Small and Fast LLMs on Commodity Hardware: Post-Training Quantization in `llama.cpp`

Lorenz Sparrenberg*, Tobias Deußer*†,
Armin Berger*†, Rafet Sifa*†
* University of Bonn, Bonn, Germany
† Fraunhofer IAIS, Sankt Augustin, Germany
lsparren@uni-bonn.de
ORCID iD: 0000-0001-9450-7387

***Abstract*—Large Language Models (LLMs) have demonstrated remarkable capabilities but their significant computational and memory demands hinder widespread deployment, especially on resource-constrained devices. Quantization, the process of reducing the numerical precision of model parameters, has emerged as a critical technique for compressing LLMs and accelerating inference. This paper provides an overview of LLM quantization, with a particular focus on the Post-Training Quantization (PTQ) methods implemented within the popular `llama.cpp` framework and its GGUF file format. We begin by covering quantization fundamentals, including the distinction between PTQ and Quantization-Aware Training (QAT). We then describe the specific PTQ schemes employed by `llama.cpp`, including legacy methods, advanced K-quants, and recent IQ-quants, along with their underlying mathematical principles. The paper also discusses the impact of these techniques on model fidelity, hardware requirements, inference speed, and traces the adoption of GGUF as a *de facto* standard in the open-source community. This work serves as a practical guide and comprehensive reference for researchers aiming to deploy LLMs on resource-constrained hardware. By systematically documenting and comparing the PTQ methods within `llama.cpp`, we provide the necessary insights to navigate the trade-offs between model fidelity, inference speed, and memory footprint. This enables informed decision-making for real-world applications, from local CPU-based inference to efficient edge deployment.**

***Index Terms*—Large Language Models, LLM, Quantization, Model Compression, Post-Training Quantization, `llama.cpp`, GGUF, K-quants, Inference Efficiency.**

I. Introduction

Large Language Models (LLMs) such as LLaMA, GPT-3, and PaLM have revolutionized natural language processing, exhibiting impressive performance across a diverse range of tasks [1]–[3]. However, state-of-the-art LLMs often comprise billions of parameters, typically stored in high-precision formats like FP16 or FP32. This results in substantial memory footprints and high computational costs, posing significant challenges for deployment on consumer-grade hardware or edge devices [4], [5]. This limitation is particularly acute in domains handling sensitive information, where on-device processing is crucial for ensuring patient privacy in healthcare [6], maintaining confidentiality for financial data [7], [8], and protecting privileged legal documents [9].

Model quantization addresses these challenges by converting model weights to lower-precision numerical formats (e.g., INT8 or INT4), which reduces model size, memory bandwidth requirements, and can significantly accelerate computations [10]. The `llama.cpp` library [11] plays a pivotal role in this domain, providing highly optimized Post-Training Quantization (PTQ) implementations that have democratized access to powerful LLMs. Its associated GGUF file format has become a *de facto* standard for distributing these quantized models, as evidenced by its rapid and widespread adoption on platforms like Hugging Face (Figure 1).

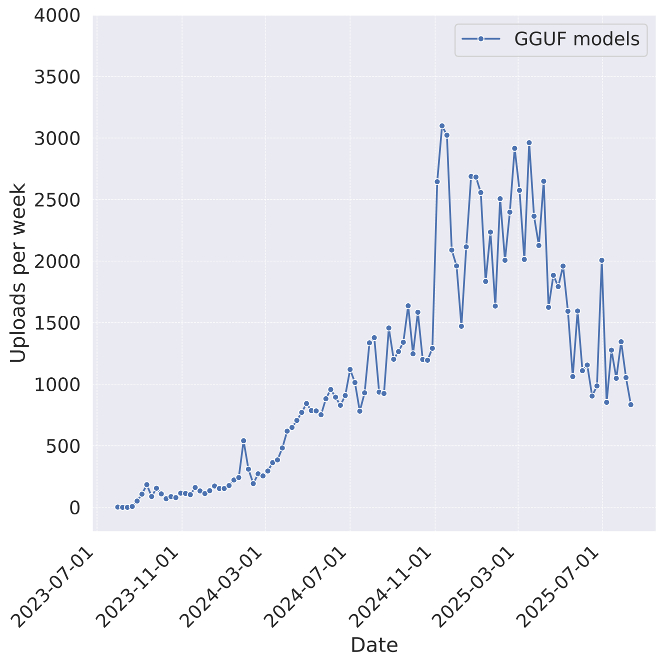

Fig. 1. Number of weekly uploaded GGUF models on Hugging Face. Since its introduction in August 2023, the GGUF format has been widely adopted and is now a *de facto* standard for running quantized large language models.

Despite their widespread adoption, the specific PTQ techniques within `llama.cpp` are primarily documented through source code, pull requests, and community discussions rather

This research has been funded by the Federal Ministry of Education and Research of Germany and the state of North-Rhine Westphalia as part of the Lamarr-Institute for Machine Learning and Artificial Intelligence.

than formal academic publications. This paper bridges that critical gap by providing the first consolidated, citable overview of these methods from legacy schemes to advanced K-quants and emerging IQ-quants. Our goal is to offer an application-focused perspective, equipping practitioners with the knowledge to navigate the trade-offs between model fidelity, hardware efficiency, and inference speed, thereby enabling informed decisions for real-world LLM deployment.
Scope and roadmap. We begin by recapping quantization fundamentals (II) and describing the pivotal role of the GGUF format (III). We then provide a detailed technical breakdown of the core PTQ methods implemented within `llama.cpp` (IV). Subsequently, we analyze the performance implications of these strategies, offering a practical guide to their trade-offs (V). We conclude with a discussion of open challenges and future research directions (VI, VII).

II. Quantization Basics

Quantization is the process of mapping continuous or high-precision discrete values to a smaller set of discrete values, typically represented with fewer bits. In the context of LLMs, this primarily involves reducing the precision of model weights and, occasionally, activations.

A. *Uniform Linear Quantization*

Uniform linear quantization is a common technique where the input range of values is divided into a fixed number of equally spaced segments. Each segment is then mapped to a unique integer value. Let b be the bit-width for quantization (e.g., 2, 4, 8 bits), and consider a block of N weights $\{w_1, \dots, w_N\}$. For each real-valued weight w_i, we seek an integer representation q_i. There are two main types:

a) Type-0 (Symmetric Quantization): This approach assumes that the weights are symmetrically distributed around zero.

1) Find the maximum absolute value in the block: $w_{\text{abs\_max}} = \max_i |w_i|$.
2) Calculate the scale factor (or step size) Δ:
$$\Delta = \frac{w_{\text{abs\_max}}}{2^{b-1}-1} \tag{1}$$
This choice approximates a mapping from $[-w_{\text{abs\_max}}, w_{\text{abs\_max}}]$ to the signed integer range $[-2^{b-1}, 2^{b-1}-1]$; note the negative bound is slightly extended when using -2^{b-1} with denominator $(2^{b-1}-1)$, a common implementation detail.
3) Quantize each weight w_i to a b-bit signed integer q_i:
$$q_i = \text{clamp}\left(\text{round}\left(\frac{w_i}{\Delta}\right), -2^{b-1}, 2^{b-1}-1\right) \tag{2}$$
4) Dequantize the integer q_i back to an approximate real value $\hat{w}_i$:
$$\hat{w}_i = q_i \times \Delta \tag{3}$$

In `llama.cpp`, methods like `Q4_0` and `Q8_0` are symmetric quantization schemes where only a scale factor d (equivalent to Δ) is stored per block.

b) Type-1 (Asymmetric Quantization): If the weight distribution is not centered around zero, asymmetric quantization can be more effective. It uses both a scale factor and a zero-point (or offset).

1) Determine the minimum $w_{\min} = \min_i w_i$ and maximum $w_{\max} = \max_i w_i$ values in the block.
2) Calculate the scale factor Δ:
$$\Delta = \frac{w_{\max} - w_{\min}}{2^b - 1} \tag{4}$$
This maps the range $[w_{\min}, w_{\max}]$ to the unsigned integer range $[0, 2^b - 1]$.
3) Calculate the zero-point Z, which is an integer representing the real value 0.0:
$$Z = \text{clamp}\left(\text{round}\left(-\frac{w_{\min}}{\Delta}\right), 0, 2^b - 1\right) \tag{5}$$
4) Quantize each weight w_i to a b-bit unsigned integer q_i:
$$q_i = \text{clamp}\left(\text{round}\left(\frac{w_i}{\Delta}\right) + Z, 0, 2^b - 1\right) \tag{6}$$
5) Dequantize the integer q_i back to $\hat{w}_i$:
$$\hat{w}_i = (q_i - Z) \times \Delta \tag{7}$$

Example: Asymmetric Quantization

Given parameters for a block of weights:
- Bit-width for quantization: $b = 4$ bits
- Minimum weight in block: $w_{\min} = -0.5$
- Maximum weight in block: $w_{\max} = 0.5$
- Exemplary weight to quantize: $w = -0.45$

1. Calculate Scale Factor Δ (using Eq. 4):
$$\Delta = \frac{w_{\max} - w_{\min}}{2^b - 1} = \frac{0.5 - (-0.5)}{2^4 - 1} = \approx 0.06667$$

2. Calculate Zero-Point Z (using Eq. 5):
$$Z = \text{round}\left(-\frac{w_{\min}}{\Delta}\right) = \text{round}\left(-\frac{-0.5}{0.06667}\right) = 8$$

3. Quantize the Weight w (using Eq. 6):
The quantization range for q_i is $[0, 2^b - 1] = [0, 15]$.
$$\begin{aligned} q &= \text{clamp}\left(\text{round}\left(\frac{w}{\Delta}\right) + Z,\ 0,\ 15\right) \\ &= \text{clamp}\left(\text{round}\left(\frac{-0.45}{0.06667}\right) + 8,\ 0,\ 15\right) \\ &= \text{clamp}(1,\ 0,\ 15) = 1 \end{aligned}$$

4. Reconstruct the Weight $\hat{w}$ (using Eq. 7):
$$\begin{aligned} \hat{w} &= (q - Z)\Delta \\ &= (1 - 8) \times 0.06667 \\ &= -7 \times 0.06667 \approx -0.46669 \end{aligned}$$

The original weight $w = -0.45$ is approximated as $\hat{w} \approx -0.46669$ after 4-bit asymmetric quantization.

TABLE I
PROMINENT ACADEMIC LLM QUANTIZATION METHODS

Method	Bits	Type	Key Idea	Ref.
LLM.int8()	8	PTQ	Mixed-precision decomposition; handles outliers in activations	[12]
SmoothQuant	8 (W8A8)	PTQ	Migrates quantization difficulty from activations to weights via channel re-scaling	[13]
GPTQ	3–4	PTQ	Layer-wise Hessian-aware optimization for weight quantization	[14]
AWQ	4	PTQ	Activation-aware weight scaling to protect salient weights	[15]
QLoRA	4	QAT (fine-tuning)	Quantizes a 4-bit base model (using ‘bitsandbytes‘) and fine-tunes Low-Rank Adapters	[16]
ZeroQuant	3-8	PTQ/QAT	Layer-wise knowledge distillation and group-wise quantization	[17]

In `llama.cpp`, methods like `Q4_1` and `Q5_1` are asymmetric, storing both a scale d and a minimum value m per block. The dequantization $\hat{w}_i = q_i \times d + m$ is a common variant, where m effectively incorporates $-Z \times \Delta$ and any initial minimum offset from the original weight range. The exact tie-handling for rounding depends on the function/backend used during conversion and kernels. Clamping ensures q_i stays within the representable b-bit range.

B. Post-Training vs. Quantization-Aware Training

Quantization techniques are broadly categorized based on when they are applied relative to the model training process:

- **Post-Training Quantization (PTQ):** PTQ methods quantize an already trained full-precision model. This is popular due to its simplicity as it avoids retraining and typically only requires a small calibration dataset to determine quantization parameters (like scales and zero-points) [10]. The `llama.cpp` framework, being primarily an inference engine, exclusively employs PTQ techniques. The tools provided with `llama.cpp` convert pre-trained models into quantized GGUF formats.
- **Quantization-Aware Training (QAT):** QAT simulates quantization effects (e.g., by inserting ”fake quantization” nodes that mimic the precision loss) during the training or fine-tuning process [18]. This allows the model to adapt its weights to the quantization noise, often yielding better accuracy, especially at very low bit-widths (e.g., 4-bit or less) [19]. However, QAT is more complex and computationally intensive than PTQ. Challenges include the increased training time, the need to handle non-differentiable quantization operations (often addressed using Straight-Through Estimators or STEs), and potential training instability [20]. Frameworks like PyTorch (with ‘torch.ao.quantization‘ [19]) and TensorFlow (via its Model Optimization Toolkit [21]) provide QAT support. Libraries like ‘bitsandbytes‘ [12] are crucial for QAT-like approaches such as QLoRA [16], where a pre-trained model is quantized (e.g., to 4-bit NF4) and then LoRA adapters are trained on top, making the adapter training ”aware” of the quantized base model. Active research continues to make QAT more efficient and effective for LLMs [20], [22].

It is noteworthy that some popular techniques like QLoRA [16] represent a hybrid approach. While the base model is quantized using a PTQ-like method, the subsequent fine-tuning of LoRA adapters is ’aware’ of this quantization, allowing the model to recover performance. This highlights how PTQ methods, such as those in llama.cpp, can serve as a foundation for more complex training schemes.
This survey focuses on the PTQ methods as implemented and utilized within `llama.cpp`.

C. Prominent Academic LLM Quantization Methods

While `llama.cpp` has its unique set of PTQ schemes, it’s useful to understand them in the context of broader academic research. Table I summarizes some influential methods from the literature, covering both PTQ and QAT.
These academic methods provide a rich backdrop against which the specific PTQ techniques developed and adopted by the `llama.cpp` community can be understood and appreciated. Many `llama.cpp` techniques, while practical and effective, aim for simplicity, broad CPU/GPU compatibility, and ease of use for inference.

III. `LLAMA.CPP` AND THE GGUF FILE FORMAT

`llama.cpp` is a C/C++ library for LLM inference, initially created by Georgi Gerganov in 2023 to run Facebook’s LLaMA model on commodity hardware [1], [11]. It has since significantly expanded its capabilities and now supports a wide array of model architectures beyond LLaMA, including popular models such as Mistral [23], Qwen [24], Gemma [25], Phi [26], and even non-Transformer architectures like Mamba [27]. A core strength of `llama.cpp` lies in its efficient implementation of various Post-Training Quantization (PTQ) schemes. These schemes enable diverse models to run on standard CPUs and various GPU backends (among others: CUDA, HIP/ROCm, Metal, Vulkan, SYCL, MUSA) by significantly reducing memory footprint and often accelerating computation through optimized kernels that perform on-the-fly dequantization of model weights during inference [11], [28].
The GGUF (GGML Universal File) was introduced in August 2023 by the `llama.cpp` developers as a successor to the earlier GGML format [29]. GGUF is a binary format meticulously designed to store LLMs, including their quantized weights and all necessary metadata, in a single, portable file. Figure 2 illustrates the general structure of a GGUF file.

Its key features, which directly support `llama.cpp`’s efficient inference capabilities, include [11]:

- **Unified Storage:** Contains model metadata (architecture, tokenizer information, special tokens, prompt templates),

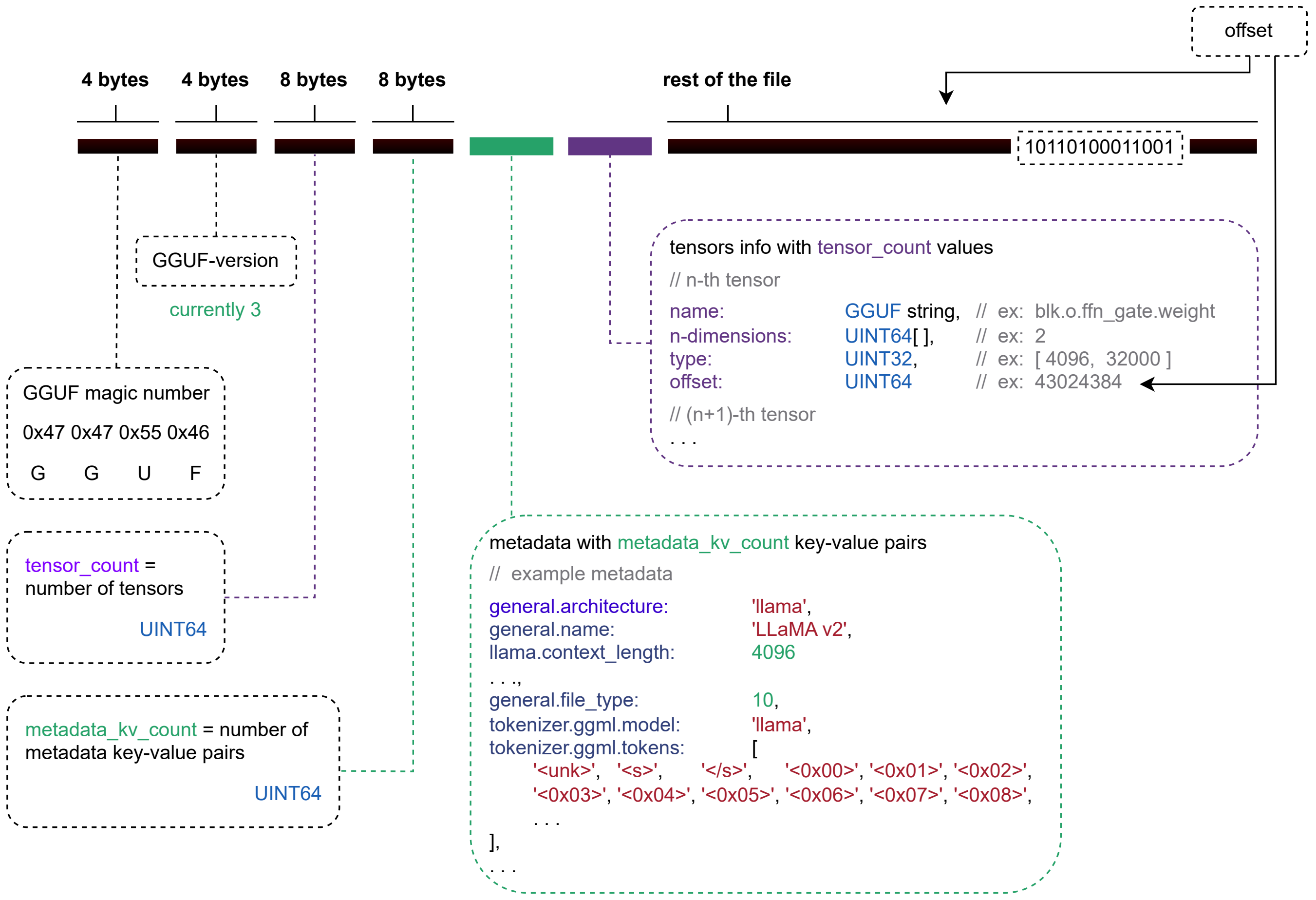
offset
4 bytes
4 bytes
8 bytes
8 bytes
rest of the file
10110100011001
GGUF-version
currently 3
GGUF magic number
0x47 0x47 0x55 0x46
G G U F
tensor_count = number of tensors
UINT64
metadata_kv_count = number of metadata key-value pairs
UINT64
tensors info with tensor_count values
// n-th tensor
name: GGUF string, // ex: blk.o.ffn_gate.weight
n-dimensions: UINT64[ ], // ex: 2
type: UINT32, // ex: [4096, 32000]
offset: UINT64 // ex: 43024384
// (n+1)-th tensor
. . .
metadata with metadata_kv_count key-value pairs
// example metadata
general.architecture: 'llama',
general.name: 'LLaMA v2',
llama.context_length: 4096
. . .,
general.file_type: 10,
tokenizer.ggml.model: 'llama',
tokenizer.ggml.tokens: [
'<unk>', '<s>', '</s>', '<0x00>', '<0x01>', '<0x02>',
'<0x03>', '<0x04>', '<0x05>', '<0x06>', '<0x07>', '<0x08>',
. . .
],
. . .

Fig. 2. Overall layout of a GGUF model file (version 3) adapted from Mishig Davaadorj [30]. From left to right the file contains: **(a)** a 24 byte fixed-size header that stores the ASCII magic word "GGUF", the format version, and the counts of tensors and metadata entries; **(b)** a variable-length block of hierarchical metadata key–value pairs, which also defines the global byte-alignment value `general.alignment`; **(c)** an array of tensor descriptors holding each tensor's name, rank, dimensions, data type, and the byte offset of its payload; and **(d)** the contiguous, alignment-padded tensor-data section containing the quantised or floating-point weight matrices. All multi-byte fields are little-endian by default. Every region begins at an offset that is a multiple of the global alignment, enabling zero-copy `mmap` loading.

tensor data (weights), and all quantization parameters (e.g., scales, offsets, lookup tables for K-quants) in one file. This self-contained nature simplifies distribution and loading.

- **Extensibility:** Allows for the addition of new metadata fields and tensor types without breaking backward compatibility. This is crucial for rapidly evolving model architectures and the continuous development of novel quantization methods within `llama.cpp`.
- **Quantization Support:** Natively designed to store the diverse range of PTQ methods developed within `llama.cpp`, from legacy block-wise schemes to advanced K-quants and IQ-quants. The format directly embeds the parameters required for on-the-fly dequantization.
- **Efficiency:** Optimized for fast loading through sequential data layout and memory mapping (often via 'mmap'). The defined alignment (see Figure 2) facilitates direct memory access to tensor data, allowing `llama.cpp` to efficiently stream and process quantized weights with minimal overhead, keeping the bulk of the model in its compressed state in memory.

Supported by its robust design and tight integration with `llama.cpp`'s optimized execution engine, GGUF has rapidly become the dominant format for sharing and running quantized LLMs, particularly on Hugging Face and other platforms. This widespread adoption facilitates community efforts in model distribution and enables local execution of increasingly powerful LLMs on a broad spectrum of hardware.

IV. Quantization Methods in `llama.cpp`

The quantization methods implemented in `llama.cpp` are exclusively Post-Training Quantization (PTQ) techniques primarily focused on weight-only quantization. These methods operate by dividing the model's weight tensors into smaller, contiguous blocks of weights. Each block is then quantized independently using its own set of quantization parameters, such as a scale factor and an offset (or minimum value), as detailed in Section II. This block-wise approach allows the quantization to adapt to the local distribution of weights, improving accuracy compared to per-tensor quantization.

A key aspect of `llama.cpp`'s efficiency with quantized models lies in its inference execution strategy. Instead of fully dequantizing entire weight tensors back to high precision (e.g., FP32) before computation, `llama.cpp` employs highly optimized compute kernels that perform dequantization on-the-fly [11]. During operations like matrix-vector multiplication, these kernels read the compact, quantized weight blocks (e.g., `Q4_K_M`, `Q6_K`) from memory. The necessary dequantization calculations—reconstructing floating-point values from the stored low-bit integers and their associated scales and metadata—are executed for small segments of data, often just in time for their use in the arithmetic units of the CPU or GPU [28], [31]. This approach significantly reduces memory bandwidth requirements and allows the bulk of the model weights to remain in their compressed form in memory, which is crucial for achieving fast inference speeds and accommodating large models on resource-constrained hardware. The specifics of these on-the-fly dequantization steps are embedded within the specialized routines for each quantization type within the 'ggml' library underpinning `llama.cpp`.

A. Legacy Quantization Schemes

These earlier PTQ methods in `llama.cpp` generally use a block size of $N = 32$ weights. The quantization parameters (scale factor Δ and optional minimum value m) are stored in FP16 format. They follow the principles of symmetric (Type-0) or asymmetric (Type-1) quantization outlined previously, with `llama.cpp` often using a specific variant for the asymmetric dequantization.

- **`Q4_0`:** This is a 4-bit symmetric PTQ method. For each block of 32 weights, a single FP16 scale factor Δ (as in Eq. 1) is stored. The weights w_i are quantized to $q_i \in [-8, 7]$. Dequantization is performed as $\hat{w}_i = q_i \times \Delta$. The effective bits per weight (bpw) is calculated as:
$$\frac{32 \text{ weights} \times 4 \text{ bits/weight} + 16 \text{ bits}}{32 \text{ weights}} = 4.5 \text{ bpw}.$$
- **`Q4_1`:** This 4-bit asymmetric PTQ method extends `Q4_0` by adding an FP16 minimum value m per block of 32 weights, along with the scale factor Δ. Quantized weights q_i are unsigned 4-bit integers $[0, 15]$. Dequantization follows $\hat{w}_i = q_i \times \Delta + m$, where m is the stored effective minimum of the dequantized range for the block. This scheme generally offers better precision for weight distributions not centered at zero. The effective bpw is
$$\frac{32 \text{ weights} \times 4 \text{ bits/weight} + 2 \times 16 \text{ bits}}{32 \text{ weights}} = 5.0 \text{ bpw}.$$
- **`Q8_0`:** An 8-bit symmetric PTQ method. Each block of 32 weights stores one FP16 scale factor Δ. Weights $q_i \in [-128, 127]$. Dequantization is $\hat{w}_i = q_i \times \Delta$. The effective bpw is
$$\frac{32 \text{ weights} \times 8 \text{ bits/weight} + 16 \text{ bits}}{32 \text{ weights}} = 8.5 \text{ bpw}.$$
- Other legacy types like `Q5_0` (5-bit, scale Δ only) and `Q5_1` (5-bit, scale Δ and minimum m) follow similar PTQ principles. For `Q5_1`, effective bpw is
$$\frac{32 \text{ weights} \times 5 \text{ bits/weight} + 2 \times 16 \text{ bits}}{32 \text{ weights}} = 6.0 \text{ bpw}.$$

B. K-Quants

Introduced in `llama.cpp` (see [32]), K-quants are advanced PTQ methods that significantly improved the trade-off between model compression and performance degradation. A key innovation of K-quants is the use of larger "super-blocks" (typically $N_K = 256$ weights, a constant defined in 'ggml-common.h') and the quantization of the parameters for smaller sub-blocks within them. This hierarchical structure reduces the overhead of storing quantization parameters compared to storing FP16 parameters for many small blocks, leading to lower effective bits per weight (bpw).

A K-quant super-block contains $N_K = 256$ weights, which are divided into N_{sub} smaller sub-blocks. For example, the `Q4_K_M` (a `Q4_K` variant, often "medium") and `Q5_K_M` types divide the 256 weights into $N_{sub} = 8$ sub-blocks of 32 weights each, while `Q6_K` uses $N_{sub} = 16$ sub-blocks of 16 weights. Each super-block stores the following:

- An FP16 super-block scale factor, Δ_{sb}.
- For asymmetric K-quant types like `Q4_K_M` and `Q5_K_M`, an FP16 super-block minimum/offset, m_{sb}. Symmetric types like `Q6_K` omit m_{sb} and only use Δ_{sb} for dequantizing sub-block parameters.
- For each sub-block $j \in [1, N_{sub}]$:
  - A k_s-bit quantized parameter, $q_{\Delta,j}$, used to derive the sub-block's own scale factor. For `Q4_K_M` and `Q5_K_M`, $k_s = 6$; for `Q6_K`, $k_s = 8$ (these are packed differently across the K-quant types).
  - A k_m-bit quantized parameter, $q_{m,j}$, used to derive the sub-block's own minimum/offset. For `Q4_K_M` and `Q5_K_M`, $k_m = 6$; for `Q6_K`, $k_m = 0$ (as it is symmetric at the sub-block level). These k_s and k_m bit values for the N_{sub} sub-blocks are packed together (e.g., into 12 bytes for `Q4_K_M` types).
- The N_K weights themselves, with each weight w_i quantized to b_w bits, denoted $q_{w,i}$ (e.g., $b_w = 4$ for `Q4_K` types, $b_w = 5$ for `Q5_K` types, $b_w = 6$ for `Q6_K`).

The dequantization of a b_w-bit weight $q_{w,i}$ (located in sub-block j) is a two-stage process [28]. First, the effective floating-point quantization parameters for sub-block j, i.e. its scale factor S_j and minimum/offset M_j, are reconstructed. These are derived from the stored quantized sub-block parameters ($q_{\Delta,j}$ and $q_{m,j}$) using the super-block's parameters (Δ_{sb} and, if present, m_{sb}). The specific functions, $S_j = f_S(q_{\Delta,j}, \Delta_{sb}, m_{sb})$ and $M_j = f_M(q_{m,j}, \Delta_{sb}, m_{sb})$, vary per K-quant type and involve bitwise operations and scaling, often utilizing lookup mechanisms. Second, once S_j (the effective scale for sub-block j) and M_j (the effective minimum/offset for sub-block j) are obtained, the weight $q_{w,i}$ is dequantized:

$$\hat{w}_i = q_{w,i} \times S_j + M'_j \tag{8}$$

Here, M'_j is the final effective offset applied to the weights in sub-block j. For asymmetric types like `Q4_K_M` and `Q5_K_M`, M'_j is typically the reconstructed sub-block minimum M_j (and $q_{w,i}$ is treated as unsigned). For symmetric K-quant types like `Q6_K`, M'_j is zero, and $q_{w,i}$ is treated as a signed integer offset from zero. This hierarchical approach yields a compact representation, as illustrated by the examples below.

Example: K-Quant Super-block Memory Footprint and BPW Calculations

For a K-quant super-block of $N_K = 256$ weights:

1. `Q4_K_M` (4-bit weights, asymmetric):

- $N_{sub} = 8$ sub-blocks. Weights: $b_w = 4$ bits.
- Sub-block metadata: $k_s = 6$ bits for $q_{\Delta,j}$, $k_m = 6$ bits for $q_{m,j}$ per sub-block.
- Super-block parameters: Δ_{sb} (FP16), m_{sb} (FP16).

$$\begin{aligned}\text{Bits} &= \underbrace{(256\times 4)}_{\text{weights}} + \underbrace{(8\times(6+6))}_{\text{sub-block meta}} + \underbrace{(2\times 16)}_{\text{super-block params}}\\ &= 1024 + 96 + 32\\ &= 1152 \text{ bits}\end{aligned}$$

$$\text{bpw} = \frac{1152 \text{ bits}}{256 \text{ weights}} = 4.50$$

2. `Q6_K` (6-bit weights, symmetric):

- $N_{sub} = 16$ sub-blocks. Weights: $b_w = 6$ bits.
- Sub-block metadata: $k_s = 8$ bits for $q_{\Delta,j}$ per sub-block (total 128 bits for all sub-block scales, packed efficiently). $k_m = 0$.
- Super-block parameters: Δ_{sb} (FP16 only). m_{sb} is omitted.

$$\begin{aligned}\text{Bits} &= \underbrace{(256\times 6)}_{\text{weights}} + \underbrace{(16\times 8)}_{\text{sub-block scales}} + \underbrace{(1\times 16)}_{\text{super-block scale}}\\ &= 1536 + 128 + 16\\ &= 1680 \text{ bits}\end{aligned}$$

$$\text{bpw} = \frac{1680 \text{ bits}}{256 \text{ weights}} = 6.5625$$

This demonstrates how the overhead of quantization parameters is amortized over the 256 weights in the super-block. The specific packing of sub-block metadata varies between K-quant types.

Table II lists some common K-quant configurations. "Sub-Blk" indicates the number of sub-blocks and weights per sub-block. b_w is bits for weights $q_{w,i}$ and k_s, k_m are bits for parameters $q_{\Delta,j}, q_{m,j}$.

a) Rationale for K-Quants.: Usually, high-magnitude weights are often sparse. Thus, giving each smaller region (sub-block) its own refined quantization parameter helps retain local shape information effectively. Quantizing the metadata itself keeps the overall bit-per-weight overhead low, especially compared to storing FP16 scales/mins for many small blocks.

TABLE II
REPRESENTATIVE K-QUANT CONFIGURATIONS (SUPER-BLOCK OF 256 WEIGHTS)

Format	b_w	k_s/k_m	SubBlk	bpw
`Q2_K`	2	4/4	16×16	2.5625
`Q3_K`	3	6/–	16×16	3.4375
`Q4_K_M`	4	6/6	8×32	4.50
`Q5_K_M`	5	6/6	8×32	5.50
`Q6_K`	6	8/–	16×16	6.5625

For the full list of tensor encoding schemes and bpw, see the `llama.cpp` wiki [33]

b) Limitations.: The super-block size (e.g., 256) is a trade-off: too small and metadata overhead increases; too large and the quantization parameters may not adapt well to local weight distributions. The integer metadata for sub-blocks must be unpacked to derive actual scales/mins during inference, which can introduce a slight runtime overhead compared with legacy formats.

C. IQ-Quants (Importance Quantization)

Introduced in early 2024, IQ-Quants are a family of post-training quantization (PTQ) methods in `llama.cpp` designed for very low bit-rates (roughly 1.5–3.5 bpw). The initial releases included 2-bit variants (`IQ2_XXS` at 2.0625 bpw and `IQ2_XS` at 2.31 bpw), followed by `IQ1_S` (1.5 bpw) and `IQ3_S` (3.4375 bpw). Compared to K-Quants, IQ-Quants use a data-driven *importance* signal during conversion to allocate precision where it matters most, together with compact codebooks for magnitude/sign encoding [34]–[37].

a) The Importance Matrix: IQ uses an **importance matrix** ("iMatrix") computed once on a small calibration set. For each layer, `imatrix` stores per-input-channel second-moment statistics (e.g., $\langle a_i^2\rangle$) used to weight quantization error over tokens, yielding a *vector* of length equal to the input/hidden dimension. This diagonal approximates second-order curvature of a per-row quantization objective and serves as weights during quantization. [38] This is conceptually similar to GPTQ, which uses $\langle aa^\top\rangle$ (including off-diagonals). [14]

b) Quantization and Dequantization Mechanism: During quantization, the importance vector weights the error metric and informs sign-parity choices and E8-lattice codebook selection within each block. The resulting GGUF stores the chosen codebook indices and per-block scales; the iMatrix itself is not embedded in the model. At inference, dequantization performs codebook lookups plus per-block scaling. The performance is backend-dependent (e.g., IQ3_S matches Q3_K on CUDA/AVX2 but is slower on ARM NEON due to lookups). [34], [37], [38]

c) IQ-Quant Types and Characteristics: Representative IQ types and their bits per weights (bpw):

IQ1_S:	Effective bpw = 1.5. The most aggressive widely used IQ type.
IQ2_XXS:	Effective bpw = 2.0625. A true 2-bit variant with strong compression.

IQ2_XS: Effective bpw = 2.31. Higher quality than `IQ2_XXS` with modest size increase.

IQ3_S: Effective bpw = 3.4375. Significantly outperforms `Q3_K`; approaches `Q4_K` on some models.

Conversion is slower and needs a small calibration dataset to compute the iMatrix, but quality-per-bit is excellent, especially below 4 bpw. [34], [37], [38]

V. Performance Implications of `llama.cpp` Quantization

The PTQ methods in `llama.cpp` primarily aim to improve model efficiency, which involves trade-offs with model fidelity. Table III shows exemplary data for LLaMA-2 models.

A. Model Fidelity

Quantization inevitably introduces information loss, which can manifest as degraded performance on evaluation benchmarks. A common intrinsic metric for this is perplexity, which measures a model's uncertainty in predicting the next token. However, while perplexity is a useful indicator of raw language modeling capability, its correlation with performance on downstream tasks is often weak, particularly for complex reasoning or instruction-following scenarios [41], [42]. The perplexity values in Table III, therefore, should be interpreted as a measure of linguistic information loss rather than a definitive predictor of final application performance.

A more robust evaluation involves measuring performance on a suite of downstream benchmarks. The community around `llama.cpp` heavily relies on standardized academic benchmarks such as MMLU for general knowledge [43], HellaSwag for commonsense reasoning [44], and TruthfulQA for factuality [45], often run using standardized frameworks like the EleutherAI Language Model Evaluation Harness [42]. The consensus from extensive community testing, as well as academic studies on 4-bit quantization like QLoRA [16], confirms several key trends:

TABLE III
Exemplary LLaMA-2 quantizations in `llama.cpp`

Model	Quant	BPW	Size (GB)	Δ PPL	Ref.
LLaMA-2 7B	FP16	16.0	13.0	0.000	[39]
	`Q8_0`	8.50	6.70	+0.0004	[40]
	`Q6_K`	6.56	5.15	+0.0044	[31]
	`Q5_K_M`	5.50	4.45	+0.0142	[31]
	`Q4_K_M`	4.50	3.80	+0.0535	[31]
LLaMA-2 13B	FP16	16.0	25.0	0.000	[39]
	`Q8_0`	8.50	13.0	+0.0005	[40]
	`Q5_1`	6.00	9.1	+0.0163	[40]
	`Q5_0`	5.50	8.3	+0.0313	[40]
	`Q4_1`	5.00	7.6	+0.1065	[40]
	`Q4_0`	4.50	6.8	+0.1317	[40]

Overview of PTQ methods based on LLaMA-2 models. The deltas are computed against the full-precision FP16 perplexity on WikiText-2 with a 512-token context window. **BPW**: effective bits per weight, including scale/min overhead. **Size**: checkpoint size on disk after quantization (GGUF). Δ **PPL**: increase over FP16 perplexity (lower is better).

- **8-bit PTQ** (e.g., `Q8_0`) results in minimal degradation and is often considered near-lossless, retaining over 99.5% of the FP16 model's score on benchmarks like MMLU.
- **High-quality 4-bit and 5-bit K-quants** (e.g., `Q4_K_M`, `Q5_K_M`) represent a "sweet spot," frequently retaining over 98% of the FP16 model's capability on these benchmarks while offering substantial resource savings [16]. This has made them a *de facto* standard for practical, high-performance deployment.
- **Extreme low-bit quantizations** (2-bit and 3-bit K-quants and IQ-quants) show more pronounced degradation but have become viable for scenarios where memory is the primary constraint. Newer methods like IQ-quants are specifically designed to outperform older methods at the same low bit rate, as discussed in their initial implementation proposals [34].
- **Stronger degradation in small models**, whereas larger models (e.g., 13B parameters and above) have proven to be more robust to aggressive quantization, a phenomenon also observed in academic works like ZeroQuant [17].

B. Hardware Efficiency and Inference Speed

The primary motivations for PTQ in `llama.cpp` are reduced resource usage and faster inference.

- **Model Size and Memory:** This is the most direct benefit. A 16 bpw FP16 model is roughly halved in size by 8-bit quantization (e.g., `Q8_0` at 8.5 bpw) and quartered by 4-bit quantization (e.g., `Q4_K_M` at 4.5 bpw). This reduction is crucial for fitting large models into consumer-grade VRAM (e.g., 8GB to 24GB) or running them on CPUs with sufficient system RAM.
- **Inference Speed:** Speedups arise from several interrelated factors:
  - **Alleviating the Memory Bandwidth Bottleneck:** LLM inference, particularly for large batch sizes or long sequences, is often bound by memory bandwidth—the speed at which model weights can be moved from VRAM or system RAM to the processor's on-chip compute units [41], [46]. By significantly reducing the size of the weights, quantization allows for more data to be transferred in the same amount of time, directly increasing the rate of token generation.
  - **Optimized Compute Kernels:** The speed improvements are not merely from data transfer. `llama.cpp` employs highly-optimized compute kernels where the dequantization from a low-bit integer to a floating-point value occurs on-the-fly, just-in-time for the matrix multiplication operation. This "dequantize-and-compute fusion" is implemented in the core 'ggml' library and ensures that only a small block of weights needs to be in a high-precision format in the processor's registers at any given moment [11]. The specific routines for each quantization type, which fuse these steps, can be found in the project's source code, for example in the implementation of the K-quants [31].

TABLE IV
PRACTICAL GUIDE TO `llama.cpp` QUANTIZATION METHODS

Quant Type	Typical BPW	Relative Quality	Inference Speed	Primary Use Case
FP16 / BF16	16.0	Baseline	1.0x (Baseline)	Research, fine-tuning, or when VRAM is not a constraint. Performance is often limited by memory bandwidth on consumer hardware.
`Q8_0`	8.50	Close to FP16	1.0x – 1.5x	Best for high-quality results where VRAM allows. Often indistinguishable from FP16 in practice and serves as a robust, high-fidelity baseline.
`Q5_K_M`/`Q6_K`	5.50 – 6.56	Very High	1.4x – 2.4x	The "go-to" choice for balancing high quality and excellent performance. `Q6_K` offers top-tier fidelity, while `Q5_K_M` is a good all-rounder.
`Q4_K_M`	4.50	High	2.0x – 3.5x	**The most popular choice for general use on consumer GPUs.** Offers an optimal blend of speed, reduced memory usage, and high quality, making large models accessible.
`Q3_K`/`Q2_K`	2.56 – 3.44	Moderate	2.5x – 4.0x	Use when memory is extremely constrained (e.g., running larger models on CPUs or low-VRAM GPUs). Expect noticeable quality degradation.
IQ-Quants (`IQ3_S`/`IQ2_XS`)	2.06 – 3.63	Moderate	2.5x – 4.0x	Pushing the limits of compression for the most resource-scarce environments. Best for scenarios where model size is the absolute priority; can outperform K-quants at the same low bit rate.

  – **KV Cache Quantization:** Beyond weights, `llama.cpp` supports quantizing the KV cache, which stores the key/value attention states for previously generated tokens. As context length grows, this cache can consume gigabytes of memory, becoming a secondary memory bandwidth bottleneck. Quantizing the KV cache to formats like INT8 not only reduces its memory footprint, allowing for much longer contexts on the same hardware, but can also improve generation speed by reducing the I/O pressure during auto-regressive decoding, supported in mainline `llama.cpp` (since PR #7412), with maintainer tests showing negligible perplexity change at 8-bit cache [47].

Overall, the combination of smaller weight files, reduced memory bandwidth pressure, and optimized on-the-fly dequantization kernels allows quantized models in `llama.cpp` to achieve significant inference speedups over their FP16 counterparts.

C. *Practical implications*

To translate these performance characteristics into actionable guidance, Table IV provides a practical decision-making framework. This table offers a synthesized view of the trade-offs between model size, fidelity, and inference speed for the primary quantization families in `llama.cpp`.

The Relative Speed column reports the estimated speedup vs. an FP16 baseline on the same hardware. Unless noted otherwise, these figures primarily reflect the decode phase (token-by-token generation) in memory-bandwidth-bound scenarios common on consumer GPUs and large-model CPU runs. During the prefill phase (prompt ingestion with large matrix multiplies), workloads are more compute-bound and quantized backends incur dequantization overhead; as a result, speedups are typically smaller and can even reverse (FP16/BF16 matching or outperforming some quantizations at medium–large batch sizes). Generally, real-world performance varies with hardware (CPU vs. GPU, VRAM/RAM bandwidth), backend, model architecture, batch size, and context length.

VI. DISCUSSION AND FUTURE DIRECTIONS

The Post-Training Quantization (PTQ) methods in `llama.cpp` have made significant strides in making LLMs accessible. However, several challenges and promising research avenues remain to push the boundaries of efficient LLM deployment:

- **Pushing Compression Limits in PTQ:** While 2-bit and 3-bit methods show promise, maintaining fidelity at these extreme levels remains a key challenge for pure PTQ. Future work could involve quantizing activations in addition to weights, a difficult but potentially rewarding task. Integrating academic approaches like SmoothQuant [13] or developing practical 1-bit PTQ schemes [48] are important research directions.
- **Rigor in Evaluation and Method Design:** Progress in quantization requires moving beyond perplexity, which often fails to capture degradation on downstream tasks [49]. The development of comprehensive, standardized benchmarks is crucial. Similarly, a deeper theoretical understanding of why empirically-driven methods like K-quants or IQ-quants succeed would enable the design of more robust and predictable schemes.
- **Hardware Co-Design and Kernel Optimization:** Further performance gains can be unlocked by minimizing the runtime overhead of complex PTQ schemes through better kernel fusion. Tighter co-design with hardware, such as fully exploiting specialized instructions for low-bit arithmetic, remains a key avenue for improving inference speed.
- **Hybrid PTQ-QAT Approaches:** Combining the simplicity of PTQ with the power of Quantization-Aware Training (QAT) is a highly promising direction. For instance, using an aggressively quantized GGUF model as a base for a brief period of lightweight fine-tuning (e.g., with LoRA) could recover significant performance with minimal training cost, representing a "best-of-both-worlds" approach for practitioners.

Addressing these challenges will be key to enabling even more powerful models to run on increasingly resource-constrained devices, whether through advanced PTQ or more accessible hybrid techniques.

VII. Conclusion

Quantization is an indispensable technology for making Large Language Models practical and accessible for a wide range of data science and AI applications. The `llama.cpp` framework, along with its GGUF file format, has been instrumental in this endeavor by providing a suite of effective Post-Training Quantization (PTQ) methods for a growing number of model architectures. This paper has offered a structured overview of these techniques, starting from fundamental quantization principles (differentiating PTQ and QAT) and prominent academic approaches, then delving into the specifics of `llama.cpp`'s legacy PTQ schemes, the advanced K-quants, and the emerging IQ-quants. We have detailed their underlying concepts, mathematical basis where available, how `llama.cpp` executes computations with them, and their impact on model size, inference speed, and fidelity.

The rapid evolution and widespread adoption of GGUF and its associated PTQ types underscore the open-source community's drive for efficient LLM deployment. By significantly reducing model size and accelerating inference primarily through post-training approaches, these methods empower users to run powerful LLMs on consumer-grade hardware, fostering innovation and broader access. While, as discussed, challenges remain in areas like extreme low-bit PTQ, activation quantization within PTQ frameworks, and standardized evaluation, the ongoing work within the `llama.cpp` project and the wider research community continues to push the boundaries of what is achievable. This survey aims to serve as a citable academic reference to acknowledge these impactful, community-driven advancements in LLM quantization.

Acknowledgment

The authors acknowledge the pioneering work of Georgi Gerganov and the entire `llama.cpp` community in developing, refining, and popularizing the methods and tools discussed in this paper. Their dedication has been crucial in democratizing access to large language models.

Parts of the text in all sections have been refined with support from OpenAI GPT (o3-2025-04-16) [50] and Google Gemini (gemini-2.5-pro-preview-05-06) [51].

References

[1] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *ArXiv*, vol. abs/2302.13971, 2023. [Online]. Available: https://arxiv.org/abs/2302.13971

[2] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[3] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "Palm: scaling language modeling with pathways," *Journal of Machine Learning Research*, vol. 24, no. 1, Jan. 2023. [Online]. Available: https://www.jmlr.org/papers/volume24/22-1144/22-1144.pdf

[4] Z. Wan, X. Wang, C. Liu, S. Alam, Y. Zheng, J. Liu, Z. Qu, S. Yan, Y. Zhu, Q. Zhang *et al.*, "Efficient large language models: A survey," *Transactions on Machine Learning Research*, 2024. [Online]. Available: https://openreview.net/pdf?id=bsCCJHbO8A

[5] R. Gong, Y. Ding, Z. Wang, C. Lv, X. Zheng, J. Du, H. Qin, J. Guo, M. Magno, and X. Liu, "A survey of low-bit large language models: Basics, systems, and algorithms," *ArXiv*, vol. abs/2409.16694, 2024. [Online]. Available: https://arxiv.org/abs/2409.16694

[6] T. Deußer, A. M. Siddiqi, L. Sparrenberg, T. Adams, C. Bauckhage, and R. Sifa, "Fusing speech and language models for dementia detection," in *2024 IEEE International Conference on Big Data (BigData)*. IEEE, 2024, pp. 3908–3914. [Online]. Available: https://doi.org/10.1109/BigData62323.2024.10825055

[7] A. Berger, L. Hillebrand, D. Leonhard, T. Deußer, T. B. F. De Oliveira, T. Dilmaghani, M. Khaled, B. Kliem, R. Loitz, C. Bauckhage, and R. Sifa, "Towards automated regulatory compliance verification in financial auditing with large language models," in *2023 IEEE International Conference on Big Data (BigData)*, 2023, pp. 4626–4635. [Online]. Available: https://doi.org/110.1109/BigData59044.2023.10386518

[8] T. Deußer, C. Zhao, D. Uedelhoven, L. Sparrenberg, L. Hillebrand, C. Bauckhage, and R. Sifa, "Leveraging large language models for few-shot kpi extraction from financial reports," in *2024 IEEE International Conference on Big Data (BigData)*. IEEE, 2024, pp. 4864–4868. [Online]. Available: https://doi.org/10.1109/BigData62323.2024.10825458

[9] T. Deußer, C. Zhao, L. Sparrenberg, D. Uedelhoven, A. Berger, M. Pielka, L. Hillebrand, C. Bauckhage, and R. Sifa, "A comparative study of large language models for named entity recognition in the legal domain," in *2024 IEEE International Conference on Big Data (BigData)*. IEEE, 2024, pp. 4737–4742. [Online]. Available: https://doi.org/10.1109/BigData62323.2024.10825695

[10] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. Van Baalen, and T. Blankevoort, "A white paper on neural network quantization," *ArXiv*, vol. abs/2106.08295, 2021. [Online]. Available: https://arxiv.org/abs/2106.08295

[11] G. Gerganov and contributors. (2023) llama.cpp: Inference of llama model in pure c/c++. GitHub. Accessed: May 7, 2025. [Online]. Available: https://github.com/ggml-org/llama.cpp

[12] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, "Llm.int8(): 8-bit matrix multiplication for transformers at scale," in *Proceedings of the 36th International Conference on Neural Information Processing Systems*, ser. NIPS '22. Red Hook, NY, USA: Curran Associates Inc., 2022. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/c3ba4962c05c49636d4c6206a97e9c8a-Paper-Conference.pdf

[13] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han, "Smoothquant: Accurate and efficient post-training quantization for large language models," in *International Conference on Machine Learning*. PMLR, 2023, pp. 38 087–38 099. [Online]. Available: https://proceedings.mlr.press/v202/xiao23c/xiao23c.pdf

[14] E. Frantar, S. Ashkboos, T. Hoefler, and D.-A. Alistarh, "Gptq: Accurate post-training quantization for generative pre-trained transformers," in *11th International Conference on Learning Representations*, 2023. [Online]. Available: https://research-explorer.ista.ac.at/download/17378/17385/2023_ICLR_Frantar.pdf

[15] J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, and S. Han, "Awq: Activation-aware weight quantization for on-device llm compression and acceleration," *Proceedings of Machine Learning and Systems*, vol. 6, pp. 87–100, 2024. [Online]. Available: https://proceedings.mlsys.org/paper_files/paper/2024/file/42a452cbafa9dd64e9ba4aa95cc1ef21-Paper-Conference.pdf

[16] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," in *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., vol. 36. Curran Associates, Inc., 2023, pp. 10 088–10 115. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/file/1feb87871436031bdc0f2beaa62a049b-Paper-Conference.pdf

[17] Z. Yao, R. Yazdani Aminabadi, M. Zhang, X. Wu, C. Li, and Y. He, "Zeroquant: Efficient and affordable post-training quantization for large-scale transformers," *Advances in Neural Information Processing Systems*, vol. 35, pp. 27 168–27 183, 2022. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/adf7fa39d65e2983d724ff7da57f00ac-Paper-Conference.pdf

[18] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. G. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2704–2713, 2018. [Online]. Available: https://doi.org/10.1109/CVPR.2018.00286

[19] PyTorch Team, *Quantization - PyTorch 2.7 documentation*, The PyTorch Foundation, 2024, accessed: May 15, 2025. [Online]. Available: https://pytorch.org/docs/stable/quantization.html

[20] Y. Bondarenko, R. D. Chiaro, and M. Nagel, "Low rank quantization-aware training for LLMs," in *Workshop on Efficient Systems for Foundation Models II @ ICML2024*, 2024. [Online]. Available: https://openreview.net/forum?id=RJCBalFiCg

[21] TensorFlow Team, *Quantization aware training - TensorFlow 2.19 documentation*, Google, 2024, accessed: May 15, 2025. [Online]. Available: https://www.tensorflow.org/model_optimization/guide/quantization/training

[22] M. Chen, W. Shao, P. Xu, J. Wang, P. Gao, K. Zhang, Y. Qiao, and P. Luo, "Efficientqat: Efficient quantization-aware training for large language models," *CoRR*, vol. abs/2407.11062, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2407.11062

[23] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, "Mistral 7b," Mistral, Tech. Rep., 2023. [Online]. Available: https://arxiv.org/abs/2310.06825

[24] J. Bai, S. Bai, Y. Chu, Z. Cui, K. Dang, X. Deng, Y. Fan, W. Ge, Y. Han, F. Huang *et al.*, "Qwen technical report," Alibaba Cloud, Tech. Rep., 2023. [Online]. Available: https://arxiv.org/abs/2309.16609

[25] T. Mesnard, C. Hardin, R. Dadashi, S. Bhupatiraju, S. Pathak, L. Sifre, M. Rivière, M. S. Kale, J. Love, P. Tafti *et al.*, "Gemma: Open models based on gemini research and technology," *CoRR*, vol. abs/2403.08295, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2403.08295

[26] M. Abdin, J. Aneja, S. Bubeck, C. C. T. Mendes, W. Chen, A. D. Giorno, S. G. Ronen Eldan, S. Gunasekar, M. Javaheripi, P. Kauffmann *et al.*, "Phi-2: The surprising power of small language models," https://www.microsoft.com/en-us/research/blog/phi-2-the-surprising-power-of-small-language-models/, 2023, accessed: May 15, 2025.

[27] A. Gu and T. Dao, "Mamba: Linear-time sequence modeling with selective state spaces," *arXiv preprint arXiv:2312.00752*, 2023. [Online]. Available: https://arxiv.org/abs/2312.00752

[28] G. Gerganov and contributors, "ggml-quants.c and ggml-quants.h," GitHub, 2023-2024, source code for quantization routines. Accessed: May 7, 2025. [Online]. Available: https://github.com/ggml-org/ggml/tree/master/src

[29] ——. (2023) GGUF (pull request #2398, ggml-org/llama.cpp). GitHub. Accessed: May 16, 2025. [Online]. Available: https://github.com/ggml-org/llama.cpp/pull/2398

[30] M. Davaadorj, "GGUF file-structure diagram (v3)," GitHub, 2024, diagram embedded in project documentation; author credited as @mishig25. Accessed: 8 May, 2025. [Online]. Available: https://github.com/ggml-org/ggml/blob/master/docs/gguf.md

[31] I. Kawrakow and contributors. (2023) K-quants (pull request: #1684, ggml-org/llama.cpp). GitHub. Accessed: May 7, 2025. [Online]. Available: https://github.com/ggml-org/llama.cpp/pull/1684

[32] G. Gerganov and contributors. (2024) Introduce *k*-bit group quantization formats (pull request #3360, ggml-org/llama.cpp). GitHub. Accessed: May 8, 2025. [Online]. Available: https://github.com/ggml-org/llama.cpp/pull/3360

[33] ggml org/llama.cpp. (2024) Tensor encoding schemes (iq/k quant bpw table). GitHub. Accessed: August 14, 2025. [Online]. Available: https://github.com/ggml-org/llama.cpp/wiki/Tensor-Encoding-Schemes

[34] I. Kawrakow and contributors. (2024) Sota 2-bit quants (pull request: #4773, ggml-org/llama.cpp). GitHub. Pull request introducing state-of-the-art 2-bit quantization (IQ2_XXS) to llama.cpp. Accessed: May 8, 2025. [Online]. Available: https://github.com/ggml-org/llama.cpp/pull/4773

[35] ——. (2024) Sota 2-bit quants - part 2 (adds iq2_xs 2.31 bpw), (pull request #4856). GitHub. Accessed: August 14, 2025. [Online]. Available: https://github.com/ggml-org/llama.cpp/pull/4856

[36] ——. (2024) 1.5 bit quantization (iq1_s) (pull request #5453). GitHub. Accessed: August 14, 2025. [Online]. Available: https://github.com/ggml-org/llama.cpp/pull/5453

[37] ——. (2024) Iq3_s: a much better alternative to q3_k (pull request #5676). GitHub. Accessed: August 14, 2025. [Online]. Available: https://github.com/ggml-org/llama.cpp/pull/5676

[38] ——. (2024) Importance matrix calculation (pull request #4861). GitHub. Accessed: August 14, 2025. [Online]. Available: https://github.com/ggml-org/llama.cpp/pull/4861

[39] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023. [Online]. Available: https://arxiv.org/pdf/2307.09288

[40] G. Gerganov and contributors, "Quantization results table and formats (Q4_K_M, Q5_K_M, Q6_K, etc.)," GitHub, 2025, accessed: May 8, 2025. [Online]. Available: https://github.com/ggml-org/llama.cpp/blob/master/tools/quantize/quantize.cpp

[41] Y. Tay, M. Dehghani, S. Abnar, Y. Shen, D. Bahri, P. Pham, J. Rao, L. Yang, S. Ruder, and D. Metzler, "Efficient transformers: A survey," *ACM Computing Surveys*, vol. 55, no. 6, pp. 1–28, 2023. [Online]. Available: https://doi.org/10.1145/3530811

[42] L. Gao, J. Tow, S. Biderman, S. Black, A. DiPofi, C. Foster, L. Gold, J. Hsu, K. Le Noan, L. Li *et al.*, "A framework for few-shot language model evaluation," in *ACL 2023 Workshop on Benchmarking: Past, Present and Future*, 2023. [Online]. Available: https://doi.org/10.5281/zenodo.5371629

[43] D. Hendrycks, C. Burns, S. Basart, A. Critch, J. Li, D. Ippolito, D. Israel, J. Kaplan, A. Askell, A. Glad *et al.*, "Measuring massive multitask language understanding," *arXiv preprint arXiv:2009.03300*, 2020. [Online]. Available: https://arxiv.org/abs/2009.03300

[44] R. Zellers, A. Holtzman, Y. Bisk, A. Farhadi, and Y. Choi, "HellaSwag: Can a machine really finish your sentence?" in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 4791–4800. [Online]. Available: https://www.aclweb.org/anthology/P19-1472

[45] S. Lin, J. Hilton, and O. Evans, "TruthfulQA: Measuring how models mimic human falsehoods," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 3214–3252. [Online]. Available: https://aclanthology.org/2022.acl-long.229

[46] N. Shazeer, "Fast transformer decoding: One write-head is all you need," *arXiv preprint arXiv:1911.02150*, 2019. [Online]. Available: https://arxiv.org/pdf/1911.02150

[47] G. Gerganov *et al.* (2024) feat: add support for quantizing the k-cache (pull request #7412). GitHub. Accessed: August 27, 2025. [Online]. Available: https://github.com/ggml-org/llama.cpp/pull/7412

[48] J. Chee, Y. Cai, V. Kuleshov, and C. M. De Sa, "Quip: 2-bit quantization of large language models with guarantees," *Advances in Neural Information Processing Systems*, vol. 36, pp. 4396–4429, 2023. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/file/0df38cd13520747e1e64e5b123a78ef8-Paper-Conference.pdf

[49] R. Jin, J. Du, W. Huang, W. Liu, J. Luan, B. Wang, and D. Xiong, "A comprehensive evaluation of quantization strategies for large language models," in *Findings of the Association for Computational Linguistics ACL 2024*, 2024, pp. 12 186–12 215. [Online]. Available: https://aclanthology.org/2024.findings-acl.726.pdf

[50] OpenAI, "Introducing openai o3 and o4-mini," https://openai.com/index/introducing-o3-and-o4-mini/, 2025, official OpenAI announcement and overview of the o3 and o4-mini reasoning models.

[51] Gemini Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, K. Millican *et al.*, "Gemini: A family of highly capable multimodal models," 2025. [Online]. Available: https://arxiv.org/abs/2312.11805