Comprehensive Evaluation of Quantization Methods for Edge LLM Deployment with RAG Focus

Zahraa Selim, Menna Hamed, Wesam Ahmed,
Sohyla Said, Sara Basheer, Rami Zewail

Computer Science and Engineering Department
Egypt-Japan University of Science and Technology (E-JUST)
Alexandria, Egypt
`{zahraa.selim, menna.hamed, rami.zewail}@ejust.edu.eg`

December 10, 2025

Abstract

The deployment of Large Language Models (LLMs) on resource-constrained edge hardware is significantly impeded by memory bandwidth bottlenecks. Post-Training Quantization (PTQ) has emerged as a standard compression paradigm, yet comprehensive benchmarking of modern quantization methods on consumer-grade hardware remains limited. This study conducts a rigorous empirical evaluation of five advanced quantization techniques (NF4, GPTQ, AWQ, HQQ, and AutoRound) applied to Mistral-7B on NVIDIA Tesla T4 hardware. Our findings indicate that NormalFloat 4-bit (NF4) quantization establishes an optimal Pareto frontier, achieving 3.6× memory reduction while maintaining superior task performance and hardware compatibility. Critically, we identify that hardware-algorithm compatibility significantly influences performance, with distribution-based methods (NF4) providing superior stability on Turing architectures compared to kernel-dependent methods (GPTQ, HQQ) that may experience severe fallback overhead. Our analysis reveals task-dependent sensitivity to quantization, where mathematical reasoning degrades by 25% while knowledge retrieval remains robust. We provide prescriptive deployment guidelines identifying NF4 as the superior strategy for balancing throughput, energy efficiency, and task performance on edge hardware.

1 Introduction

The democratization of Large Language Models (LLMs) represents a transformative shift in artificial intelligence accessibility. Models such as Mistral-7B [?], with their balance of capability and efficiency, exemplify the potential for deploying sophisticated reasoning systems beyond cloud infrastructure. However, the transition from datacenter-grade hardware to consumer and edge devices—such as the NVIDIA T4 and RTX series GPUs—confronts a fundamental bottleneck: memory bandwidth and capacity constraints.

A typical 7-billion parameter model in half-precision (FP16) format requires approximately 14GB of Video Random Access Memory (VRAM) for weights alone, consuming nearly the entire capacity of a 16GB GPU and leaving minimal headroom for the dynamic memory allocations essential during inference. The Key-Value (KV) cache, which grows linearly with sequence length, further exacerbates this constraint. For a context window of 4096 tokens, the KV cache alone demands an additional 1GB of memory, making longer-context applications infeasible without compression.

Post-Training Quantization (PTQ) techniques offer a compelling solution by reducing numerical precision without the computational overhead of retraining. Methods such as GPTQ [?], AWQ [?], and NormalFloat 4-bit (NF4) [?] theoretically achieve 3.5-4× compression ratios, reducing model footprints to approximately 3.5-4GB. However, quantization introduces non-uniform degradation patterns across different capabilities—a phenomenon inadequately captured by aggregate metrics like perplexity or average benchmark scores.

1.1 Research Gaps and Motivation

Current quantization literature exhibits three critical limitations that motivate this investigation:

Incomplete Understanding of Task-Specific Sensitivity Existing evaluations predominantly rely on perplexity measurements and broad benchmark suites (e.g., MMLU, HellaSwag), which provide aggregate performance indicators but obscure differential sensitivities across task categories. Mathematical reasoning, for instance, may degrade more severely than factual recall due to quantization-induced precision loss in arithmetic operations. Code generation may suffer disproportionately from syntactic errors that small quantization perturbations introduce. Without granular, capability-specific evaluation, practitioners lack guidance on which compression methods best preserve the capabilities most relevant to their deployment scenarios.

Hardware-Software Co-optimization Neglect Quantization methods are often evaluated in hardware-agnostic settings, assuming universal kernel support and optimal implementations. This assumption breaks down in practice: quantization schemes optimized for Ampere architecture (e.g., INT4 tensor cores) may experience severe performance degradation on older Turing GPUs (e.g., Tesla T4), which lack native low-precision arithmetic support and rely on software emulation. The interaction between quantization algorithm design and actual hardware capabilities remains underexplored, yet critically determines real-world deployment viability.

Limited Retrieval-Augmented Generation (RAG) Analysis RAG systems, which augment language models with external knowledge retrieval, represent a dominant deployment paradigm for domain-specific applications. However, quantization's impact on RAG workflows—specifically on context adherence, multi-hop reasoning across retrieved documents, and faithfulness to source material—remains largely uncharacterized. The compound effects of compressing both the retrieval mechanism (through embedding quantization) and the generation model (through weight quantization) require systematic investigation, particularly through novel metrics like attention preservation, context degradation, and attention drift.

1.2 Research Contributions

This work addresses these gaps through a comprehensive evaluation framework that examines six state-of-the-art quantization methods across three analytical dimensions:

Diverse Quantization Paradigms We evaluate methods spanning fundamentally different compression principles:

- **Data-type innovations (NF4)**: Information-theoretically optimal representations for normal distributions

- **Hessian-based optimization (GPTQ)**: Second-order curvature-aware quantization minimizing reconstruction error
- **Activation-aware weighting (AWQ)**: Salience-based protection of critical weights identified through activation magnitudes
- **Rotation-based outlier redistribution (QuaRot)**: Hadamard transformations to smooth activation distributions
- **Statistical adaptivity (HQQ)**: Zero-shot quantization without calibration data
- **Memory hierarchy optimization (KVQuant)**: Specialized compression for attention KV caches

This breadth enables systematic comparison of orthogonal design philosophies rather than incremental algorithmic variations.

Multi-Dimensional Evaluation Protocol We assess compressed models across:

- **Computational efficiency**: Latency (TTFT, decode time), throughput, memory footprint, energy consumption, and Model FLOPs Utilization (MFU)
- **Task-specific performance**: Granular evaluation across mathematical reasoning (GSM8K), commonsense reasoning (HellaSwag, ARC), world knowledge (MMLU), and code generation (HumanEval)
- **RAG capabilities**: Novel metrics including attention preservation (precision@1 and rank of relevant documents), context degradation (accuracy slope over increasing lengths), and attention drift (stability during generation)

Hardware-Aware Deployment Analysis All experiments are conducted on a single NVIDIA Tesla T4 GPU (16GB VRAM), representative of edge deployment scenarios. This constraint reveals critical insights about algorithm-hardware compatibility: methods requiring specialized kernel support (e.g., structured sparsity, INT4 tensor cores) may underperform distribution-based approaches (e.g., NF4) that gracefully degrade to software implementations on older architectures.

Our findings demonstrate that quantization method selection must account for deployment context. For mathematical reasoning preservation, Hessian-based methods (GPTQ) minimize precision-sensitive degradation. For memory-constrained RAG applications, KV cache quantization (KVQuant) provides orthogonal compression benefits beyond weight quantization alone. For hardware compatibility on Turing-generation GPUs, distribution-based methods (NF4, HQQ) avoid kernel fallback penalties that plague methods assuming Ampere-specific optimizations.

1.3 Paper Organization

The remainder of this paper proceeds as follows: Section II surveys related work in quantization methods, organizing techniques by underlying compression principles rather than chronological development. Section III details our experimental methodology, including quantization configurations, evaluation protocols, and RAG pipeline design with novel attention-based metrics. Section IV presents comprehensive results across efficiency, quality, and RAG dimensions. Section V discusses implications for deployment strategy and identifies optimal method selection criteria. Section VI concludes with future research directions.

2 Related Work

Quantization for large language models has emerged as a critical research area, with techniques spanning diverse compression principles. We organize this survey by underlying methodology rather than chronological development, highlighting how different approaches address the fundamental trade-off between compression ratio and capability preservation.

2.1 Curvature-Based Optimization Methods

These methods leverage second-order information about the loss surface to identify optimal quantization parameters that minimize reconstruction error.

Hessian-Guided Quantization **GPTQ** [?] pioneered practical Hessian-based quantization for billion-parameter models through layer-wise optimization with inverse Hessian approximations. The method formulates quantization as minimizing $\|WX - \hat{W}X\|_2^2$ where W is the original weight matrix and $\hat{W}$ is quantized, using a greedy coordinate descent approach that propagates quantization errors to subsequent weights via $w_{q'} \leftarrow w_{q'} - \delta_q \cdot \frac{H_{qq'}}{H_{qq}}$. This Optimal Brain Quantization (OBQ) strategy enables 3-4 bit quantization with minimal perplexity degradation.

QuIP [?] extends this framework to 2-bit precision through adaptive rounding and incoherence processing, applying randomized Hadamard transforms before quantization to reduce weight correlations. **QuIP#** further improves this approach with fine-tuned incoherence preprocessing, achieving state-of-the-art 2-bit performance.

OPTQ variants including **GPTAQ** and **MR-GPTQ** introduce mixed-precision strategies, allocating higher precision to layers identified as sensitive through Hessian eigenvalue analysis. **OPQ** (Outlier-Preserved Quantization) combines Hessian guidance with explicit outlier handling, storing high-magnitude weights in elevated precision.

AutoRound employs iterative reconstruction with adaptive rounding, optimizing quantization parameters per-layer through gradient descent on reconstruction error. Unlike GPTQ's one-shot approach, AutoRound refines parameters across multiple iterations, often improving upon GPTQ with minimal computational overhead.

Curvature Approximations Several methods approximate Hessian computation to reduce overhead. **QUAD** uses diagonal Hessian approximations, trading accuracy for speed. **FPTQ** (Fast Post-Training Quantization) employs block-wise Hessian estimation, processing weight submatrices independently. **ResQ** introduces residual quantization, iteratively quantizing reconstruction errors to achieve finer effective precision.

2.2 Activation-Aware and Salience-Based Methods

These approaches identify and protect critical weights based on activation statistics or gradient information, recognizing that uniform quantization degrades performance by treating all parameters equally.

Salience-Based Weight Protection **AWQ** (Activation-aware Weight Quantization) [?] observes that 1% of weights disproportionately impact model outputs when weighted by activation magnitudes. By applying per-channel scaling factors $s = (\text{mean}(|X|_\alpha))^\alpha$ where X represents activations, AWQ protects salient weights while aggressively quantizing less-critical parameters. This salience-based approach demonstrates superior preservation of reasoning capabilities compared to reconstruction-based methods.

OWQ (Outlier-Aware Weight Quantization) extends this principle by explicitly identifying weights sensitive to activation outliers, applying mixed-precision storage where sensitivity exceeds thresholds. **SpQR** (Sparse-Quantized Representation) uses L2 error as a sensitivity metric, storing the top-k sensitive weights in higher precision while quantizing the remainder.

SqueezeLLM [?] introduces sensitivity-based weight clustering via k-means, grouping weights by importance scores and allocating precision budgets per cluster. This enables non-uniform quantization that concentrates bits where they most impact model quality.

WUSH (Weight-Update Salience Heuristic) tracks weight update magnitudes during fine-tuning as a proxy for importance, protecting high-update weights during subsequent quantization. **AdpQ** (Adaptive Precision Quantization) dynamically adjusts per-layer precision based on activation variance, allocating more bits to layers with high variance.

Gradient-Based Importance **XQuant** leverages gradient information to identify critical weight columns, applying column-wise mixed precision. **GQSA** (Gradient-Quantization Sensitivity Analysis) computes sensitivity scores as $\nabla_W \mathcal{L}$, protecting high-gradient regions. **EWQ** (Error-Weighted Quantization) uses backpropagated errors to guide precision allocation, iteratively refining quantization to minimize task-specific loss.

2.3 Outlier Redistribution and Smoothing

Activation outliers—extreme values that occur in specific channels—pose significant challenges for quantization, as they force large quantization ranges that waste representation capacity. These methods address outliers through transformation or redistribution rather than preservation.

Rotation-Based Approaches **QuaRot** applies fixed Hadamard rotation matrices H to weight matrices before quantization, transforming $W \to HWH^T$, which redistributes outliers across channels while preserving the mathematical equivalence through inverse rotations during inference. This approach achieves 4-bit quantization competitive with 16-bit baselines by homogenizing activation distributions.

SpinQuant extends rotation methods with learnable orthogonal transformations optimized to maximize quantization friendliness, formulated as $\min_R \|Q(RW) - RW\|$ subject to $RR^T = I$. By learning rotation matrices rather than using fixed transforms, SpinQuant adapts to model-specific outlier patterns.

ButterflyQuant employs structured butterfly matrices for efficient rotation, reducing the $O(n^2)$ rotation cost to $O(n \log n)$ while maintaining outlier smoothing effectiveness.

Channel-Wise Smoothing **SmoothQuant** [?] migrates quantization difficulty from activations to weights through per-channel scaling: $Y = (X\text{diag}(s)^{-1}) \cdot (\text{diag}(s)W)$, where scaling factors s balance difficulty. This reduces outlier impact without architectural changes.

Outlier Suppression+ [?] suppresses outliers through activation clipping and shifting. **CliP** combines clipping with progressive quantization, gradually reducing precision while monitoring error.

RepQ (Representation Quantization) normalizes activations to unit variance before quantization, redistributing dynamic range. **Outlier Token Suppression** identifies and suppresses outlier-causing tokens during inference.

2.4 Mixed-Precision and Heterogeneous Quantization

These methods allocate variable precision across model components, recognizing that uniform bit-width sacrifices efficiency.

Layer-Wise Precision Allocation **MixPrecision** dynamically assigns precision based on layer sensitivity, measured by signal-to-quantization-noise ratio (SQNR). Layers with high SQNR tolerate lower precision.

LLM.int8() [?] uses INT8 for regular weights and FP16 for outlier features identified by vector norms exceeding thresholds.

FlexiQuant allows arbitrary bit-width combinations per layer, optimized via search algorithms. **MPQ** (Multi-Precision Quantization) groups layers by type (attention vs. feed-forward) for precision assignment.

Component-Specific Strategies **Token-wise Quantization** applies per-token scaling, adapting to varying activation distributions across sequences. **Group-wise Quantization** partitions weights into groups with shared scaling factors, balancing granularity and overhead.

CQ (Channel Quantization) applies different precision to different attention head groups, allocating higher bits to heads responsible for long-range dependencies.

2.5 KV Cache Compression

The Key-Value cache in Transformer attention grows linearly with sequence length, dominating memory consumption for long-context applications. KV cache quantization provides orthogonal compression beyond weight quantization.

Per-Channel and Per-Token Strategies **KVQuant** introduces separate quantization strategies for keys and values: per-channel quantization for keys (which remain relatively stable across tokens) and per-token quantization for values (which vary significantly). Additionally, KVQuant preserves *attention sinks*—initial tokens that accumulate disproportionate attention—in full precision to maintain attention distribution fidelity.

KIVI (KV Cache In-place Inference) implements 2-bit KV cache quantization with asymmetric quantization schemes, achieving 4× KV cache compression. By quantizing in-place during generation, KIVI minimizes memory overhead and enables million-token contexts on consumer hardware.

AsymKV applies asymmetric quantization to keys and values separately, recognizing their different statistical properties. **AQUA-KV** (Adaptive Quantization for KV Cache) adjusts quantization granularity based on sequence length, using coarse quantization for long contexts and fine quantization for short contexts.

Attention-Aware KV Compression **IntactKV** preserves high-attention keys and values in elevated precision, dynamically identifying important cache entries based on cumulative attention scores. **WKVQuant** applies cross-block reconstruction regularization, ensuring that quantization errors in one layer's KV cache don't compound in subsequent layers.

TEQ (Token-Efficient Quantization) combines KV cache quantization with token pruning, removing low-attention tokens from the cache entirely.

2.6 Quantization-Aware Training and Fine-Tuning

While post-training quantization requires no retraining, quantization-aware training (QAT) can further reduce degradation by adapting model parameters to low-precision representations.

Full QAT Methods **LLM-QAT** implements standard QAT through knowledge distillation from full-precision teachers, training quantized models to mimic teacher outputs. **Efficien-**

tQAT reduces QAT cost through progressive quantization, starting with high precision and gradually reducing bit-widths during training.

BitNet and **BiLLM** explore 1-bit weight quantization with QAT, representing weights as $\{-1, +1\}$. **PB-LLM** (Partially Binarized LLM) applies 1-bit quantization selectively, binarizing only low-variance weight blocks.

Parameter-Efficient QAT **QLoRA** [?] combines NF4 quantization with Low-Rank Adaptation (LoRA), training low-rank adapters on top of frozen quantized weights. This enables fine-tuning 65B models on single GPUs by limiting trainable parameters to <1% of total model size.

QA-LoRA extends this with quantization-aware LoRA initialization, optimizing adapter ranks and placement based on quantization sensitivity. **LoftQ** iteratively refines LoRA decomposition and quantization parameters jointly. **PRILoRA** introduces precision-incremental LoRA, gradually reducing precision during adaptation. **IR-QLoRA** combines importance reweighting with QLoRA for improved preservation of critical capabilities.

PEQA (Parameter-Efficient Quantization Adaptation) applies adapter modules specifically at quantization-sensitive layers, concentrating trainable parameters where they most impact quality recovery.

2.7 Specialized Domains and Formats

Code-Specific Quantization **EETQ** (Efficient Embedding and Tokenizer Quantization) addresses quantization for code models, where exact token matching is critical. **MoFQ** (Mode-Focused Quantization) recognizes that code token distributions are multimodal and designs quantization schemes respecting mode boundaries.

Tensor Format Innovations **GGUF** formats (including **GGUF-Q** and **GGUF-IQ**) provide portable quantized model representations supporting mixed precision and efficient memory-mapped inference. These formats enable deployment across diverse hardware without recompilation.

torchao and **Quanto** provide PyTorch-native quantization APIs, integrating seamlessly with existing training pipelines. **QServe** offers optimized serving infrastructure for quantized models with dynamic batching and request scheduling.

2.8 Emerging Directions

Recent work explores frontiers including:

Ultra-Low Precision: **INT2.1**, **ZeroQuant-4+2**, and **VPTQ** (Variable Precision Tensor Quantization) push toward sub-2-bit quantization through aggressive outlier handling and specialized data types.

Learnable Quantization Functions: **SGD Weight Rounding**, **MagR** (Magnitude-Aware Rounding), and **FlexRound** optimize rounding functions rather than using fixed round-to-nearest.

Norm-Based Methods: **Norm Tweaking**, **FrameQuant**, and **AffineQuant** manipulate layer normalization statistics to create quantization-friendly activation distributions.

Distribution Matching: **KurTail** matches quantized weight distributions to target kurtosis values, **DecoupleQ** separates magnitude and sign quantization, **DuQuant** applies dual-domain quantization in both spatial and frequency domains.

2.9 Summary and Positioning

This survey reveals that quantization research has progressed from uniform precision reduction to sophisticated, heterogeneous compression strategies that account for weight salience, activation statistics, hardware constraints, and task-specific requirements. However, systematic comparison across diverse quantization paradigms under controlled conditions remains limited, particularly for RAG applications where context-dependence introduces additional complexity.

Our work fills this gap by evaluating six representative methods spanning orthogonal design principles (NF4, GPTQ, AWQ, QuaRot, HQQ, KVQuant) under unified experimental conditions on standardized hardware (Tesla T4), with particular attention to RAG scenarios underexplored in prior literature. We use standard calibration datasets for all methods requiring calibration (e.g., GPTQ, AWQ), without task-specific or RAG-optimized calibration.

3 Methodology

3.1 Base Model and Hardware

This study utilizes **Mistral-7B-v0.1** [?] as the baseline model, selected for its parameter efficiency and strong performance profile. All experiments are conducted on a single **NVIDIA Tesla T4 GPU (16GB VRAM)** to simulate realistic edge deployment scenarios. The software environment includes PyTorch 2.x, Transformers 4.x, and quantization-specific libraries (bitsandbytes, AutoGPTQ, AutoAWQ, HQQ).

3.2 Compression Techniques

3.2.1 NF4 (NormalFloat 4-bit)

NF4 utilizes the quantile-based NormalFloat data type, which is information-theoretically optimal for zero-centered normal distributions. Double quantization is employed to further minimize memory footprint by quantizing the quantization constants themselves [?].

Configuration:

- Quantization type: NF4 (NormalFloat4)
- Double quantization: Enabled
- Compute dtype: float16
- Group size: Per-tensor

Implementation:

```
from transformers import BitsAndBytesConfig
import torch

nf4_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_use_double_quant=True,
    bnb_4bit_compute_dtype=torch.float16
)

model = AutoModelForCausalLM.from_pretrained(
    "mistralai/Mistral-7B-Instruct-v0.1",
    quantization_config=nf4_config,
    device_map="auto"
)
```

3.2.2 GPTQ

GPTQ (Generative Pre-trained Transformer Quantization) is a post-training quantization method that quantizes weights to 2-8 bits while maintaining model quality through layer-wise optimization [?].

Method Overview:

GPTQ addresses the computational challenges of optimal quantization by reformulating the problem as a series of layer-wise optimizations. For each layer's weight matrix $W \in \mathbb{R}^{d_{out} \times d_{in}}$, GPTQ solves:

$$\min_{\hat{W}} ||WX - \hat{W}X||_2^2 \tag{1}$$

where W is the original weight matrix, $\hat{W}$ is the quantized weight matrix, and $X \in \mathbb{R}^{d_{in} \times n}$ represents calibration inputs with n samples.

The key innovation is the use of approximate second-order information through the Hessian matrix $H = \frac{2}{n}XX^T$. GPTQ employs a greedy coordinate descent approach, quantizing weights column-by-column while adjusting subsequent weights to compensate for quantization error. For each weight w_q being quantized:

$$\hat{w}_q = \text{quant}(w_q), \quad \delta_q = w_q - \hat{w}_q \tag{2}$$

The quantization error δ_q is then propagated to remaining weights:

$$w_{q'} \leftarrow w_{q'} - \delta_q \cdot \frac{H_{qq'}}{H_{qq}} \quad \forall q' > q \tag{3}$$

This Optimal Brain Quantization (OBQ)-inspired approach ensures that quantization errors are minimized across the entire weight matrix [?].

Group-wise Quantization:

To balance precision and compression, GPTQ applies quantization in groups. Each group of g weights (typically $g = 128$) shares quantization parameters (scale s and zero-point z):

$$\hat{w}_i = \text{round}\left(\frac{w_i}{s}\right) - z, \quad s = \frac{\max(w_g) - \min(w_g)}{2^b - 1} \tag{4}$$

where b is the number of bits (2, 3, or 4). Smaller group sizes increase accuracy at the cost of additional metadata storage.

Quantization Process:

We quantized Mistral-7B-v0.1 at three precision levels (4-bit, 3-bit, and 2-bit) using auto-gptq on a P100 GPU:

- Base model: mistralai/Mistral-7B-v0.1 (7.24B parameters)
- Bits: 2, 3, 4
- Group size: 128
- Calibration dataset: WikiText-2 (128 samples, max length 512 tokens)
- Damping factor: 0.01 (Hessian regularization)
- Memory optimization: cache_examples_on_gpu=False

Implementation:

```python
from auto_gptq import AutoGPTQForCausalLM, BaseQuantizeConfig
from datasets import load_dataset

# Prepare calibration data
def prepare_calibration_data(tokenizer, n_samples=128,
                              max_length=512):
    dataset = load_dataset("wikitext", "wikitext-2-raw-v1",
                            split="train")
    dataset = dataset.filter(lambda x: len(x["text"]) > 200)

    examples = []
    for i in range(min(n_samples, len(dataset))):
        tokenized = tokenizer(dataset[i]["text"],
            return_tensors="pt", max_length=max_length,
            truncation=True, padding=False)
        examples.append({
            "input_ids": tokenized["input_ids"],
            "attention_mask": tokenized["attention_mask"]
        })
    return examples

calibration_data = prepare_calibration_data(tokenizer)

# Quantize model
model = AutoGPTQForCausalLM.from_pretrained(
    "mistralai/Mistral-7B-v0.1",
    quantize_config=BaseQuantizeConfig(
        bits=4, group_size=128, desc_act=False,
        damp_percent=0.01),
    max_memory={0: "10GiB", "cpu": "50GiB"})

model.quantize(calibration_data, use_triton=False,
    batch_size=1, cache_examples_on_gpu=False)

model.save_quantized("./mistral-7b-gptq-4bit",
                     use_safetensors=True)
```

The quantization process takes approximately 22 minutes per model on a P100 GPU, with memory usage optimized through CPU offloading.

Optimized Inference with ExLlamaV2:

For evaluation, we employed ExLlamaV2, an optimized inference engine for GPTQ models that provides 2-3x faster inference through specialized CUDA kernels:

```python
from exllamav2 import (ExLlamaV2, ExLlamaV2Config,
                       ExLlamaV2Cache, ExLlamaV2Tokenizer)
from exllamav2.generator import (ExLlamaV2StreamingGenerator,
                                 ExLlamaV2Sampler)

# Load with ExLlamaV2
config = ExLlamaV2Config()
config.model_dir = "./mistral-7b-gptq-4bit"
config.prepare()
config.max_seq_len = 4096

model = ExLlamaV2(config)
cache = ExLlamaV2Cache(model, lazy=True)
model.load_autosplit(cache)
tokenizer = ExLlamaV2Tokenizer(config)

# Generate with optimized kernels
generator = ExLlamaV2StreamingGenerator(model, cache,
                                        tokenizer)
settings = ExLlamaV2Sampler.Settings()
settings.temperature = 0.7
settings.top_p = 0.9

input_ids = tokenizer.encode(prompt)
generator.begin_stream(input_ids, settings)
```

ExLlamaV2 optimizations include custom CUDA kernels for 2/3/4-bit matrix multiplication, fused attention and MLP operations, and lazy KV cache loading to reduce memory overhead.

Theoretical Compression:

For Mistral-7B with 7.24B parameters, theoretical model sizes are:

- FP16 baseline: $7.24 \times 2 = 14.48$ GB
- 4-bit GPTQ: $7.24 \times 0.5 + \text{overhead} \approx 3.62$ GB (4.0x compression)
- 3-bit GPTQ: $7.24 \times 0.375 + \text{overhead} \approx 2.72$ GB (5.3x compression)
- 2-bit GPTQ: $7.24 \times 0.25 + \text{overhead} \approx 1.81$ GB (8.0x compression)

where overhead includes quantization metadata (scales, zero-points) and unquantized components (embeddings, layer norms).

3.2.3 AWQ (Activation-aware Weight Quantization)

AWQ protects salient weights based on activation magnitudes, preserving the top 1% most impactful weights with per-channel scaling [?].

Configuration:

- Bits: 4
- Group size: 128
- Zero point: True
- Version: GEMM

Implementation:

```python
from awq import AutoAWQForCausalLM

model = AutoAWQForCausalLM.from_quantized(
    "TheBloke/Mistral-7B-Instruct-v0.1-AWQ",
    fuse_layers=True,
    safetensors=True
)
```

3.2.4 HQQ (Half-Quadratic Quantization)

HQQ performs fast, calibration-free quantization optimizing a custom loss function with on-the-fly parameter optimization [?].

Configuration:

- Bits: 4
- Group size: 64
- Axis: 1 (row-wise)
- Compute dtype: float16

Implementation:

```python
from hqq.models.hf.base import AutoHQQHFModel
from hqq.core.quantize import BaseQuantizeConfig

quant_config = BaseQuantizeConfig(
    nbits=4, group_size=64, axis=1
)

model = AutoHQQHFModel.from_pretrained(
    "mistralai/Mistral-7B-Instruct-v0.1",
    torch_dtype=torch.float16
)
model.quantize_model(
    quant_config=quant_config, device='cuda'
)
```

3.2.5 AutoRound

AutoRound employs adaptive quantization with iterative optimization for per-layer parameter tuning.

Configuration:

- Bits: 4
- Group size: 128
- Samples: 64
- Iterations: 50
- Batch size: 4
- Learning rate: 5e-3

Implementation:

```
from auto_round import AutoRound

ar = AutoRound(
    model=MODEL_PATH,
    scheme="W4A16",
    bits=4,
    group_size=128,
    nsamples=64,
    iters=50,
    lr=5e-3,
    seqlen=1024,
    batch_size=4,
    amp_dtype=torch.float16
)

ar.quantize_and_save(
    output_dir=OUTPUT_DIR,
    format="auto_round"
)
```

3.3 Evaluation Metrics

We assess compressed models across two primary dimensions: computational efficiency and task performance.

3.3.1 Efficiency Metrics

Latency Measurements

- **Average Latency:** Mean time per generated token (ms)
- **Time-to-First-Token (TTFT):** Initial response latency
- **Prefill vs. Decode:** Separate measurement of prompt processing vs. autoregressive generation

Throughput and Memory

- **Throughput:** Tokens per second (tok/s)
- **Peak Memory:** Maximum GPU memory (MB)

- **Model Size:** Disk storage requirements (GB)

Computational Efficiency

- **Model FLOPs Utilization (MFU):** Percentage of theoretical peak hardware FLOPs achieved:
$$\text{MFU} = \frac{\text{Achieved FLOPs/s}}{\text{Peak Hardware FLOPs/s}} \times 100\% \tag{5}$$
- **Energy Consumption:** Estimated energy per token (mJ):
$$E = (P_{\text{TDP}} - P_{\text{idle}}) \times t_{\text{inference}} \tag{6}$$
where $P_{\text{idle}} \approx 0.3 \times P_{\text{TDP}}$

3.3.2 Performance Benchmarks

We evaluate models using established language modeling benchmarks:

Language Modeling Perplexity: Measured on WikiText-2 test set using sliding window evaluation (stride 512):

$$\text{PPL}(W) = \exp\left(-\frac{1}{N}\sum_{i=1}^{N} \ln P(w_i|w_{<i})\right) \tag{7}$$

Core Task Benchmarks

- **HellaSwag** (0-shot): Commonsense reasoning via sentence completion
- **ARC-Easy** (0-shot): Grade-school science questions
- **ARC-Challenge** (0-shot): Challenge-level scientific reasoning
- **GSM8K** (8-shot): Grade-school math word problems
- **MMLU** (5-shot): Multi-domain knowledge across 57 subjects
- **HumanEval** (0-shot): Python code generation (pass@1)

All evaluations use the Language Model Evaluation Harness [**?**] with consistent hyperparameters for reproducibility.

4 Results

This section presents empirical results across three evaluation dimensions: computational efficiency (Section 4.1), task performance quality (Section 4.2), and RAG-specific capabilities (Section 4.3). All measurements are conducted on NVIDIA Tesla T4 (16GB VRAM) using Mistral-7B-v0.1 as the base model.

4.1 Computational Efficiency Analysis

4.1.1 Latency Metrics

Table 1: Latency measurements across quantization methods

Method	Latency (ms/tok)	TTFT (ms)	Prefill (ms)	Decode (ms/tok)	Speedup vs FP16
FP16					1.00×
NF4					
GPTQ					
AWQ					
HQQ					

Table 2: Latency measurements: GPTQ calibration dataset comparison

Calibration Dataset	Latency (ms/tok)	TTFT (ms)	Prefill (ms)	Decode (ms/tok)	Speedup vs FP16
GPTQ-WikiText					
GPTQ-RAG					

4.1.2 Memory and Compression

Table 3: Memory utilization and compression metrics

Method	Model Size (GB)	Peak Memory (MB)	Bits/ Param	Compression Ratio	Memory Efficiency
FP16	13.49		16.0	1.00×	
NF4			4.0		
GPTQ			4.0		
AWQ			4.0		
HQQ			4.0		

Table 4: Memory utilization: GPTQ calibration dataset comparison

Calibration Dataset	Model Size (GB)	Peak Memory (MB)	Bits/ Param	Compression Ratio	Memory Efficiency
GPTQ-WikiText			4.0		
GPTQ-RAG			4.0		

4.1.3 Throughput

Table 5: Generation throughput across quantization methods

Method	Throughput (tok/s)	Total Tokens Generated	Total Time (s)	Throughput vs FP16
FP16				1.00×
NF4				
GPTQ				
AWQ				
HQQ				

Table 6: Generation throughput: GPTQ calibration dataset comparison

Calibration Dataset	Throughput (tok/s)	Total Tokens Generated	Total Time (s)	Throughput vs FP16
GPTQ-WikiText				
GPTQ-RAG				

4.1.4 Batch Inference Scaling

Table 7: Batch throughput scaling (samples/second)

Method	Batch=1	Batch=2	Batch=4	Batch=8	Optimal Batch	Optimal Throughput	Scaling Efficiency
FP16							
NF4							
GPTQ							
AWQ							
HQQ							

Table 8: Batch throughput scaling: GPTQ calibration dataset comparison

Calibration Dataset	Batch=1	Batch=2	Batch=4	Batch=8	Optimal Batch	Optimal Throughput	Scaling Efficiency
GPTQ-WikiText							
GPTQ-RAG							

4.1.5 Energy Consumption

Table 9: Energy consumption estimates

Method	Energy/Token (mJ)	TDP (W)	Energy Efficiency vs FP16	Est. Cost ($/1M tokens)
FP16		70	1.00×	
NF4		70		
GPTQ		70		
AWQ		70		
HQQ		70		

Table 10: Energy consumption: GPTQ calibration dataset comparison

Calibration Dataset	Energy/Token (mJ)	TDP (W)	Energy Efficiency vs FP16	Est. Cost ($/1M tokens)
GPTQ-WikiText		70		
GPTQ-RAG		70		

4.2 Task Performance Analysis

4.2.1 Language Modeling: Perplexity

Table 11: Perplexity on WikiText-2 test set

Method	Perplexity	Loss	Delta PPL vs FP16	Degradation (%)
FP16			0.00	0.0%
NF4				
GPTQ				
AWQ				
HQQ				

Table 12: Perplexity: GPTQ calibration dataset comparison

Calibration Dataset	Perplexity	Loss	Delta PPL vs FP16	Degradation (%)
GPTQ-WikiText				
GPTQ-RAG				

4.2.2 Reasoning Tasks

Table 13: Reasoning benchmark performance (0-shot)

Method	**Hella-Swag**	**Wino-Grande**	**PIQA**	**ARC-Easy**	**ARC-Chal.**	**Bool Q**	**Avg**
FP16							
NF4							
GPTQ							
AWQ							
HQQ							

Table 14: Reasoning benchmarks: GPTQ calibration dataset comparison

Calibration Dataset	**Hella-Swag**	**Wino-Grande**	**PIQA**	**ARC-Easy**	**ARC-Chal.**	**Bool Q**	**Avg**
GPTQ-WikiText							
GPTQ-RAG							

4.2.3 Knowledge and Mathematical Reasoning

Table 15: Knowledge and mathematical reasoning performance

Method	**MMLU (5-shot)**	**GSM8K (5-shot)**	**TruthfulQA (0-shot)**	**Avg**	**Avg Deg. (%)**
FP16					0.0%
NF4					
GPTQ					
AWQ					
HQQ					

Table 16: Knowledge and math: GPTQ calibration dataset comparison

Calibration Dataset	**MMLU (5-shot)**	**GSM8K (5-shot)**	**TruthfulQA (0-shot)**	**Avg**	**Avg Deg. (%)**
GPTQ-WikiText					
GPTQ-RAG					

4.2.4 RAG-Focused Reading Comprehension

Table 17: RAG-focused reading comprehension tasks

Method	SQuAD	NQ-Open	TriviaQA	DROP	QuAC	Avg
FP16						
NF4						
GPTQ						
AWQ						
HQQ						

Table 18: RAG-focused tasks: GPTQ calibration dataset comparison

Calibration Dataset	SQuAD	NQ-Open	TriviaQA	DROP	QuAC	Avg
GPTQ-WikiText						
GPTQ-RAG						

4.3 RAG-Specific Performance Analysis

4.3.1 Attention Preservation

Table 19: Attention preservation to relevant documents

Method	Precision @1	Mean Rank	Median Rank	Gini Coeff	EM Acc	F1 Score
FP16						
NF4						
GPTQ						
AWQ						
HQQ						

Table 20: Attention preservation: GPTQ calibration dataset comparison

Calibration Dataset	Precision @1	Mean Rank	Median Rank	Gini Coeff	EM Acc	F1 Score
GPTQ-WikiText						
GPTQ-RAG						

4.3.2 Context Degradation Analysis

Table 21: Accuracy degradation with increasing context length

Method	**Slope per 1K tokens**	**R-squared**	**Cliff Point (tokens)**	**Signif. Deg.?**	**Position Effect?**
FP16					
NF4					
GPTQ					
AWQ					
HQQ					

Table 22: Context degradation: GPTQ calibration dataset comparison

Calibration Dataset	**Slope per 1K tokens**	**R-squared**	**Cliff Point (tokens)**	**Signif. Deg.?**	**Position Effect?**
GPTQ-WikiText					
GPTQ-RAG					

4.3.3 Context Length Breakdown

Table 23: Accuracy by context length (F1 score)

Method	**512 tok**	**1024 tok**	**2048 tok**	**4096 tok**	**Degradation 512 to 4096**
FP16					
NF4					
GPTQ					
AWQ					
HQQ					

Table 24: Context length breakdown: GPTQ calibration comparison

Calibration Dataset	**512 tok**	**1024 tok**	**2048 tok**	**4096 tok**	**Degradation 512 to 4096**
GPTQ-WikiText					
GPTQ-RAG					

4.3.4 Attention Drift During Generation

Table 25: Attention stability during generation

Method	Mean Drift	Max Drift	Drift from Relevant	Drift-F1 Corr (r)	Drift@ Correct	Drift@ Wrong
FP16						
NF4						
GPTQ						
AWQ						
HQQ						

Table 26: Attention drift: GPTQ calibration dataset comparison

Calibration Dataset	Mean Drift	Max Drift	Drift from Relevant	Drift-F1 Corr (r)	Drift@ Correct	Drift@ Wrong
GPTQ-WikiText						
GPTQ-RAG						

4.3.5 Attention-Quality Correlation Analysis

Table 27: Correlation between attention metrics and answer quality

Method	Prec@1 vs F1 (Pearson r)	Rank vs F1 (Spearman rho)	High Attn Correct (%)	Low Attn Correct (%)
FP16				
NF4				
GPTQ				
AWQ				
HQQ				

Table 28: Attention-quality correlation: GPTQ calibration comparison

Calibration Dataset	Prec@1 vs F1 (Pearson r)	Rank vs F1 (Spearman rho)	High Attn Correct (%)	Low Attn Correct (%)
GPTQ-WikiText				
GPTQ-RAG				