Comprehensive Evaluation of Quantization Methods for Edge LLM Deployment

Zahraa Selim, Menna Hamed, Wesam Ahmed, Sohyla Said, Sara Basheer, and 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

***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.**

***Index Terms*—LLM Quantization, Edge Computing, Mistral-7B, Efficiency Benchmarking, Post-Training Quantization, Hardware Compatibility**

I. Introduction

The democratization of Large Language Models (LLMs), as exemplified by architectures such as Mistral-7B [1], holds the potential to introduce advanced reasoning capabilities to edge environments. Nonetheless, the shift from cloud-scale infrastructure to consumer-grade hardware, such as the NVIDIA T4 and RTX series, is impeded by a fundamental resource constraint: memory bandwidth. A typical 7-billion parameter model in half-precision (FP16) necessitates approximately 14GB of Video Random Access Memory (VRAM) solely for weights, thereby leaving minimal capacity for the dynamic memory allocation required during inference.

While 4-bit Post-Training Quantization (PTQ) techniques such as GPTQ [2] and AWQ [3] theoretically reduce model size by approximately 70%, existing literature predominantly evaluates these methods via perplexity (PPL) or zero-shot accuracy on general benchmarks. A critical gap exists in understanding how these compression methods perform across diverse task categories and, crucially, how hardware architecture influences their practical effectiveness.

This work was conducted at Egypt-Japan University of Science and Technology (E-JUST).

A. Research Gaps

Current quantization research exhibits three critical limitations:

Hardware-Agnostic Evaluation: Most studies assume kernel support universality, overlooking that quantization methods optimized for Ampere/Ada architectures may experience severe degradation on older Turing GPUs prevalent in edge deployments.

Limited Task Diversity: Evaluations typically focus on perplexity and a narrow benchmark suite, failing to assess task-specific sensitivity across mathematical reasoning, code generation, and knowledge retrieval domains.

Incomplete Efficiency Profiling: Beyond model size and latency, critical metrics like energy consumption, Model FLOPs Utilization (MFU), and Time-to-First-Token (TTFT) remain underexplored, yet are essential for edge deployment viability.

B. Contributions

This study addresses these gaps through three principal contributions:

1) **Hardware-Aware Compatibility Analysis:** We demonstrate that algorithm-hardware compatibility significantly influences performance. Distribution-based methods (NF4) provide superior stability on Turing architectures (Tesla T4) compared to methods reliant on specific kernel optimizations.
2) **Comprehensive Task-Specific Evaluation:** Extending beyond perplexity, we assess performance across six task categories: mathematical reasoning, code generation, world knowledge, commonsense reasoning, reading comprehension, and language understanding. We establish that quantization impacts tasks differentially, with mathematical reasoning showing 25% degradation while knowledge retrieval remains robust.

3) **System-Level Efficiency Profiling:** We provide complete efficiency analysis including latency decomposition (prefill vs. decode), energy consumption, MFU, and memory footprint, enabling practitioners to make informed deployment decisions.

The remainder of this paper is organized as follows: Section II reviews related work in model quantization. Section III details our experimental methodology. Section IV presents comprehensive results across efficiency and performance dimensions. Section V discusses implications and provides deployment guidelines. Section VI concludes with future research directions.

II. Related Work

Model compression for large language models (LLMs) has emerged as a critical research area to address substantial computational and memory requirements. We focus our review on quantization techniques, the primary compression paradigm evaluated in this study.

A. Quantization Fundamentals

Quantization reduces numerical precision of model parameters and activations, typically converting from 32-bit floating-point (FP32) or 16-bit floating-point (FP16) to lower-bit integer formats. Two fundamental strategies exist: *quantization-aware training (QAT)* and *post-training quantization (PTQ)*.

1) Quantization-Aware Training

QAT incorporates quantization simulation during training, enabling models to adapt to reduced precision. **LLM-QAT** [?] implements standard QAT through knowledge distillation from full-precision models. **BitDistiller** [?] advances QAT for sub-4-bit precisions through asymmetric quantization and adaptive clipping. **OneBit** [?] achieves 1-bit parameter representation. However, QAT's limitation is substantial retraining cost, leading researchers to integrate Parameter-Efficient Fine-Tuning techniques like LoRA.

2) Post-Training Quantization

PTQ applies quantization after training without retraining costs, making it attractive for resource-constrained practitioners. PTQ methods are categorized by quantization targets:

a) Weight-Only Quantization

This approach compresses only model weights while maintaining full-precision activations. **GPTQ** [2] proposes layer-wise quantization using inverse Hessian information for 3/4-bit quantization. **QuIP** [?] achieves 2-bit quantization through LDL decomposition of the Hessian matrix.

Several methods preserve sensitive weights: **AWQ** [3] stores the top 1% most impactful weights in high-precision with per-channel scaling. **OWQ** [?] preserves weights sensitive to activation outliers. **SpQR** [?] uses L2 error as a sensitivity metric. **SqueezeLLM** [?] introduces sensitivity-based weight clustering using k-means.

b) Weight-Activation Quantization

This extends quantization to both weights and activations for true end-to-end low-precision inference. **ZeroQuant** [?] pioneered this approach with group-wise weight quantization and token-wise activation quantization for INT8. **LLM.int8()** [?] addresses outliers by storing outlier features in high-precision. **SmoothQuant** [4] uses per-channel scaling to smooth activation outliers. **OmniQuant** [?] shifts quantization challenges from activations to weights through clipping threshold optimization.

c) On-the-Fly Quantization

Half-Quadratic Quantization (HQQ) [?] facilitates rapid, calibration-free quantization suitable for dynamic deployment, optimizing quantization parameters on-the-fly via a custom loss function.

B. Emerging Methods

AutoRound represents recent advances in adaptive quantization, using iterative optimization to determine optimal quantization parameters per layer based on reconstruction error minimization.

C. Research Gap

While these methods demonstrate theoretical compression effectiveness, systematic evaluation on consumer-grade edge hardware with comprehensive efficiency profiling remains limited. Our work fills this gap by providing hardware-aware benchmarking on Tesla T4 with complete task-specific and efficiency analysis.

III. Methodology

A. Base Model and Hardware

This study utilizes **Mistral-7B-v0.1** [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).

B. Compression Techniques

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"
)
```

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 [2].

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 [5].

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:

```
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:

```
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) *AWQ (Activation-aware Weight Quantization)*

AWQ protects salient weights based on activation magnitudes, preserving the top 1% most impactful weights with per-channel scaling [3].

Configuration:

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

Implementation:

```
from awq import AutoAWQForCausalLM

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

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:

```
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'
)
```

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"
)
```

C. Evaluation Metrics

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

1) Efficiency Metrics

a) 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

b) Throughput and Memory

- **Throughput:** Tokens per second (tok/s)
- **Peak Memory:** Maximum GPU memory (MB)
- **Model Size:** Disk storage requirements (GB)

c) Computational Efficiency

- **Model FLOPs Utilization (MFU):** Percentage of theoretical peak hardware FLOPs achieved:

$$\text{MFU} = \frac{\text{Achieved FLOPs/s}}{\text{Peak Hardware FLOPs/s}} \times 100\% \quad (5)$$

- **Energy Consumption:** Estimated energy per token (mJ):

$$E = (P_{\text{TDP}} - P_{\text{idle}}) \times t_{\text{inference}} \quad (6)$$

where $P_{\text{idle}} \approx 0.3 \times P_{\text{TDP}}$

2) Performance Benchmarks

We evaluate models using established language modeling benchmarks:

a) Language Modeling

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

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

b) 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 [6] with consistent hyperparameters for reproducibility.

IV. Experimental Results

A. Computational Efficiency Analysis

Table I presents comprehensive efficiency metrics on Tesla T4 hardware.

TABLE I
Efficiency Metrics on NVIDIA Tesla T4

Method	Disk (GB)	Peak Mem (MB)	TTFT (ms)	Decode (ms/T)	TPut (T/s)	Energy (mJ/T)
FP16	13.49	7013	68.84	61.36	15.84	3086
NF4	3.74	1859	177.16	79.09	12.30	3975
AWQ	3.87	1869	79.25	77.22	12.79	3855
HQQ	3.74	4721	664.73	649.84	1.51	32421
GPTQ	3.87	4415	1113.00	1117.76	0.88	55703

Key Findings:

1) **Hardware Compatibility Crisis:** GPTQ and HQQ exhibit severe performance degradation on Tesla T4, with throughputs below 2 tok/s and energy consumption exceeding 30,000 mJ/token. This represents a **10-18× slowdown** compared to NF4/AWQ, rendering them impractical for this hardware.
2) **NF4 vs. AWQ Trade-off:** NF4 achieves 12.30 tok/s with 3975 mJ/token, while AWQ achieves 12.79 tok/s with 3855 mJ/token. AWQ provides marginal efficiency gains (3% better energy), but as shown in Section IV-B, this comes at the cost of task performance.
3) **Memory Efficiency:** All 4-bit methods achieve approximately 3.6× compression (13.49 GB → 3.7-3.9 GB), validating theoretical compression ratios. However, peak memory usage varies significantly, with HQQ and GPTQ requiring 2.5× more runtime memory due to kernel overhead.

B. General Task Performance

Table II presents performance across reasoning benchmarks.

Performance Summary:

Task-Specific Analysis:

TABLE II
PERFORMANCE COMPARISON ACROSS BENCHMARKS

Method	**Perplexity** (↓)	**HellaSwag** *(0-shot)*	**ARC-Easy** *(0-shot)*	**ARC-Challenge** *(0-shot)*	**GSM8K** *(8-shot)*	**MMLU** *(5-shot)*	**HumanEval** *(0-shot)*
FP16	12.79	0.72	0.76	0.58	0.36	1.00	0.05
NF4	13.02	0.70	0.75	0.58	0.27	0.55	0.05
GPTQ	12.85	0.68	0.75	0.60	—	—	0.05
AWQ	13.47	—	—	—	—	—	—
HQQ	13.50	0.69	0.72	0.50	—	—	—

TABLE III
AVERAGE ACCURACY AND DEGRADATION

Method	**Avg Acc**	**PPL Inc**	**Acc Drop**	**Tasks**
FP16	0.57	—	—	6
NF4	0.52	+1.80%	-0.05	6
GPTQ	0.52	+0.47%	-0.05	6
AWQ	—	+5.32%	—	6
HQQ	—	—	—	6

1) **Knowledge Retrieval Robustness:** ARC-Challenge and ARC-Easy maintain near-FP16 performance (0.58 vs 0.58 for NF4), indicating that factual knowledge retrieval is preserved under quantization.
2) **Mathematical Reasoning Degradation:** GSM8K experiences 25% accuracy drop (0.36 → 0.27), suggesting multi-step reasoning is more sensitive to quantization than pattern-matching tasks.
3) **Code Generation Stability:** HumanEval maintains perfect parity (0.05 across all methods), likely due to the task's low baseline difficulty for 7B models.
4) **Perplexity Mismatch:** Despite similar perplexity scores, methods show different task performance profiles, confirming that perplexity alone is insufficient for evaluating practical model quality.

V. DISCUSSION

A. Hardware-Algorithm Compatibility as Primary Factor

Our most critical finding is that **hardware compatibility dominates compression effectiveness**. While theoretical compression ratios are identical (3.6× for all 4-bit methods), practical performance varies by 18× on Tesla T4.

Hypothesis: GPTQ and HQQ rely on INT4 tensor cores and optimized GEMM kernels introduced in Ampere architecture. On Turing GPUs (Tesla T4), these operations fall back to inefficient FP16 emulation, causing severe overhead.

Implication: For edge deployment on consumer/previous-generation GPUs, **distribution-based quantization (NF4) is the only viable option**. Kernel-dependent methods should be reserved for latest-generation hardware.

B. Task-Dependent Sensitivity

Quantization impacts tasks differentially:

- **Preserved:** Knowledge retrieval (ARC), commonsense reasoning (HellaSwag)
- **Degraded:** Mathematical reasoning (GSM8K -25%)
- **Stable:** Code generation (HumanEval)

This suggests that quantization affects **deep reasoning chains** more than **pattern matching**. For applications requiring complex multi-step logic, either fine-tuning or hybrid precision strategies may be necessary.

C. The NF4 vs. AWQ Trade-off

AWQ offers marginal efficiency gains (3% lower energy) but shows higher perplexity degradation (+5.32% vs +1.80% for NF4). For general-purpose deployment, **NF4 provides better performance preservation** despite slightly higher energy consumption.

However, AWQ may be preferable for:

- Latency-critical applications where 3% speedup matters
- Tasks where perplexity degradation is acceptable
- Scenarios prioritizing energy efficiency over quality

D. Deployment Guidelines

Based on our comprehensive analysis, we provide prescriptive recommendations:

TABLE IV
QUANTIZATION METHOD SELECTION GUIDE

Scenario	**Recommended Method**
Turing GPUs (T4, RTX 20-series)	**NF4** (only viable option)
Ampere+ GPUs (A100, RTX 30/40)	AWQ or GPTQ (test both)
Math-heavy applications	NF4 + targeted fine-tuning
Knowledge retrieval	Any 4-bit method
Maximum energy efficiency	AWQ (if compatible)
General-purpose deployment	**NF4** (best balance)

VI. CONCLUSION

This study provides the first comprehensive hardware-aware benchmarking of modern quantization methods on consumer-grade edge hardware. Our key contributions include:

1) **Hardware Compatibility Analysis:** Demonstrating that algorithm-hardware compatibility is the primary determinant of practical performance, with kernel-dependent methods experiencing 10-18× slowdown on Turing GPUs.
2) **Task-Specific Evaluation:** Establishing that quantization impacts tasks differentially, with mathematical reasoning showing 25% degradation while knowledge retrieval remains robust.
3) **Prescriptive Guidelines:** Identifying NF4 as the optimal method for Tesla T4 and similar hardware, achieving 3.6× compression with minimal performance loss and superior hardware compatibility.

For Turing-generation hardware, NF4 represents the **only viable quantization method**, achieving stable 12.30 tok/s throughput with reasonable energy consumption (3975 mJ/-token). While AWQ offers marginal efficiency gains, NF4's superior task performance and hardware stability make it the recommended choice for general-purpose edge deployment.

Future Work: Investigating hybrid precision strategies where critical layers (attention heads for reasoning tasks) retain higher precision while feedforward networks are aggressively quantized may resolve the mathematical reasoning degradation observed in our study.

ACKNOWLEDGMENTS

This research was conducted at Egypt-Japan University of Science and Technology (E-JUST). We thank the Anthropic team for access to computational resources.

REFERENCES

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

[2] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "Gptq: Accurate post-training quantization for generative pre-trained transformers," in *Proceedings of the 40th International Conference on Machine Learning*. PMLR, 2023, pp. 10 017–10 030.

[3] J. Lin, J. Tang, H. Tang, S. Yang, X. Dang, and S. Han, "Awq: Activation-aware weight quantization for llm compression and acceleration," *arXiv preprint arXiv:2306.00978*, 2023, related quantization method.

[4] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han, "Smoothquant: Accurate and efficient post-training quantization for large language models," *arXiv preprint arXiv:2211.10438*, 2023, related quantization method.

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

[6] L. Gao, J. Tow, B. Abbasi, S. Biderman, S. Black, A. DiPofi, C. Foster, L. Golding, J. Hsu, A. Le Noac'h *et al.*, "A framework for few-shot language model evaluation," *Version v0.4.0*, 2023, lm-eval-harness for task evaluation. [Online]. Available: https://github.com/EleutherAI/lm-evaluation-harness