an auxiliary weight matrix when setting the element at ith row and jth column as 0 in $\mathbf{W}$. λ is a trade-off factor. By using this metric, a better trade-off between preserving outliers for more information and minimizing the activation range for better quantization can be achieved.

Quantization. In addition to combining quantization with other compression methods, different quantization techniques can also be integrated to achieve better results. A recent work (Sharify et al., 2024) combines the SmoothQuant and GPTQ together. Actually, most of the equivalent transformation methods and the compensation quantization methods are orthogonal which can be merged for further exploration.

5.2.5. More Quantization Forms

Beyond integer quantization, more forms of quantization are being introduced for LLMs, as they can also compress the average bit-width of a 32-bit or 16-bit model down to 4 or lower bits. While these methods do not always offer significant acceleration benefits when saving memory, they generally lead to improvements in precision.

More Quantization Datatypes. Integer quantization typically assigns a single scaling factor to an entire block and quantizes each element individually into an integer number. This reduces memory usage while also enabling the acceleration of fixed-point operations after quantizing both weights and activations. However, as higher precision is demanded for LLM quantization, formats that better match the

original distribution of values have been proposed. Normal Float (Dettmers et al., 2021, 2024), proposed alongside Quantile Quantization, is based on the assumption that the weight distribution follows a normal distribution. It is considered an information-theoretically optimal data type that ensures each quantization bin has an equal number of values assigned from the input tensor. However, Dotzel et al. (Dotzel et al., 2024b) conducted a statistical analysis and found that the distributions of most LLM weights and activations follow a Student's t-distribution. Based on this, they derived a new theoretically optimal format, Student Float (SF4). Floating-Point (FP) quantization offers better hardware support compared to NF/SF and is more flexible than integer quantization, allowing it to more effectively handle long-tail or bell-shaped distributions. Since FP can support flexible allocation of exponent and mantissa bits, several allocation schemes have been proposed. FPQ (Liu et al., 2023a) determines FP quantizers through a joint format and max value search combined with a pre-shifted exponent bias. FP8 Quantization (Kuzmin et al., 2022) tests various allocation schemes by evaluating metrics like quantization error and proposes FP8 quantization simulation for learnable allocation and quantization.

Vector Quantization. Vector Quantization (VQ) quantizes multiple vector dimensions jointly. It achieves this by learning codebooks $C_1, ..., C_M$, each containing 2^B vectors (for B-bit codes) . To encode a given database vector, VQ splits it into sub-groups of entries, and then encodes every group by choosing a vector from the learned codebook. A part of the weights of i-th layer is encoded by choosing a single code from each codebook and summing them up:

$$\widehat{W}_{i,j} = \sum_{m=1}^{M} C_m b_{ijm}$$

where $b_{ijm} \in \mathbb{R}^{2^B}$ represents a one-hot code for the i-th output unit, j-th group of input dimensions and m-th codebook.

To represent the full weights of i-th layer, simply concatenate:

$$\widehat{W} * i = \widehat{W}_1 \oplus ... \oplus \widehat{W}_{d_{in}/g}$$

where $\oplus$ denotes concatenation.

Transformer-VQ (Lingle, 2023) applies vector quantization (VQ) to the key vector sequence of Attention, reducing the complexity of Attention to linear. Most other VQ works focus on optimizing the codebooks $C_m \in \mathbb{R}^{2^B}$, and the discrete codes represented by one-hot b. AQLM (Egiazarian et al., 2024b) learns additive quantization of weight matrices in an input-adaptive fashion and jointly optimizes codebook parameters across each transformer block. QuIP# (Egiazarian et al.,

Benchmark	Algorithms	Model family	Tracks	Device	Organ.
LMQuant	AWQ, Atom, GPTQ, QoQ, QuaRot, SmoothQuant	Yi, llama, llama2, llama3, mistral, mixtral	Perplexity, Throughput	NVIDIA GPU	MIT EECS
llmc	AWQ, AdaDim, DGQ, GPTQ, HQQ, LLM.int8(), NormTweaking, OS+, OWQ, OmniQuant, QUIK, SmoothQuant, SpQR	LLaVA, OPT, Qwen, llama3, mixtral	Perplexity, Calibration time, Deployable	NVIDIA GPU	Beihang & SenseTime
MI-optimize	AWQ, FP8, GPTQ, QuIP, RTN, SmoothQuant, SpQR, ZeroQuant, combinations	Baichuan2, chatglm, llama2	BOSS (Robust), Perplexity, ceval, hellaswag, mmlu, piqa, winogrande	NVIDIA GPU	TsingmaoAI
qllm-eval	AWQ, SmoothQuant	LLaVA, OPT, bloomz, chatglm, falcon, gemma, llama, llama2, longchat, mamba, mistral, vicuna	Dialogue, Emergent, Long-Context, OpenCompass, Trustworthiness	NVIDIA GPU	Tsinghua University
gpt-fast	GPTQ	llama, mixtral	Throughput	AMD GPU, NVIDIA GPU	PyTorch
FastChat	AWQ, ExLlama, GPTQ	Alpaca, llama, vicuna	c4	AMD GPU, Metal, NVIDIA GPU	http://lmsys.org/
OpenLLM	AWQ, GPTQ, SqeezeLLM	llama, mistral, mixtral	c4	AMD GPU, NVIDIA GPU	BentoML

Table 5: Quantization toolkits and benchmarks for large language models.

2024b) uses vector quantization to exploit the spherical sub-Gaussian distribution inherent in incoherent weights by introducing a hardware-efficient codebook based on the highly symmetrical E8 lattice. GPTVQ (van Baalen et al., 2024) interleaves the quantization of one or more columns with updates to the remaining unquantized weights, using information from the Hessian of the per-layer output reconstruction MSE, and further compresses the codebooks by using integer quantization and SVD-based compression. PV-Tuning (Malinovskii et al., 2024) notes that using straight-through estimators (STE) leads to suboptimal results and proposes an alternating iterative optimization strategy for scales, codebooks, zeros (continuous parameters), and assignments (discrete codes) during fine-tuning. QTIP (Tseng et al., 2024b) uses a stateful decoder that separates the codebook size from the bitrate and effective dimension to achieve ultra-high-dimensional quantization.

5.3. Quantization Toolkit and Benchmark

5.3.1. Overall Process

To quantize the LLMs, there are always three basic strategies, quantization aware-training (QAT), post-training quantization (PTQ), and parameter-efficient fine-tuning (PEFT).

Table 5 shows that for quantization toolkits and benchmarks, most are based on PyTorch and run on NVIDIA GPUs. Few of them support other hardware, for example, GPT-fast, FastChat, and OpenLLM provide official procedures for AMD GPUs and Metal (for Apple). The quantization benchmarks that are devoted to providing comprehensive comparisons have good support for the prevailing models and quantization algorithms in various aspects. Most benchmarks include well-known models like the Llama series, Mixtral, Vicuna, and so on. Those who pay more attention to the model diversity, such as qllm-eval, have further support for various models. As well as the algorithms, LLMC, LMQuant, and MI-optimize

focus on the performance of different quantization algorithms, and provide uniform, fair, comprehensive tracks for comparisons. All the benchmarks leave interfaces for users to define and evaluate any custom models and algorithms easily.

5.3.2. Tracks

The tracks in the benchmarks showcase the most interesting aspects of quantization LLMs, i.e., efficiency and generation quality. We list the detailed tracks in Table 5. **For efficiency**, the inference efficiency is measured by deployability and throughput, which are the most crucial features in LLMs compression Lin et al. (2024b); Gong et al. (2024). Typically, reducing the storage of the parameters can speed up the inference theoretically, but it depends on the actual system implementation. The benchmark provides us with a fair and convenient probe to distinguish the algorithms and implementations that have practical acceleration and storage saving. The production efficiency is measured by calibration time, which indicates the time and computational resources cost of the PTQ algorithms Gong et al. (2024). Methods that spare lots of resources usually have better generation quality, while those that require less time may have worse generation performance. It is a trade-off in producing quantized LLMs. **For generation quality**, it has many aspects, such as perplexity, accuracy, logic, completion, trustworthiness and so on Lin et al. (2024b); Gong et al. (2024); Li et al. (2024c); Liu et al. (2024d). Most benchmarks evaluate emergent capability, which is the key feature of LLMs. Specifically, models and algorithms that are tested under diverse scenarios, like the dialogue, long-context, or multi-task Li et al. (2024c). And some benchmarks are aware of the safety of generative contents, and estimate the trustworthiness and robustness of LLMs Li et al. (2024c); Liu et al. (2024d).

6. Future Trends and Directions

As the field of large language model quantization continues to evolve, several emerging trends and research directions are poised to shape its future. This section explores the anticipated advancements in quantization techniques, model architectures, and hardware design that will drive improvements in the efficiency, performance, and application of quantized models.

Quantization Techniques. Despite progress, several challenges remain in quantization techniques. Firstly, one major issue is the unclear origin of outliers in large language models (LLMs), which presents a significant barrier to further reducing quantization bit widths. Research aimed at uncovering the internal mechanisms behind these outliers is crucial and will provide valuable insights for the community, potentially advancing the state of quantization and enabling more efficient models. Secondly, pushing the boundaries of minimal bit representation

with acceptable accuracy is highly valuable. Achieving the lowest possible bit width while maintaining performance can fully leverage hardware capabilities and maximize its potential. Thirdly, exploring unified strategies for mixed-bit quantization, including both bit selection and intra-layer/inter-layer bit allocation, is essential for optimizing model performance and efficiency. Current methods primarily emphasize intra-layer mixed precision, often overlooking the potential benefits of inter-layer mixed precision. Last but not least, developing semantic-guided strategies for achieving even lower-bit quantization and compression of key-value (KV) caches will be a major focus. During inference with long context lengths, the primary bottleneck often lies in the substantial memory usage of KV caches. Therefore, identifying effective methods for compressing KV caches is crucial for overcoming this limitation and enhancing model efficiency.

Model Architecture. Innovations in model architecture will also play a pivotal role. Firstly, quantizing models that handle multiple modalities will be explored to ensure efficiency across diverse data types and applications. Secondly, research will expand to include quantization strategies for new and emerging model structures such as the Mixture of Experts (MOE) and other large-scale architectures. Third, exploring the relationship between quantization and model size will provide insights into optimizing smaller models for performance while managing quantization trade-offs.

Hardware Design. Advancements in hardware and quantization co-design will be essential for unlocking new potential. The first area of focus is the development of systems for new types of extremely low-bit quantization. Innovative formats for low-bit representation and efficient system implementations may offer new solutions to the challenges posed by Moore's Law. The second area involves accelerating training with lower-bit precision, such as FP4. Research into hardware that supports training with such low-bit precision will be essential for speeding up model training while preserving performance.

7. Conclusions

In this survey, we have presented an in-depth exploration of low-bit quantization techniques for large language models (LLMs), highlighting their significance in addressing the computational and memory challenges associated with deploying these models in constrained environments. We began by elucidating the fundamentals of low-bit quantization, including the novel data formats and granularities that cater specifically to LLMs. Our review of systems and frameworks has illustrated the diverse approaches and tools available for supporting low-bit LLMs across different hardware platforms. We have also categorized and discussed various techniques for optimizing training and inference, providing a comprehensive

understanding of current methodologies. Lastly, we have explored future directions and emerging trends in the field, emphasizing potential research areas and technological advancements that could further enhance the efficiency and effectiveness of LLM quantization. As the landscape of LLM research continues to evolve, this survey aims to serve as a valuable resource for advancing the development of low-bit quantization techniques,

References

, 2019. Ieee standard for floating-point arithmetic. IEEE Std 754-2019 (Revision of IEEE 754-2008) , 1–84doi:10.1109/IEEESTD.2019.8766229.

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

van Baalen, M., Kuzmin, A., Nagel, M., Couperus, P., Bastoul, C., Mahurin, E., Blankevoort, T., Whatmough, P., 2024. Gptvq: The blessing of dimensionality for llm quantization. arXiv preprint arXiv:2402.15319 .

Badri, H., Shaji, A., 2023. Half-quadratic quantization of large machine learning models. URL: https://mobiusml.github.io/hqq_blog/.

Badri, H., Shaji, A., 2024. Towards 1-bit machine learning models. URL: https://mobiusml.github.io/1bit_blog/.

Bai, H., Hou, L., Shang, L., Jiang, X., King, I., Lyu, M.R., 2021. Towards efficient post-training quantization of pre-trained language models. URL: https://arxiv.org/abs/2109.15082, arXiv:2109.15082.

Behdin, K., Acharya, A., Gupta, A., Keerthi, S., Mazumder, R., 2023. Quantease: Optimization-based quantization for language models–an efficient and intuitive algorithm. arXiv preprint arXiv:2309.01885 .

Bondarenko, Y., Del Chiaro, R., Nagel, M., 2024. Low-rank quantization-aware training for llms. arXiv preprint arXiv:2406.06385 .

Bondarenko, Y., Nagel, M., Blankevoort, T., 2021. Understanding and overcoming the challenges of efficient transformer quantization. arXiv preprint arXiv:2109.12948 .

Cai, W.P., Li, W.J., 2024. Lcq: Low-rank codebook based quantization for large language models. arXiv preprint arXiv:2405.20973 .

Chai, Y., Gkountouras, J., Ko, G.G., Brooks, D., Wei, G.Y., 2023. Int2. 1: Towards fine-tunable quantized large language models with error correction through low-rank adaptation. arXiv preprint arXiv:2306.08162 .

Chee, J., Cai, Y., Kuleshov, V., De Sa, C.M., 2024. Quip: 2-bit quantization of large language models with guarantees. Advances in Neural Information Processing Systems 36.

Chen, H., Lv, C., Ding, L., Qin, H., Zhou, X., Ding, Y., Liu, X., Zhang, M., Guo, J., Liu, X., Tao, D., 2024a. Db-llm: Accurate dual-binarization for efficient llms. URL: https://arxiv.org/abs/2402.11960, arXiv:2402.11960.

Chen, M., Shao, W., Xu, P., Wang, J., Gao, P., Zhang, K., Qiao, Y., Luo, P., 2024b. Efficientqat: Efficient quantization-aware training for large language models. arXiv preprint arXiv:2407.11062 .

Cui, W., Wang, Q., 2024. Cherry on top: Parameter heterogeneity and quantization in large language models. arXiv preprint arXiv:2404.02837 .

Dettmers, T., Lewis, M., Belkada, Y., Zettlemoyer, L., 2022a. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. Advances in Neural Information Processing Systems 35, 30318–30332.

Dettmers, T., Lewis, M., Belkada, Y., Zettlemoyer, L., 2022b. Llm.int8(): 8-bit matrix multiplication for transformers at scale. URL: https://arxiv.org/abs/2208.07339, arXiv:2208.07339.

Dettmers, T., Lewis, M., Shleifer, S., Zettlemoyer, L., 2021. 8-bit optimizers via block-wise quantization. arXiv preprint arXiv:2110.02861 .

Dettmers, T., Pagnoni, A., Holtzman, A., Zettlemoyer, L., 2024. Qlora: Efficient finetuning of quantized llms. Advances in Neural Information Processing Systems 36.

Dettmers, T., Svirschevski, R., Egiazarian, V., Kuznedelev, D., Frantar, E., Ashkboos, S., Borzunov, A., Hoefler, T., Alistarh, D., 2023. Spqr: A sparse-quantized representation for near-lossless llm weight compression. arXiv preprint arXiv:2306.03078 .

Ding, N., Qin, Y., Yang, G., Wei, F., Yang, Z., Su, Y., Hu, S., Chen, Y., Chan, C.M., Chen, W., et al., 2023. Parameter-efficient fine-tuning of large-scale pre-trained language models. Nature Machine Intelligence 5, 220–235.

Ding, Y., Feng, W., Chen, C., Guo, J., Liu, X., 2024. Reg-ptq: Regression-specialized post-training quantization for fully quantized object detector, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 16174–16184.

Dong, S., Cheng, W., Qin, J., Wang, W., 2024. Qaq: Quality adaptive quantization for llm kv cache. URL: https://arxiv.org/abs/2403.04643, arXiv:2403.04643.

Dotzel, J., Chen, Y., Kotb, B., Prasad, S., Wu, G., Li, S., Abdelfattah, M.S., Zhang, Z., 2024a. Learning from students: Applying t-distributions to explore accurate and efficient formats for llms. arXiv preprint arXiv:2405.03103 .

Dotzel, J., Chen, Y., Kotb, B., Prasad, S., Wu, G., Li, S., Abdelfattah, M.S., Zhang, Z., 2024b. Learning from students: Applying t-distributions to explore accurate and efficient formats for llms. URL: https://arxiv.org/abs/2405.03103, arXiv:2405.03103.

Du, D., Zhang, Y., Cao, S., Guo, J., Cao, T., Chu, X., Xu, N., 2024. Bitdistiller: Unleashing the potential of sub-4-bit llms via self-distillation. arXiv preprint arXiv:2402.10631 .

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al., 2024. The llama 3 herd of models. arXiv preprint arXiv:2407.21783 .

Egiazarian, V., Panferov, A., Kuznedelev, D., Frantar, E., Babenko, A., Alistarh, D., 2024a. Extreme compression of large language models via additive quantization. URL: https://arxiv.org/abs/2401.06118, arXiv:2401.06118.

Egiazarian, V., Panferov, A., Kuznedelev, D., Frantar, E., Babenko, A., Alistarh, D., 2024b. Extreme compression of large language models via additive quantization. arXiv preprint arXiv:2401.06118 .

Frantar, E., Alistarh, D., 2022. Optimal brain compression: A framework for accurate post-training quantization and pruning. Advances in Neural Information Processing Systems 35, 4475–4488.

Frantar, E., Alistarh, D., 2024. Marlin: a fast 4-bit inference kernel for medium batchsizes. https://github.com/IST-DASLab/marlin.

Frantar, E., Ashkboos, S., Hoefler, T., Alistarh, D., 2022. Gptq: Accurate post-training quantization for generative pre-trained transformers. arXiv preprint arXiv:2210.17323 .

Gong, R., Yong, Y., Gu, S., Huang, Y., Zhang, Y., Liu, X., Tao, D., 2024. Llm-qbench: A benchmark towards the best practice for post-training quantization of large language models. arXiv preprint arXiv:2405.06001 .

Guo, C., Tang, J., Hu, W., Leng, J., Zhang, C., Yang, F., Liu, Y., Guo, M., Zhu, Y., 2023a. Olive: Accelerating large language models via hardware-friendly outlier-victim pair quantization, in: Proceedings of the 50th Annual International Symposium on Computer Architecture, pp. 1–15.

Guo, C., Zhang, C., Leng, J., Liu, Z., Yang, F., Liu, Y., Guo, M., Zhu, Y., 2022. Ant: Exploiting adaptive numerical data type for low-bit deep neural network quantization, in: 2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO), IEEE. pp. 1414–1433.

Guo, H., Greengard, P., Xing, E.P., Kim, Y., 2023b. Lq-lora: Low-rank plus quantized matrix decomposition for efficient language model finetuning. arXiv preprint arXiv:2311.12023 .

Guo, J., Wu, J., Wang, Z., Liu, J., Yang, G., Ding, Y., Gong, R., Qin, H., Liu, X., 2024. Compressing large language models by joint sparsification and quantization, in: Forty-first International Conference on Machine Learning.

Han, S., Mao, H., Dally, W.J., 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. arXiv preprint arXiv:1510.00149 .

Han, Z., Gao, C., Liu, J., Zhang, S.Q., et al., 2024. Parameter-efficient fine-tuning for large models: A comprehensive survey. arXiv preprint arXiv:2403.14608 .

Hassibi, B., Stork, D.G., Wolff, G.J., 1993. Optimal brain surgeon and general network pruning, in: IEEE international conference on neural networks, IEEE. pp. 293–299.

Hayou, S., Ghosh, N., Yu, B., 2024. Lora+: Efficient low rank adaptation of large models. arXiv preprint arXiv:2402.12354 .

Henry, G., Tang, P.T.P., Heinecke, A., 2019. Leveraging the bfloat16 artificial intelligence datatype for higher-precision computations. URL: `https://arxiv.org/abs/1904.06376`, `arXiv:1904.06376`.

Heo, J.H., Kim, J., Kwon, B., Kim, B., Kwon, S.J., Lee, D., 2023. Rethinking channel dimensions to isolate outliers for low-bit weight quantization of large language models. arXiv preprint arXiv:2309.15531 .

Holmes, C., Tanaka, M., Wyatt, M., Awan, A.A., Rasley, J., Rajbhandari, S., Aminabadi, R.Y., Qin, H., Bakhtiari, A., Kurilenko, L., He, Y., 2024. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference. URL: https://arxiv.org/abs/2401.08671, arXiv:2401.08671.

Hooper, C., Kim, S., Mohammadzadeh, H., Mahoney, M.W., Shao, Y.S., Keutzer, K., Gholami, A., 2024. Kvquant: Towards 10 million context length llm inference with kv cache quantization. arXiv preprint arXiv:2401.18079 .

Hu, E.J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W., 2021. Lora: Low-rank adaptation of large language models. arXiv preprint arXiv:2106.09685 .

Huang, W., Liu, Y., Qin, H., Li, Y., Zhang, S., Liu, X., Magno, M., Qi, X., 2024. Billm: Pushing the limit of post-training quantization for llms. arXiv preprint arXiv:2402.04291 .

Jeon, H., Kim, Y., Kim, J.j., 2024. L4q: Parameter efficient quantization-aware training on large language models via lora-wise lsq. arXiv preprint arXiv:2402.04902 .

Jeong, G., Tsai, P.A., Keckler, S.W., Krishna, T., 2024. Sdq: Sparse decomposed quantization for llm inference. arXiv preprint arXiv:2406.13868 .

Kang, H., Zhang, Q., Kundu, S., Jeong, G., Liu, Z., Krishna, T., Zhao, T., 2024. Gear: An efficient kv cache compression recipefor near-lossless generative inference of llm. arXiv preprint arXiv:2403.05527 .

Kim, J., Lee, J.H., Kim, S., Park, J., Yoo, K.M., Kwon, S.J., Lee, D., 2024. Memory-efficient fine-tuning of compressed large language models via sub-4-bit integer quantization. Advances in Neural Information Processing Systems 36.

Kim, S., Hooper, C., Gholami, A., Dong, Z., Li, X., Shen, S., Mahoney, M.W., Keutzer, K., 2023. Squeezellm: Dense-and-sparse quantization. arXiv preprint arXiv:2306.07629 .

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

Krishnamoorthi, R., 2018. Quantizing deep convolutional networks for efficient inference: A whitepaper. URL: https://arxiv.org/abs/1806.08342, arXiv:1806.08342.

Kuzmin, A., Van Baalen, M., Ren, Y., Nagel, M., Peters, J., Blankevoort, T., 2022. Fp8 quantization: The power of the exponent. Advances in Neural Information Processing Systems 35, 14651–14662.

Le Scao, T., Fan, A., Akiki, C., Pavlick, E., Ilić, S., Hesslow, D., Castagné, R., Luccioni, A.S., Yvon, F., Gallé, M., et al., 2023. Bloom: A 176b-parameter open-access multilingual language model .

LeCun, Y., Denker, J., Solla, S., 1989. Optimal brain damage. Advances in neural information processing systems 2.

Lee, C., Jin, J., Kim, T., Kim, H., Park, E., 2024a. Owq: Outlier-aware weight quantization for efficient fine-tuning and inference of large language models, in: Proceedings of the AAAI Conference on Artificial Intelligence, pp. 13355–13364.

Lee, J., Lee, W., Sim, J., 2024b. Tender: Accelerating large language models via tensor decomposition and runtime requantization. URL: `https://arxiv.org/abs/2406.12930`, `arXiv:2406.12930`.

Lee, J.H., Kim, J., Kwon, S.J., Lee, D., 2023. Flexround: Learnable rounding based on element-wise division for post-training quantization, in: International Conference on Machine Learning, PMLR. pp. 18913–18939.

Li, B., Chen, J., Zhu, J., 2024a. Memory efficient optimizers with 4-bit states. Advances in Neural Information Processing Systems 36.

Li, P., Jin, X., Cheng, Y., Chen, T., 2024b. Examining post-training quantization for mixture-of-experts: A benchmark. arXiv preprint arXiv:2406.08155 .

Li, Q., Meng, R., Li, Y., Zhang, B., Li, L., Lu, Y., Chu, X., Sun, Y., Xie, Y., 2023a. A speed odyssey for deployable quantization of llms. URL: `https://arxiv.org/abs/2311.09550`, `arXiv:2311.09550`.

Li, Q., Zhang, Y., Li, L., Yao, P., Zhang, B., Chu, X., Sun, Y., Du, L., Xie, Y., 2023b. Fptq: Fine-grained post-training quantization for large language models. arXiv preprint arXiv:2308.15987 .

Li, S., Ning, X., Wang, L., Liu, T., Shi, X., Yan, S., Dai, G., Yang, H., Wang, Y., 2024c. Evaluating quantized large language models. arXiv preprint arXiv:2402.18158 .

Li, Y., Xu, S., Lin, M., Cao, X., Liu, C., Sun, X., Zhang, B., 2024d. Bi-vit: Pushing the limit of vision transformer quantization, in: Proceedings of the AAAI Conference on Artificial Intelligence, pp. 3243–3251.

Li, Y., Yu, Y., Liang, C., He, P., Karampatziakis, N., Chen, W., Zhao, T., 2023c. Loftq: Lora-fine-tuning-aware quantization for large language models. arXiv preprint arXiv:2310.08659 .

Lin, J., Tang, J., Tang, H., Yang, S., Chen, W.M., Wang, W.C., Xiao, G., Dang, X., Gan, C., Han, S., 2024a. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. Proceedings of Machine Learning and Systems 6, 87–100.

Lin, Y., Tang, H., Yang, S., Zhang, Z., Xiao, G., Gan, C., Han, S., 2024b. Qserve: W4a8kv4 quantization and system co-design for efficient llm serving. arXiv preprint arXiv:2405.04532 .

Lingle, L.D., 2023. Transformer-vq: Linear-time transformers via vector quantization. arXiv preprint arXiv:2309.16354 .

Liu, A., Feng, B., Wang, B., Wang, B., Liu, B., Zhao, C., Dengr, C., Ruan, C., Dai, D., Guo, D., et al., 2024a. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. arXiv preprint arXiv:2405.04434 .

Liu, P., Gao, Z.F., Zhao, W.X., Ma, Y., Wang, T., Wen, J.R., 2024b. Unlocking data-free low-bit quantization with matrix decomposition for kv cache compression. arXiv preprint arXiv:2405.12591 .

Liu, R., Bai, H., Lin, H., Li, Y., Gao, H., Xu, Z., Hou, L., Yao, J., Yuan, C., 2024c. Intactkv: Improving large language model quantization by keeping pivot tokens intact. arXiv preprint arXiv:2403.01241 .

Liu, S.y., Liu, Z., Huang, X., Dong, P., Cheng, K.T., 2023a. Llm-fp4: 4-bit floating-point quantized transformers. arXiv preprint arXiv:2310.16836 .

Liu, Y., Meng, Y., Wu, F., Peng, S., Yao, H., Guan, C., Tang, C., Ma, X., Wang, Z., Zhu, W., 2024d. Evaluating the generalization ability of quantized llms: Benchmark, analysis, and toolbox. arXiv preprint arXiv:2406.12928 .

Liu, Z., Oguz, B., Zhao, C., Chang, E., Stock, P., Mehdad, Y., Shi, Y., Krishnamoorthi, R., Chandra, V., 2023b. Llm-qat: Data-free quantization aware training for large language models. arXiv preprint arXiv:2305.17888 .

Liu, Z., Wang, Y., Han, K., Ma, S., Gao, W., 2022. Instance-aware dynamic neural network quantization, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp. 12434–12443.

Liu, Z., Wu, B., Luo, W., Yang, X., Liu, W., Cheng, K.T., 2018. Bi-real net: Enhancing the performance of 1-bit cnns with improved representational capability and advanced training algorithm, in: Proceedings of the European conference on computer vision (ECCV), pp. 722–737.

Liu, Z., Zhao, C., Fedorov, I., Soran, B., Choudhary, D., Krishnamoorthi, R., Chandra, V., Tian, Y., Blankevoort, T., 2024e. Spinquant–llm quantization with learned rotations. arXiv preprint arXiv:2405.16406 .

Lozhkov, A., Li, R., Allal, L.B., Cassano, F., Lamy-Poirier, J., Tazi, N., Tang, A., Pykhtar, D., Liu, J., Wei, Y., et al., 2024. Starcoder 2 and the stack v2: The next generation. arXiv preprint arXiv:2402.19173 .

Luo, Z., Kulmizev, A., Mao, X., 2020. Positional artefacts propagate through masked language model embeddings. arXiv preprint arXiv:2011.04393 .

Ma, S., Wang, H., Ma, L., Wang, L., Wang, W., Huang, S., Dong, L., Wang, R., Xue, J., Wei, F., 2024a. The era of 1-bit llms: All large language models are in 1.58 bits. arXiv preprint arXiv:2402.17764 .

Ma, Y., Li, H., Zheng, X., Ling, F., Xiao, X., Wang, R., Wen, S., Chao, F., Ji, R., 2024b. Affinequant: Affine transformation quantization for large language models. arXiv preprint arXiv:2403.12544 .

Malinovskii, V., Mazur, D., Ilin, I., Kuznedelev, D., Burlachenko, K., Yi, K., Alistarh, D., Richtarik, P., 2024. Pv-tuning: Beyond straight-through estimation for extreme llm compression. arXiv preprint arXiv:2405.14852 .

Meo, C., Sycheva, K., Goyal, A., Dauwels, J., 2024. Bayesian-lora: Lora based parameter efficient fine-tuning using optimal quantization levels and rank values trough differentiable bayesian gates. arXiv preprint arXiv:2406.13046 .

Micikevicius, P., Stosic, D., Burgess, N., Cornea, M., Dubey, P., Grisenthwaite, R., Ha, S., Heinecke, A., Judd, P., Kamalu, J., et al., 2022. Fp8 formats for deep learning. arXiv preprint arXiv:2209.05433 .

Oklobdzija, V.G., 1994. An algorithmic and novel design of a leading zero detector circuit: Comparison with logic synthesis. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 2, 124–128.

OpenAI, Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F.L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., Avila, R., Babuschkin, I., Balaji, S., Balcom, V., et al., 2024. Gpt-4 technical report. arXiv:2303.08774.

Ping, B., Wang, S., Wang, H., Han, X., Xu, Y., Yan, Y., Chen, Y., Chang, B., Liu, Z., Sun, M., 2024. Delta-come: Training-free delta-compression with mixed-precision for large language models. arXiv preprint arXiv:2406.08903 .

Qin, H., Ding, Y., Zhang, M., Yan, Q., Liu, A., Dang, Q., Liu, Z., Liu, X., 2022. Bibert: Accurate fully binarized bert. arXiv preprint arXiv:2203.06390 .

Qin, H., Ma, X., Zheng, X., Li, X., Zhang, Y., Liu, S., Luo, J., Liu, X., Magno, M., 2024. Accurate lora-finetunsing quantization of llms via information retention. arXiv preprint arXiv:2402.05445 .

Rajabzadeh, H., Valipour, M., Zhu, T., Tahaei, M., Kwon, H.J., Ghodsi, A., Chen, B., Rezagholizadeh, M., 2024. Qdylora: Quantized dynamic low-rank adaptation for efficient large language model tuning. arXiv preprint arXiv:2402.10462 .

Rouhani, B.D., Zhao, R., More, A., Hall, M., Khodamoradi, A., Deng, S., Choudhary, D., Cornea, M., Dellinger, E., Denolf, K., Dusan, S., Elango, V., Golub, M., Heinecke, A., James-Roxby, P., Jani, D., Kolhe, G., Langhammer, M., Li, A., Melnick, L., Mesmakhosroshahi, M., Rodriguez, A., Schulte, M., Shafipour, R., Shao, L., Siu, M., Dubey, P., Micikevicius, P., Naumov, M., Verrilli, C., Wittig, R., Burger, D., Chung, E., 2023. Microscaling data formats for deep learning. URL: `https://arxiv.org/abs/2310.10537`, `arXiv:2310.10537`.

Shang, Y., Yuan, Z., Wu, Q., Dong, Z., 2023. Pb-llm: Partially binarized large language models. arXiv preprint arXiv:2310.00034 .

Shao, W., Chen, M., Zhang, Z., Xu, P., Zhao, L., Li, Z., Zhang, K., Gao, P., Qiao, Y., Luo, P., 2023. Omniquant: Omnidirectionally calibrated quantization for large language models. arXiv preprint arXiv:2308.13137 .

Sharify, S., Xu, Z., Wang, X., et al., 2024. Combining multiple post-training techniques to achieve most efficient quantized llms. arXiv preprint arXiv:2405.07135 .

Shi, S., Zhao, E., Cai, D., Cui, L., Huang, X., Li, H., 2024. Inferflow: an efficient and highly configurable inference engine for large language models. URL: `https://arxiv.org/abs/2401.08294`, `arXiv:2401.08294`.

Smith, R., 2020. Nvidia ampere unleashed: Nvidia announces new gpu architecture, a100 gpu, and accelerator. AnandTech .

Sun, M., Liu, Z., Bair, A., Kolter, J.Z., 2023. A simple and effective pruning approach for large language models. arXiv preprint arXiv:2306.11695 .

Tambe, T., Yang, E.Y., Wan, Z., Deng, Y., Reddi, V.J., Rush, A., Brooks, D., Wei, G.Y., 2019. Adaptivfloat: A floating-point based data type for resilient deep learning inference. arXiv preprint arXiv:1909.13271 .

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al., 2023a. Llama: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971 .

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al., 2023b. Llama 2: Open foundation and fine-tuned chat models. arXiv preprint arXiv:2307.09288 .

Tseng, A., Chee, J., Sun, Q., Kuleshov, V., De Sa, C., 2024a. Quip#: Even better llm quantization with hadamard incoherence and lattice codebooks. arXiv preprint arXiv:2402.04396 .

Tseng, A., Sun, Q., Hou, D., De Sa, C., 2024b. Qtip: Quantization with trellises and incoherence processing. arXiv preprint arXiv:2406.11235 .

Wang, H., Ma, S., Dong, L., Huang, S., Wang, H., Ma, L., Yang, F., Wang, R., Wu, Y., Wei, F., 2023. Bitnet: Scaling 1-bit transformers for large language models. arXiv preprint arXiv:2310.11453 .

Warren, H., 2012. Hacker's Delight. Pearson Education. URL: https://books.google.com.my/books?id=VicPJYM0I5QC. pages 72-73.

Wei, X., Gong, R., Li, Y., Liu, X., Yu, F., 2023a. Qdrop: Randomly dropping quantization for extremely low-bit post-training quantization. URL: https://arxiv.org/abs/2203.05740, arXiv:2203.05740.

Wei, X., Zhang, Y., Li, Y., Zhang, X., Gong, R., Guo, J., Liu, X., 2023b. Outlier suppression+: Accurate quantization of large language models by equivalent and optimal shifting and scaling. arXiv preprint arXiv:2304.09145 .

Wilkes, M.V., Wheeler, D.J., Gill, S., Corbató, F., 1958. The preparation of programs for an electronic digital computer. Reprinted 1984, pages 191–193.

Wu, H., Judd, P., Zhang, X., Isaev, M., Micikevicius, P., 2020. Integer quantization for deep learning inference: Principles and empirical evaluation. URL: https://arxiv.org/abs/2004.09602, arXiv:2004.09602.

Xi, H., Chen, Y., Zhao, K., Zheng, K., Chen, J., Zhu, J., 2024. Jetfire: Efficient and accurate transformer pretraining with int8 data flow and per-block quantization. arXiv preprint arXiv:2403.12422 .

Xia, H., Zheng, Z., Wu, X., Chen, S., Yao, Z., Youn, S., Bakhtiari, A., Wyatt, M., Zhuang, D., Zhou, Z., et al., 2024. Fp6-llm: Efficiently serving large language models through fp6-centric algorithm-system co-design. arXiv preprint arXiv:2401.14112 .

Xiao, G., Lin, J., Seznec, M., Wu, H., Demouth, J., Han, S., 2023. Smoothquant: Accurate and efficient post-training quantization for large language models, in: International Conference on Machine Learning, PMLR. pp. 38087–38099.

Xu, Y., Xie, L., Gu, X., Chen, X., Chang, H., Zhang, H., Chen, Z., Zhang, X., Tian, Q., 2023. Qa-lora: Quantization-aware low-rank adaptation of large language models. arXiv preprint arXiv:2309.14717 .

Yang, J.Y., Kim, B., Bae, J., Kwon, B., Park, G., Yang, E., Kwon, S.J., Lee, D., 2024. No token left behind: Reliable kv cache compression via importance-aware mixed precision quantization. arXiv preprint arXiv:2402.18096 .

Yao, Z., Aminabadi, R.Y., Zhang, M., Wu, X., Li, C., He, Y., 2022. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. URL: https://arxiv.org/abs/2206.01861, arXiv:2206.01861.

Yao, Z., Wu, X., Li, C., Youn, S., He, Y., 2023. Zeroquant-v2: Exploring post-training quantization in llms from comprehensive study to low rank compensation. arXiv preprint arXiv:2303.08302 .

Yuan, Z., Niu, L., Liu, J., Liu, W., Wang, X., Shang, Y., Sun, G., Wu, Q., Wu, J., Wu, B., 2023. Rptq: Reorder-based post-training quantization for large language models. arXiv preprint arXiv:2304.01089 .

Yuan, Z., Xue, C., Chen, Y., Wu, Q., Sun, G., 2024. Ptq4vit: Post-training quantization for vision transformers with twin uniform quantization. URL: https://arxiv.org/abs/2111.12293, arXiv:2111.12293.

Yue, Y., Yuan, Z., Duanmu, H., Zhou, S., Wu, J., Nie, L., 2024. Wkvquant: Quantizing weight and key/value cache for large language models gains more. arXiv preprint arXiv:2402.12065 .

Zhang, C., Cheng, J., Constantinides, G.A., Zhao, Y., 2024a. Lqer: Low-rank quantization error reconstruction for llms. arXiv preprint arXiv:2402.02446 .

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X.V., et al., 2022. Opt: Open pre-trained transformer language models. arXiv preprint arXiv:2205.01068 .

Zhang, T., Yi, J., Xu, Z., Shrivastava, A., 2024b. Kv cache is 1 bit per channel: Efficient large language model inference with coupled quantization. arXiv preprint arXiv:2405.03917 .

Zhang, Y., Yang, F., Peng, S., Wang, F., Pan, A., 2024c. Flattenquant: Breaking through the inference compute-bound for large language models with per-tensor quantization. URL: https://arxiv.org/abs/2402.17985, arXiv:2402.17985.

Zhang, Y., Zhang, P., Huang, M., Xiang, J., Wang, Y., Wang, C., Zhang, Y., Yu, L., Liu, C., Lin, W., 2024d. Qqq: Quality quattuor-bit quantization for large language models. arXiv preprint arXiv:2406.09904 .

Zhang, Y., Zhang, P., Huang, M., Xiang, J., Wang, Y., Wang, C., Zhang, Y., Yu, L., Liu, C., Lin, W., 2024e. Qqq: Quality quattuor-bit quantization for large language models. arXiv preprint arXiv:2406.09904 .

Zhang, Z., Jaiswal, A., Yin, L., Liu, S., Zhao, J., Tian, Y., Wang, Z., 2024f. Q-galore: Quantized galore with int4 projection and layer-adaptive low-rank gradients. arXiv preprint arXiv:2407.08296 .

Zhang, Z., Zhao, D., Miao, X., Oliaro, G., Li, Q., Jiang, Y., Jia, Z., 2024g. Quantized side tuning: Fast and memory-efficient tuning of quantized large language models. arXiv preprint arXiv:2401.07159 .

Zhao, Y., Lin, C.Y., Zhu, K., Ye, Z., Chen, L., Zheng, S., Ceze, L., Krishnamurthy, A., Chen, T., Kasikci, B., 2024. Atom: Low-bit quantization for efficient and accurate llm serving. Proceedings of Machine Learning and Systems 6, 196–209.

Zhu, F., Gong, R., Yu, F., Liu, X., Wang, Y., Li, Z., Yang, X., Yan, J., 2020. Towards unified int8 training for convolutional neural network, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 1969–1979.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Braverman, V., Beidi Chen, Hu, X., 2023. Kivi : Plug-and-play 2bit kv cache quantization with streaming asymmetric quantization URL: https://rgdoi.net/10.13140/RG.2.2.28167.37282, doi:10.13140/RG.2.2.28167.37282.