LLM Compression and Deployment on Raspberry Pi 4 Using llama.cpp

Abubakar Minhas Sana Humayun Muhammad Rayed Hammad Yousaf
https://github.com/sanahumayun/LLM-Compression.git

Abstract

Large Language Models (LLMs), such as Llama 3.1 8B, offer state-of-the-art natural language processing capabilities but present significant deployment challenges on resource-constrained hardware. Standard implementations of Llama 3.1 8B require approximately 16GB of RAM in half-precision (fp16) and up to 32GB in single-precision (fp32) to load and perform inference. This substantial memory footprint renders deployment on common edge devices, such as the Raspberry Pi 4, impossible due to their strict hardware limitations. In this work, we address the challenge of running high-performance LLMs on edge hardware by compressing Llama 3.1 8B to fit within the 8GB RAM capacity of a Raspberry Pi 4. By employing model compression techniques and utilizing the `llama.cpp` framework for optimized C++ inference, we successfully reduced the model's memory requirements.

1. Introduction

Large Language Models have revolutionized natural language understanding, but their size prevents deployment on edge devices like the Raspberry Pi 4. This report details the pruning and quantization process required to bring the memory footprint of Llama 3.1 8B below the 8GB threshold.

2. Literature Review

Current research into LLM deployment at the edge focuses on structural pruning and post-training quantization. While quantization reduces the bit-precision of weights, pruning physically removes redundant parameters, providing a cumulative effect on memory savings.

The deployment of LLMs on edge devices is severely constrained by memory bandwidth and storage limitations. While Post-Training Quantization (PTQ) offers a solution by reducing weight precision, standard methods often degrade model performance significantly at low bit-widths (e.g., 3-bit or 4-bit). Addressing this, Lin et al. (2024) proposed Activation-aware Weight Quantization (AWQ), a hardware-efficient framework that improves quantization accuracy without relying on expensive retraining or weight reconstruction.

The core contribution of AWQ lies in its novel definition of weight importance. The authors observed that weight magnitudes alone are poor indicators of saliency; instead, the importance of a weight is directly correlated with the magnitude of its activation. By identifying the top 1% of "salient" weights based on activation distributions, the authors demonstrated that preserving these weights is crucial for minimizing quantization error.

Unlike mixed-precision approaches that keep sensitive weights in FP16, which causes hardware inefficiencies, AWQ utilizes an activation-aware scaling method. The framework mathematically derives optimal per-channel scaling factors to minimize quantization noise for salient weights while maintaining a uniform integer format for the entire model. This distinguishes AWQ from previous state-of-the-art methods, such as GPTQ. While GPTQ utilizes second-order information for error compensation, it relies on a reconstruction process that can overfit the calibration set, potentially harming the model's generalization capabilities. In contrast, AWQ does not depend on backpropagation or reconstruction so it preserves the model's generalization abilities.

The authors' empirical results show that AWQ consistently outperforms Round-to-Nearest (RTN) baselines with mixed-precision techniques across the Llama and OPT model families. Furthermore, the study introduced "TinyChat," a specialized inference system designed to implement AWQ. By making use of kernel fusion and efficient weight packing, TinyChat translated the theoretical memory savings of AWQ into practical 3x-4x speedups over FP16 implementations on various edge platforms, including mobile GPUs and Raspberry Pi devices.

2.1. Structural Pruning and Post-Training Recovery

Structural pruning of Large Language Models reduces computational costs by removing entire architectural components—neurons, attention heads, or layers—rather than individual weights, maintaining hardware efficiency through dense matrix operations.

Importance Estimation Methods form the foundation of effective pruning. Traditional magnitude-based approaches using L_1 or L_2 norms often yield suboptimal results for transformer architectures. More sophisticated gradient-based methods employ Taylor expansion with Hessian approximations to estimate parameter importance. However, second-order methods become impractical for LLMs due to $O(N^2)$ complexity with billions of parameters, making first-order gradient methods more suitable.

Dependency-Aware Pruning addresses coupled structures in transformer architectures by identifying groups of parameters that must be pruned together to maintain architectural integrity. This is critical for modern architectures with skip connections and multi-head attention, where removing individual components creates dimension mismatches. Dependency graphs ensure that pruning one neuron triggers removal of all dependent neurons, preventing architectural inconsistencies.

Post-Training Recovery restores model performance after compression. Traditional fine-tuning requires substantial computational resources—TinyBERT requires approximately 14 GPU days for general distillation. LoRA (Low-Rank Adaptation) offers an efficient alternative by freezing the base model and training only low-rank adapter matrices, reducing trainable parameters by 99%+ while maintaining effective adaptation. For LLM compression, LoRA enables recovery in 2-3 hours with only 50K samples compared to days of training on millions of samples for full fine-tuning.

2.2. Efficient Inference Frameworks

llama.cpp and GGUF Format: Traditional LLM inference frameworks rely on GPU-accelerated environments with substantial VRAM requirements. Llama.cpp addresses this limitation by providing a pure C/C++ implementation optimized for CPU inference on consumer hardware. The framework introduces the GGUF (GPT-Generated Unified Format) container format, which supports memory-mapped file I/O, allowing models larger than available RAM to be loaded directly from disk. This architecture is critical for edge deployment, as it prevents out-of-memory crashes on resource-constrained devices. llama.cpp implements efficient quantization schemes (K-quants) that apply mixed-precision strategies at the tensor level, preserving critical attention and output projection layers at higher bit-widths while aggressively compressing less sensitive components. The framework's modular design enables custom architecture definitions through metadata headers, making it compatible with non-standard model structures resulting from targeted pruning strategies.

2.3. Outlier Suppression Techniques

SpinQuant: In LLMs, activations and weights are not distributed evenly; ”outliers” appear in specific channels, forcing quantization scales to be very wide. SpinQuant applies Orthogonal Rotations to model activations. Mathematically, rotating a vector changes values but preserves ”energy” (Norm). SpinQuant learns four rotations: R1 (hidden states), R2 (attention heads), and R3/R4 (FFN MLP layers). Unlike previous rotation-based methods that use fixed random rotations (e.g., QuaRot), SpinQuant introduces the first framework to learn and optimize these rotation matrices specifically for the target model. This approach has been shown to reduce the performance gap to full precision compared to random rotation techniques

3. Methodology

3.1. Structural Pruning Strategy

We implement an MLP-only pruning strategy targeting intermediate dimensions of blocks in layers 4 through 27. Slicing dimensions from 14,336 to 10,752 (25% reduction) avoids GQA head-count mismatch issues and maintains GGUF compatibility. This conservative layer selection (4-27) preserves critical early layers (0-3) responsible for token embedding processing and late layers (28-31) handling output projection. The 25% MLP reduction translates to an overall 13.16% parameter reduction (8.03B $\rightarrow$ 6.97B) while maintaining all 32 attention heads per layer intact, avoiding the 4:1 query-to-KV head ratio constraints of GQA.

3.2. Importance Estimation and Dependency Handling

We utilize a Taylor-based importance criterion:

$$I(W) = \left| W \cdot \frac{\partial \mathcal{L}}{\partial W} \right|$$

We patched `torch_pruning` to correctly handle Llama 3.1 `tuple` outputs, ensuring gradient flow tracking through the first element of the output tuple. Calibration was performed on 10 samples from WikiText-103, computing gradient statistics over forward-backward passes.

3.3. Post-Training Recovery (Fine-tuning)

We perform post-training using LoRA (Rank $r = 8$, Alpha $= 16$) on `yahma/alpaca-cleaned` for 2 epochs on an NVIDIA A100 (80GB) via Modal. Training used learning rate 1×10^{-4}, batch size 64, and gradient accumulation over 16 steps. The base pruned model remained frozen, training only 72.3MB of adapter parameters.

3.4. Outlier Suppression Implementation (SpinQuant)

We applied the `optimize_rotation.sh` script to the pruned FP16 model for 50 epochs. Learnt rotations (R1, R2) were fused into the model, while R3/R4 were bypassed due to mathematical constraints with non-power of 2 dimensions. Changes were made to `modelling_llama_quant.py` and `Optimize_rotation.py` to allow mixed-dimension loading and ignore size mismatches.

3.5. GGUF Conversion and Architecture Adaptation

After the pruning phase, the model was in a raw PyTorch format. The ”MLP-only” pruning strategy created a unique challenge because it removed neurons non-uniformly across the network. This resulted in an irregular architecture where different blocks contained Feed-Forward layers of varying dimensions. Standard inference engines including llama.cpp rely on pre-calculated memory offsets based on uniform layer sizes, so loading this ”jagged” model structure typically causes immediate segmentation faults or dimension mismatch errors.

In our previous attempts in using llama.cpp as is, we were successful in converting it into GGUF format and even quantizing it, but running inference showed us that the model was indeed corrupted and invalid. To address this, we implemented a custom conversion pipeline to migrate the model into the GGUF ecosystem. The adaptation process required two steps:

Checkpoint Sanitization: The raw checkpoint from the pruning process included auxiliary tensors and masking buffers used by the LLM-Pruner library. We developed a cleaning script to identify and remove these training-specific keys, effectively ”resetting” the model structure to look like a standard Llama 3.1 checkpoint.

Variable-Tensor Mapping: Standard conversion tools assume uniform layer dimensions. We deployed a patched version of the `convert_hf_to_gguf.py` script to handle this irregularity. Our modified script performs a dynamic scan of the model architecture, reading the actual shape of every individual MLP layer and writing a custom metadata header into the GGUF container. This metadata acts as a map for the inference engine, providing exact dimensions for every tensor and preventing memory access errors.

The final output was a 16-bit Floating Point (FP16) GGUF baseline, serving as our ”Ground Truth” for all subsequent quantization efforts.

3.6. Standard Quantization (Static Mixed-Precision)

The FP16 baseline occupied approximately 13 GB, exceeding the physical memory limits of the Raspberry Pi (8GB RAM). To make the model viable for deployment, we utilized Standard K-Quantization to reduce precision.

We prioritized the `Q4_K_M` (4-bit Medium) scheme for its balance between size and accuracy. This is a mixed-precision method designed to balance compression with performance:

- **Critical Tensors:** Weights responsible for the attention mechanism (`output.weight`) and the Feed-Forward Network (`attn_v`) are kept at higher precision, typically 6-bit.
- **Standard Tensors:** Less sensitive weights are compressed to 4-bit precision.
- **Super-Block Scaling:** Weights are grouped into blocks (typically sets of 32), each assigned a floating-point scaling factor to reconstruct values during computation.

This reduced the model size to roughly 3.99 GB. We also generated a `Q3_K_M` version for comparison.

3.7. AWQ / Importance Matrix (IMatrix) Strategy

To push compression further and target file sizes around 3.27 GB (3-bit), we implemented an advanced strategy known as Importance Matrix (IMatrix) quantization, functionally equivalent to AWQ. We utilized the IMatrix implementation within `llama.cpp` instead of official AWQ repositories due to:

1. **Hardware Architecture Incompatibility:** Official AWQ relies heavily on CUDA kernels; `llama.cpp`'s IMatrix is engineered for CPU inference on ARM-based hardware like the Raspberry Pi.
2. **The GGUF Format Requirement:** GGUF supports memory mapping (`mmap`), allowing the Pi to read from disk rather than crashing from full RAM loading.
3. **Handling Variable-Layer Sparsity:** Since our model was pruned non-uniformly, native `llama.cpp` quantization tools ensured custom architecture definitions were respected.

Standard quantization is often ”blind” as it treats every weight value as equally important. Our approach corrected for this through a data-driven process:

- **Calibration Phase:** We ran the uncompressed FP16 model against the WikiText-2 dataset to measure activation magnitude for every neuron.
- **Weighted Quantization:** We generated an ”Importance Matrix” file mapping sensitivity. During conversion to 3-bit, high-importance weights were prioritized

for a lower error margin, while low-impact weights were compressed more aggressively.

This resulted in both an advanced quantization technique as well as mixed precision, since different bit widths were applied according to the imatrix computed.

3.8. Alternative Approaches Explored

Prior to the successful MLP-only strategy, we evaluated three alternative pruning configurations: **(1) Heterogeneous Pruning:** Pruning both MLP and attention across layers 4-30 achieved 19.52% parameter reduction (5.42B parameters, PPL 22.80) but produced variable layer architectures (24 vs. 32 heads) incompatible with GGUF's requirement for uniform tensor dimensions. **(2) Zero-Padding Conversion:** Attempting to restore uniform architecture by padding pruned tensors with zeros resulted in valid GGUF files but catastrophic inference failure, generating only special tokens (`<|im_start|>`, `<|name|>`). Analysis revealed zero-padding disrupted learned weight distributions irreparably. **(3) Uniform All-Layer Pruning:** Pruning all 32 layers uniformly achieved 24.03% reduction (5.12B parameters) and GGUF compatibility, but destroyed generation capability in both FP16 and quantized variants. This confirmed that early (0-3) and late (28-31) layers are critically sensitive to pruning. These failures validated our final approach: preserving attention mechanisms entirely and targeting only middle-layer MLPs (4-27) balances compression with quality preservation.

Table 1. Tensor-wise Bit-width Allocation in GGUF K-Quant Formats

Tensor Type	Q4_K_M Precision	Q3_K_M Precision
Attention.wv (Value)	6-bit (Q6_K)	4-bit (Q4_K)
Attention.wo (Output)	6-bit (Q6_K)	4-bit (Q4_K)
Feed_Forward.w2	6-bit (Q6_K)	4-bit (Q4_K)
Token Embeddings	4-bit (Q4_K)	3-bit (Q3_K)
Output Layer	6-bit (Q6_K)	4-bit (Q4_K)
All Other Tensors	4-bit (Q4_K)	3-bit (Q3_K)
Average Bits Per Weight	~4.85 bpw	~3.66 bpw

Note: Mixed precision ensures that critical tensors for attention and MLP down-projection retain higher fidelity than the base quantization level.

4. Results and Discussion

4.1. Deployment Results

4.2. Model Size and Parameter Count Reduction

The primary objective of this work was to reduce the memory footprint of Llama 3.1 8B to fit within the 8GB RAM capacity of the Raspberry Pi 4. Our multi-stage compression

Table 2. Comparison of Normal vs. AWQ Quantization (Q4_K_M) Performance on Raspberry Pi 4

Model Variant	TTFT (s)	TTML (s)	TPS	Memory (GB)
Standard Q4_K_M	4.35	189.86	0.80	4.99
AWQ Q4_K	3.47	181.25	0.72	4.99

Note: TTFT measures prefill latency. TTLM measures initial load wait. AWQ represents Activation-aware Weight Quantization results. TTML is calculated for a standard 128-token response. AWQ provides a 20.2% reduction in initial latency (TTFT) compared to standard quantization.

pipeline achieved an approximately 80% reduction in total model size.

As shown in Table 3, the base FP16 model (16.0 GB) is inherently incompatible with the target hardware. Structural pruning alone reduced the size to 12.99 GB. It was only through the final mixed-precision quantization pass using `llama.cpp` that we reached the deployment threshold of ~3.99 GB (Q4_K_M) and ~3.27 GB (Q3_K_M).

Table 3. Comparison of Llama 3.1 8B Model Variants and Their Memory Footprints.

Model Variant	Precision / Format	Size (GB)
Llama 3.1 8B Base	FP32	32.0
Llama 3.1 8B Base	FP16	16.0
Llama 3.1 8B Pruned (25%)	FP16	12.99
Llama 3.1 8B Pruned	GGUF (4,6-bit Mixed)	~3.99
Llama 3.1 8B Pruned	GGUF (3,4-bit Mixed)	~3.27

4.3. Linguistic Quality Evaluation

To assess the impact of these aggressive compression steps on model intelligence, we measured perplexity on the Wikitext-2 dataset.

The results in Table 4 demonstrate a clear trade-off: structural pruning increased baseline perplexity from 5.56 to 9.78.

While it was expected that the Activation-Aware variants would achieve better perplexity score, our experiments observed a slight degradation as compared to the Standard baseline. We believe that this was due to our additional pruning step. AWQ works by identifying a small percentage of ”critical” weights to protect while aggressively shrinking the ”useless” ones. However, our pruning process had already physically removed the 25% of least important weights. Because of this, AWQ struggled to optimize the model better than the static quantization techniques.

Table 4. Llama 3.1 8B Linguistic Quality Evaluation: Perplexity (PPL) across Model Variants Using Wikitext-2

Model Configuration	Perplexity (PPL)
FP16 (Base)	5.56
Pruned FP16	11.38
Pruned FP16 Fine-tuned	9.78
Standard Q4_K_M	10.16
AWQ + Q4_K_M	10.24
Standard Q3_K_M	11.00
AWQ + Q3_K_M	11.14

Table 5. Total Parameter Count Reduction Post-Pruning

Model State	Total Parameter Count
Before Pruning	8,030,261,248 (8.03B)
After Pruning	6,973,296,640 (6.97B)
Reduction %	13.16%

4.4. Hardware Efficiency Benchmarks

Finally, we benchmarked the execution speed and efficiency of the compressed variants. The results in Table 6 indicate that while the base model process is memory-bound, our 3-bit AWQ variant achieved a superior throughput of 20.34 TPS on the Intel Ice Lake baseline.

Table 6. Hardware Efficiency Benchmarks on Intel Ice Lake (8 vCPUs) for an input of 512 tokens

Model Variant	TTFT (s)	TTLM (s)	MBU (%)	TPS
Base (FP16)	29.27	60.95	129.8	4.04
25% Pruned (FP16)	20.82	71.21	66.0	2.54
Pruned + Standard Q4_K_M	11.38	30.77	53.0	6.60
Pruned + AWQ + Q3_K_M	6.34	12.63	133.0	20.34

Note: MBU measures the ratio of achieved bandwidth to theoretical peak. Base (FP16) measured on Modal (High-RAM) for control.

4.5. Discussion of SpinQuant Failures

SpinQuant results were unsatisfactory, showing garbage values likely due to unresolved faults in fusion methods. We decided not to implement this outlier suppression in the final pipeline to prevent improper fusing results in garbage values being carried over into the quantization process.

References

[1] ggml-org / llama.cpp: ”LLM inference in C/C++” GitHub.

[2] ”Introduction to ggml” Hugging Face Blog.

[3] Lin, J., Tang, J., Tang, H., Yang, S., Chen, W. M., Wang, W. C., Xiao, G., Dang, X., Gan, C., and Han, S. (2024). AWQ: Activation-aware weight quantization for LLM compression and acceleration. In *Proceedings of MLSys 2024*.

[4] Han, S., Xiao, L., et al. (2022). SmoothQuant: Accurate and efficient post-training quantization for large language models. *arXiv preprint arXiv:2211.10438*.

[5] Liu, Z., Zhao, C., Fedorov, I., Soran, B., Choudhary, D., Krishnamoorthi, R., Chandra, V., Tian, Y., and Blankevoort, T. (2024). SpinQuant: LLM quantization with learned rotations. *arXiv preprint arXiv:2405.16406*.

[6] Shao, W., Chen, M., Zhang, Z., Xu, P., Zhao, L., Li, Z., Zhang, K., Gao, P., Qiao, Y., and Luo, P. (2023). OmniQuant: Omnidirectionally calibrated quantization for large language models. *arXiv preprint arXiv:2308.13137*.

[7] Ma, X., Fang, G., and Wang, X. (2023). LLM-Pruner: On the structural pruning of large language models. *arXiv preprint arXiv:2305.11627*.