A 4-bit optimized inference scheme for LLM based on QAT on ARM platform

Longhao Chen
Zhuoyue Honors College
Hangzhou Dianzi University
Hangzhou, China
0009-0008-7803-8208

***Abstract*—This paper introduces a scheme for accelerating LLM inference on the ARM platform, and achieves 4-bit symmetric quantization without accuracy loss through quantization aware training. On the Yitian 710 experimental platform, the prefill performance is increased by 4.06 times, the decoding performance is increased by 14.9 times, the memory usage is reduced to 1/8 of the original**

***Index Terms*—QAT, LLM, Qwen, ARM**

I. Introduction

Large Language Models (LLMs) have achieved remarkable performance in most natural language processing downstream tasks, such as text understanding, text generation, machine translation, and interactive Q&A. However, the billions or even trillions of model parameters pose significant challenges for efficient deployment of LLMs at the edge. With the growth rate of model parameters far outpacing the improvement in hardware performance, the academic and industrial communities are exploring software and hardware collaborative methods like model compression, dataflow optimization and operator invocation to deploy and run large models under the limited hardware conditions.

Current researches widely adopt model compression techniques and have greatly reduced the computational and memory overhead. Meanwhile, some frameworks that serve as deployment tools for large models can provide optimized deployment strategies based on the characteristics of hardware architectures.

In this work, we explored LLM optimization performance on the ARMv9 computing platform, which is mainstream for intelligent edge usage. We symmetrically quantized the model to 4 bits without any loss of accuracy by using quantization aware training. And, we have modified the default quantization process of the llama.cpp framework to use 4-bit quantization for the `lm_head` and `embedding` layers, which further increases the decoding rate and reduces memory usage.

II. Quantization

A. Llama.cpp default Q4_0 quantization

The default `Q4_0` quantization for llama.cpp is to use a scaling factor shared by every group of 32 weights for quantization. For the embedding layer and the last `lm_head` layer, `Q8_0` quantization is used by default.

B. Modified Llama.cpp quantization method

We noticed that the parameter count of the lm_head layer accounts for approximately 50% of the total model parameter count, and in terms of time, the lm_head layer also requires a considerable amount of time. In order to speed up the computation, we modified the quantizer of llama.cpp to perform 4-bit symmetric quantization on the lm_head layer and embedding layer.

C. Quantization Aware Training

Quantization aware training (QAT) is a model quantization method that introduces Fake quantization during the model training process to simulate the errors caused by quantization. This approach can further reduce the accuracy loss of the quantized model. Fake quantization simulates the quantization round operation during the quantization process, treating this error as training noise. During QAT fine-tuning, the model adapts to this noise to reduce the loss of accuracy when finally quantized to INT4.

Due to time and energy constraints, we only inserted weight quantification in the online layer and did not perform activation quantification. The weight quantification here refers to the conversion of weight parameters in neural networks from high precision (such as FP32) to low precision (such as INT4). Activation quantification refers to the process of converting the output (activation value) of a network layer from high precision to low precision.

D. Our Methodology

Firstly, we modified the code of torchao and torchtune to support symmetric 4-bit quantization and ensure that the definition of the quantized groups is consistent with the Q4_0 definition of llama.cpp. This ensures that the weights quantified by QAT can be used without making significant modifications to llama.cpp.

When quantifying, we use alpaca_cleaned_dataset as the training dataset for quantization perception training. The specific training parameters are shown in the Table I.

After the quantization perception training is completed, we save the model weights in float32 format as safetensor format. Afterwards, we use the modified llama.cpp quantizer to directly quantize the saved float32 weights into Q4_0 format for llama.cpp inference.

TABLE I
PARAMETER

Parameter Name	*Value*
batch_size	3
epochs	4
max_steps_per_epoch	50000
fake_quant_after_n_steps	500
groupsize	32
lr	2e-5
gradient_accumulation_steps	1
max_seq_len	2048

III. SOME FAILED ATTEMPTS

A. Speculative sampling

Speculative sampling is a method to address memory access bandwidth by introducing a "small model" to assist decoding, allowing deployed large models to perform "parallel" decoding and thus improve inference speed. The principle of adding speculative sampling in autoregressive decoding is to use two models, one is the original target model and the other is a much smaller approximation model than the target model.

To conduct speculative sampling, the first step is to train a small draft model. The original Qwen2.5-0.5B model was a model with 24 hidden layers, while our designed draft model only has 5 hidden layers, with the remaining parameters consistent with the original model, in order to maximize the reuse of the computational code. We pre trained our draft model on multiple datasets, followed by supervised fine-tuning, and finally obtained our draft model. The trained draft model can now perform simple dialogue and completion tasks.

Afterwards, we conducted inference testing using the speculative simple in the llama.cpp source code tree. After adjusting some parameters, the prediction hit rate can reach about 50%, and the pre fill performance of the draft model is about 3000 tokens/s, and the decode performance is about 400 tokens/s. But the final performance prefill is 800-900 tokens/s, and the decode performance is about 80 tokens/s, which is lower than the default Q4_0 quantization performance.

We suspect that this is due to insufficient optimization of the computational code in the speculative simple section, but due to time constraints, further exploration was not carried out at that time.

B. Parallel decoding of hidden layer features based on Token

The speculative sampling mentioned above belongs to Token Level autoregressive decoding. The Token Level autoregressive decoding method is more effective in predicting Next Tokens using the hidden layer features of Tokens in autoregressive generation than directly using Token Embedding. This is because Token Embedding is just a simple conversion of text, without deep network feature extraction, and has insufficient expressive power. When using lightweight Draft models, the prediction performance may be compromised; The hidden layer feature of Token refers to the output of the last layer Transformer Layer and the input of LM Head. The hidden layer features, after being computed by deep networks, have stronger expressive power than Token Embedding and are more suitable for sampling.

Therefore, we thought of adding a transformer network behind the hidden layer of the token to predict the next or several tokens, which should achieve higher acceptance rates and lower computational complexity than speculative sampling.

However, due to time constraints, we have not implemented or tested this solution.

IV. EVALUATION

We use the speed of the inference in the Python package `transformers` as a baseline.

The accuracy test is shown in Table II. The memory consumption is shown in Table III. The inference rate test is shown in Table IV.

TABLE II
PRECISION

Test items	***Accuracy(arc_challenge)***	***Accuracy(hellaswag)***	***Accuracy(ceval)***
Baseline	0.3242	0.5214	0.5119
Only Q4_0	0.3319	0.5134	0.4896
QAT without lm_head quantization	0.3336	0.5324	0.5030
QAT with lm_head quantization	0.3379	0.5233	0.4849

TABLE III
MEMORY USAGE

Test items	***Physical memory (MiB)***	***Virtual memory (MiB)***
Baseline	3465.5234	5492.4570
Only Q4_0	586.2852	733.8828
QAT without lm_head quantization	586.2852	733.8828
QAT with lm_head quantization	423.9648	961.6211

TABLE IV
INFERENCE RATE

Test items	***Prefill rate (tokens/s)***	***Decode rate (tokens/s)***
Baseline	247.8398	12.9167
Only Q4_0	969.3600	129.4700
QAT without lm_head quantization	969.3600	129.4700
QAT with lm_head quantization	1003.7800	192.3500