Third Place in AICAS 2024 Challenge: From LLMs Compression to Deployment on ARM CPUs

Ziqiang Chen
Zhejiang Lab
Hangzhou, China
chenzq@mail.ustc.edu.cn

Renhuang Huang
Ant Group
Hangzhou, China
576788582@qq.com

Xuedian Hu
Zhejiang Lab
Hangzhou, China
hxdsdu@163.com

***Abstract*—The Large Language Models (LLMs) is widely employed for tasks such as intelligent assistants, text summarization, translation, and multi-modality on mobile phones. However, the current methods for on-device LLMs deployment maintain slow inference speed, which causes poor user experience. The AICAS 2024 conference hosts a performance optimization competition for LLMs inference, leveraging the ArmV9-based Yitian 710 CPU as the platform, with the aim of fostering technological research and development in this field. In our study, we introduced Log Relative Magnitude (LogRM) as a metric to assess the importance of layers in LLMs and proposed a pruning method that removes redundant layers based on their LogRM scores. We utilize the k-quants low-bit quantization method and develop high-performance matrix multiplication kernels using the new SIMD instruction set to accelerate LLM inference on ARM CPUs. Utilizing our proposed pruning method, low-bit quantization, and kernel optimization strategies, we have achieved an end-to-end inference throughput of 380 tokens/s during the prefill stage and 64 tokens/s in the generation stage for large models on an 8-core CPU. Prefill throughput outperforms PyTorch-based solutions by up to 4.38x, and decode throughput by up to 30.94x. Our approach's applicability is demonstrated through its application on Qwen-7B and Qwen-1.8B, showcasing exceptional inference efficiency on ARM CPUs, which led to a third-place finish in the finals.**

***Index Terms*—large language model, inference system, model compression, arm**

I. Introduction

Large Language Models (LLMs) have demonstrated exceptional proficiency in a wide array of natural language processing tasks, encompassing text comprehension, generation, sentiment analysis, machine translation, and interactive question answering [1]–[5]. Despite their remarkable capabilities, the challenge of effectively deploying these models on edge devices is compounded by their massive parameter counts, which can reach into the billions or even trillions [6]. Moreover, the exponential growth of model parameters has outpaced the advancements in hardware performance, prompting both academic and industrial sectors to seek innovative solutions [1], [7].

These solutions include model compression techniques [8], [9], dataflow optimizations, and operator invocation strategies to facilitate the deployment and operation of large-scale models within the constraints of limited hardware resources.

The escalating demand for edge deployment of large models, coupled with the diverse nature of computing resources, has made the deployment of these models on CPU architectures a pivotal area of development [6], [10], [11]. The prevalence of Arm-based CPUs in edge devices has further underscored the significance of this focus. In light of this, AICAS 2024 has designated the ArmV9-based Yitian 710 CPU as the computational platform for a comprehensive large model performance optimization competition. This initiative is designed to stimulate and enhance technological research and development in this domain.

While the initial phase of the competition does not mandate specific hardware platforms, our team has strategically chosen to employ CPU hardware platforms for optimizing the inference performance of large models. This decision is driven by the competition's objectives and an anticipation of the finals' content. Our efforts are concentrated on Algorithm Innovation and System Optimizations, which encompass a range of strategies such as model compression, low-bit quantization, parallel computing, memory management, operator kernel enhancement, and decoding algorithm refinement. These approaches are aimed at maximizing the efficiency and effectiveness of large model deployment and execution on CPU architectures, thereby propelling the frontiers of computational linguistics and artificial intelligence.

II. Background

A. Transformer-based LLM

Transformer-based LLMs have brought about a significant shift in the field of natural language processing, introducing a new paradigm for understanding and generating human language. At the core of this innovation lies the Transformer architecture, which is founded on the concept of self-attention mechanisms [12], enabling the model to assess the significance of various parts of the input data when making predictions. Mathematically, the self-attention mechanism in Transformers can be described as follows: For an input sequence $X = [x_1, x_2, ..., x_n]$, the Transformer computes a set of queries Q, keys K and values V using linear transformations of X . The self-attention scores are then computed as:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}}) \tag{1}$$

where d_k is the dimension of the keys. This mechanism allows the model to focus on different parts of the input

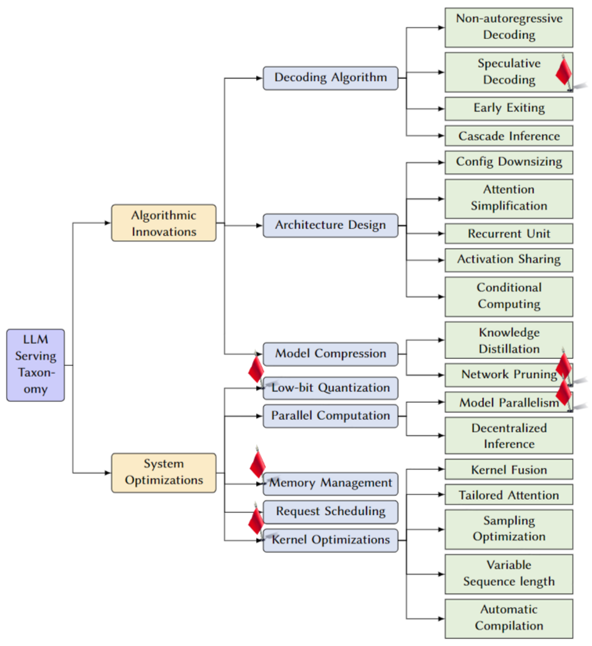

Fig. 1. The modules highlighted with a red flag indicate the specific areas that our team is prioritizing for optimization.

sequence for each element of the output, capturing complex dependencies regardless of their distance in the input sequence.

A transformer consists of multiple Multi-Layer Perceptron (MLP) and Attention blocks. The design of MLP and Attention can be different for each model. Transformer is a sequence to sequence mapping, which can be defined as $y = f(X, \theta)$, where $X \in R^{S\times n}$, $Y \in R^{S\times n}$, S is the length of the sequence, n is the vocabulary size, θ is the learnable parameters. The formal expression of an L-layer transformer is as follows:

$$\begin{aligned} \mathbf{X}_0 &= \mathbf{X}\mathbf{W}_{emb} \\ \mathbf{A}_{i+1} &= ATTN(LN(\mathbf{X}_i)) + \mathbf{X}_i \\ \mathbf{X}_{i+1} &= MLP(LN(\mathbf{A}_{i+1})) + \mathbf{A}_{i+1} \\ \mathbf{Y} &= \mathbf{X}_L\mathbf{W}_{head} \end{aligned} \tag{2}$$

where $\mathbf{W}_{emb} \in R^{n\times d}$ is the word embedding matrix, $\mathbf{W}_{head} \in R^{d\times n}$ is the output projection matrix of the transformer, which are sometimes tied with the $\mathbf{W}_{emb}$, d is the hidden dim of the transformer. $ATTN$ refers to the attention and MLP means the multiple Multi-Layer perceptron, $\mathbf{X}_i \in R^{T\times d}$ is the hidden states of the i^{th} layers.

An inference request to a transformer model consists of a prompt and is served in two phases: (i) *prefill processing*, where the prompt is processed, and (ii) *token generation*, where a series of tokens that represents the output response text is generated incrementally. The model can batch B requests into a single inference task. The sequence length S denotes the number of tokens of each request being processed in the prefill processing phase or the number of tokens of each request being generated in the token generation phase. Therefore, during prefill processing $B \geq 1, S \geq 1$ and during token generation $B \geq 1, S = 1$.

B. Efficient LLM Inference on CPUs

In the realm of artificial intelligence and machine learning, CPUs are increasingly being acknowledged for their prowess in handling inference tasks for large language models (LLMs). While GPUs are renowned for their parallel processing capabilities, CPUs offer distinct advantages owing to their architectural design, which excels in intricate control, decision-making, and context management. Modern CPUs, equipped with advanced instruction sets and enhanced vector processing, are well-suited for the matrix operations central to LLM inference. Intel's Advanced Matrix Extensions (AMX), for instance, bolster vector operation performance, rendering CPUs a competitive choice for neural network computations.

Despite GPUs' superior floating-point operations per second (FLOPs), CPUs provide a cost-effective alternative for inference, especially when considering operational costs and energy efficiency [11]. The ample cache capacity of CPUs proves beneficial for caching key-value pairs, a crucial optimization in transformer-based models, potentially reducing memory access latency and expediting inference. As technology progresses, CPUs are anticipated to become even more competitive in AI, offering a balanced and energy-efficient approach to LLM inference tasks. Moreover, LLMs are often BW bound and have a large weight memory footprint – CPU can achieve competitive performance. Llama.cpp (GGML) c/c++ runtime demonstrates performance on existing Arm platforms but fails to demonstrate the true potential of Arm CPUs [10].

C. Challenges

- **Latency and Response Time.** Efficient inference in large language models is essential for real-time applications such as chatbots and virtual assistants, where low latency and swift response are paramount. Balancing model complexity with inference speed is a critical challenge that necessitates optimizing algorithms and system architectures to minimize response time without compromising accuracy.
- **Memory Footprint and Model Size.** Large language models, with their extensive memory demands due to substantial parameter counts, present a deployment challenge on devices with limited memory. This requires the implementation of sophisticated model compression strategies and system enhancements to maintain performance while significantly shrinking the memory footprint.
- **Scalability and Throughput.** Inference systems must contend with fluctuating request volumes in production settings. To ensure scalable and high-throughput processing of concurrent requests, it is imperative to implement parallel computation and efficient request scheduling, alongside other system-level optimizations. These measures facilitate the optimal distribution of computational workloads across available resources.

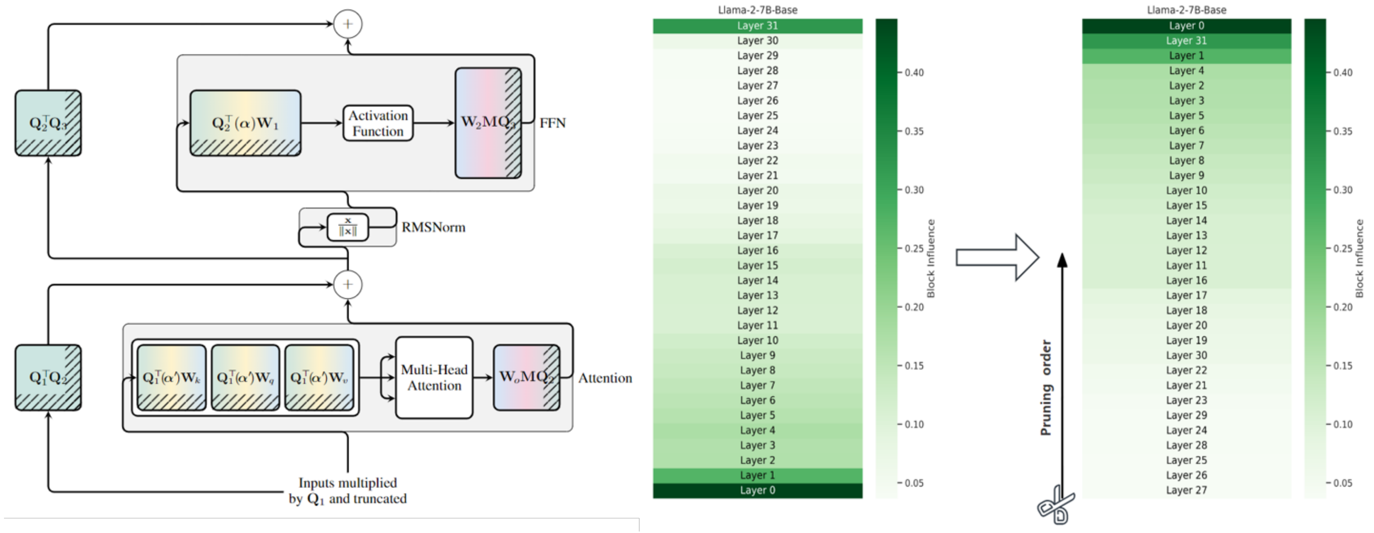

Fig. 2. Left: SliceGPT - Making LLMs Leaner, Right: ShortGPT - Making LLMs Shorter.

- **Trade-offs between Accuracy and Efficiency.** Optimizing the efficiency of LLM inference may sometimes involve trade-offs with model accuracy. Striking the right balance between model size, computational complexity, and performance is a challenging task that requires careful consideration and evaluation of various algorithmic and system-level techniques.

III. Our Approach

Considering these challenges and the prevailing landscape of CPU-based inference for large models, our primary optimization efforts are concentrated in two key areas: **Algorithm Innovation** and **System Optimizations**. Specifically, this includes model compression, model sparsification, low-bit quantization, parallel computing, memory management, operator kernel optimization, decoding algorithm optimization, and more.

A. Model Compression

Model compression aims to reduce the memory footprint and computational requirements of LLMs by creating more efficient and compact models without significant loss in performance. A majority of model compression techniques fall into one of four categories: distillation, tensor decomposition (which includes low-rank factorization), pruning and quantization. Pruning provides a solution to alleviate these resource constraints, and recent works have shown that trained models can be sparsified post-hoc.

Sparsification stands as an essential technique that aims to alleviate the storage and computational burdens associated with neural network models, all while preserving their efficacy. Our team has identified and adopted two state-of-the-art (SOTA) algorithms in this field: SliceGPT [9] and ShortGPT [8]. These algorithms epitomize distinct methodologies for pruning large models along both vertical and horizontal axes.

- **SliceGPT: The Art of Parameter Slicing.** SliceGPT targets the essence of model sparsification by identifying and removing low-impact parameters, grounded in the concept of parameter importance. This method leverages parameter significance analysis to streamline the model, maintaining its predictive power while reducing its complexity.
- **ShortGPT: Streamlining Neural Structure.** ShortGPT focuses on structural efficiency, trimming the neural network's architecture by reducing layer counts. This approach is rooted in controlling model complexity to prevent overfitting, ensuring generalization. ShortGPT also employs strategies like parameter sharing and quantization for additional model compression.

ShortGPT discovered that many layers of LLMs exhibit high similarity, and some layers play a negligible role in network functionality. In [8], [13], they proposed to use relative magnitude (RM) or Block Influence (BI) to measure the importance of layers. Based on this observation, we define a new metric called Log Relative Magnitude (LogRM) to gauge the significance of each layer in LLMs, which performs better on Qwen-1.8B [4].

$$RM = ||\frac{f(x)}{x+f(x)}|| \tag{3}$$

$$BI = 1 - \frac{\mathbf{X}_i^T\mathbf{X}_{i+1}}{||\mathbf{X}_i||_2||\mathbf{X}_{i+1}||_2} \tag{4}$$

$$LogRM = abs(log(\frac{||\frac{f(x)}{x+f(x)}||}{1-||\frac{f(x)}{x+f(x)}||})) \tag{5}$$

Suppose we aim to prune four transformer layers from the Qwen-1.8B model, guided by indicators of layer importance, followed by an assessment of the model's performance using three distinct metrics(RM, BI, LogRM) on the PIQA dataset. The anticipated outcomes of this process are as follows table I.

TABLE I
SETUP OF REMOVED LAYERS FOR QWEN-1.8B AND COMPARISON OF STRATEGIES FOR LAYER REMOVAL ON PIQA.

Method	Removed Layers	PIQA
RM [13]	7,9,8,10	0.6630
BI [8]	20,21,22,11	0.6512
LogRM (ours)	10,11,9,12	**0.6908**

Our empirical evaluation on the PIQA test suite has led us to favor ShortGPT for model sparsification. The comparative result from our experiments is presented in the accompanying table II. It is important to note that the principles underlying these two techniques are complementary, allowing for their synergistic application to further enhance model efficiency.

B. *Low-bit Quantization*

Low-bit quantization techniques provide an efficient means of encoding model weights and activation values, utilizing fewer bits than the conventional 32-bit precision to represent numerical data. This approach markedly decreases memory requirements and enhances the speed of inference processes on various hardware platforms. The primary strategies within quantization are succinctly divided into two categories: Quantization-Aware Training (QAT) and Post-Training Quantization (PTQ). Specifically, PTQ employs tailored kernels to decrease the precision of model weights and activations to INT8 or INT4, thereby achieving a balance between performance and computational efficiency.

We have adopted the K-quants [14] quantization method, implemented in llama.cpp, as a Post-Training Quantization (PTQ) technique for neural network models. Its objective is to reduce model size and enhance operational efficiency while maintaining model performance. The K-quants quantization method achieves model compression by reducing the bit-width of model parameters to between 2 and 8 bits. This approach provides a variety of quantization strategies, allowing users to select the optimal quantized model based on their computational resource limitations, such as memory capacity and inference speed. The essence of quantization is to map continuous weight parameters to a finite set of discrete values, thereby reducing storage requirements and computational complexity of the model. Llama.cpp/GGML use block quantized formats to store chunks of weight columns and activations, as shown in the figure 3.

In our evaluation, we examined a variety of quantization techniques, specifically focusing on Q2_K, Q4_0, Q4_K_M, and Q8_0, as applied to the Qwen-1.8B model. We conducted a comprehensive comparison based on several key metrics, including the count of parameters, memory footprint, and performance discrepancies as measured by the PIQA metric. The synthesized findings are succinctly presented in the table III.

TABLE II
COMPARISON OF PRUNING METHODS ON PIQA. IN SHORTGPT-N, 'N' REFERS TO THE NUMBER OF PRUNED BLOCK LAYERS.

Method	Memory (MB)	PIQA
Dense(F16)	3503	0.733
SliceGPT(15%)	2999	0.659
SliceGPT(25%)	2625	0.611
ShortGPT-2	3300	0.712
ShortGPT-4	3000	0.691

TABLE III
COMPARISON OF K-QUANTS METHODS ON QWEN-1.8B

Method	Memory (MB)	PIQA
Dense(FP16)	3420.0	0.7312
Q2_K	791.9	0.6502
Q4_0	1040.0	0.7144
Q4_K_M	1150.0	0.7198
Q8_0	1820.0	0.7265

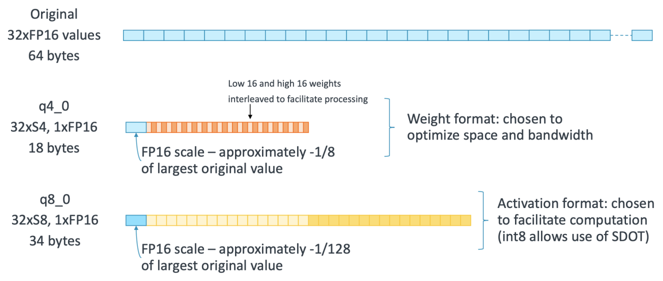
Original
32xFP16 values
64 bytes
Low 16 and high 16 weights
interleaved to facilitate processing
q4_0
32xS4, 1xFP16
18 bytes
FP16 scale – approximately -1/8
of largest original value
Weight format: chosen to
optimize space and bandwidth
q8_0
32xS8, 1xFP16
34 bytes
FP16 scale – approximately -1/128
of largest original value
Activation format: chosen
to facilitate computation
(int8 allows use of SDOT)

Fig. 3. Block Quantized Formats

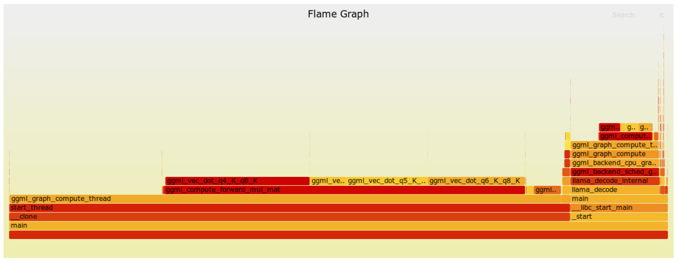

Fig. 4. ARM CPU Flame Graph

C. *Kernel Optimization*

Through testing on ARM CPUs, we found that the GEMV and GEMM kernels play a crucial role in the Qwen-1.8B inference process, as shown in illustration 4, constituting a higher proportion. We concentrate on leveraging more powerful computing instructions and selecting SIMD instruction sets with higher computational peaks to expedite core calculations, particularly in accelerating matrix multiplication operations.

Armv8.6-A has introduced general matrix multiply (GEMM) instructions to both SVE and NEON instruction sets, which reduce memory access and increase computation compared to previous multiplication and multiply-accumulate instructions. In LLM inference, GEMM constitutes a very high computational workload, so utilizing the new instructions

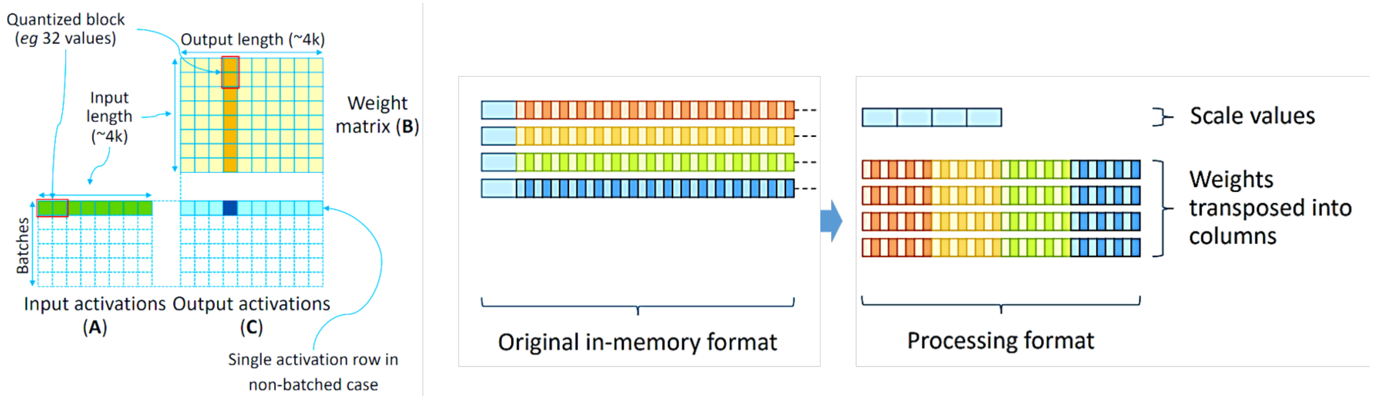

Fig. 5. To avoid pseudo-scalar operations, rearrange parameters in the weight matrix.

in ARMv8.6-A can significantly enhance model inference performance. These instructions have theoretically double the performance capability of ARMv82sdot; using them for int8 matrix multiplication can bring about a substantial performance boost. This article presents an implementation of the MatMul operator using the new instructions from ARMv86, achieving up to approximately 150% performance improvement in prompt calculations.

For typical operators in LLMs, weight matrix (B) is much larger than the input (A) and output (C), as shown in the GEMM on the left side of the figure 5. Compression of weight matrix is key to reducing memory and bandwidth consumption. In GGML, a dot-product kernel computes a single result – it's called at each point to populate the whole of C.

Half of the operations in the original code are scalar or 'pseudo-scalar', operating on a vector of values that are essentially one true value split across lanes. To circumvent the inefficiencies associated with pseudo-scalar operations, we rearrange parameters in the weight matrix. Rather than permuting weights on each iteration, we adopt a blocked memory format for storage, as shown on the right side of the figure 5. This approach embodies the space-time tradeoff paradigm in computer science, enhancing performance through the strategic allocation of memory resources at the expense of increased memory usage. As a result V, we increased the throughput of the prefill stage to 380 token/s, but our memory usage also increased by nearly one-third because we stored the same parameter data in a different order.

D. Speculative Decoding

In LLM inference, traditional autoregressive sampling methods decode tokens sequentially, leading to slow computational processes due to memory access limitations. Speculative sampling techniques address this by adjusting the computation-to-memory access ratio while maintaining the model's sampling distribution. This approach employs a dual-model framework: a large target model and a smaller, more efficient approximate model. The smaller model handles straightforward token generation, while the larger model evaluates outcomes, reducing computational load and memory access requirements.

Our research focuses on Prompt Lookup Decoding [15], a speculative sampling strategy that accelerates generation in large models. Particularly effective for tasks like summarization, content-based question answering, and multi-turn dialogues, where content primarily derives from prompt inputs. By replacing the small model with a content lookup function during speculative sampling, Prompt Lookup Decoding significantly enhances generation speed and simplifies the inference process.

IV. Results

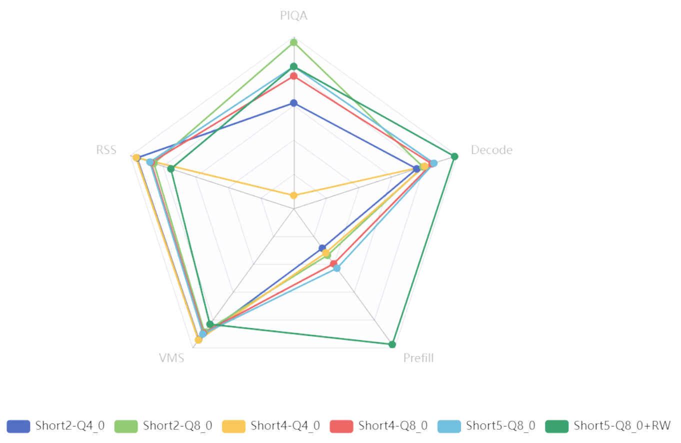

Fig. 6. Performance of different model sparsification and quantization methods. The increase in memory space resulting from the rearrangement of the weight matrix is traded for enhanced prefill throughput, denoted as **+RW**

A. System

Our experimental setup utilized an ARM CPU (Yitian 710) supplied by the event organizers, which boasted a 8-core CPU, 30GB of memory, and 50GB of disk space. Meanwhile, in compliance with the competition's specifications, we validated our optimization method using the Qwen-1.8B [4]. We evaluate the accuracy of both Q4_0 and Q8_0 models using open-source datasets from lm-evaluation-harness [16] including PIQA [17] task. To assess performance, we measure the resident set size (M_{RSS}), virtual memory usage (M_{VMS}), throughput of the prefill stage (T_{pre}), and throughput of the decode stage (T_{dec}). After normalizing each metric separately, calculate the final score using the following formula 6.

$$Score = Acc \cdot 0.30 + RSS \cdot 0.15 + VMS \cdot 0.15 + Pre \cdot 0.15 + Dec \cdot 0.25 \tag{6}$$

B. Accuracy

TABLE IV
PREDICTION ACCURACY OF DIFFERENTLY SPARSED QWEN-1.8B MODELS UNDER VARIOUS QUANTIZATION METHODS. IN SHORT-N, 'N' REFERS TO THE NUMBER OF PRUNED BLOCK LAYERS.

Quant Type	Qwen-1_8B	Short2	Short4	Short5
Q4_0	0.7144	0.7084	0.6823	0.6910
Q8_0	0.7285	0.7255	0.7160	0.7188

We evaluate the accuracy on the aforementioned datasets and show the average accuracy in Table IV. The prediction accuracy after Q8_0 quantization is higher than that of Q4_0. To balance the prediction accuracy, we prefer to use Q8_0 for quantization whenever possible. The number of removed layers affects prediction accuracy; generally, more layers removed leads to greater accuracy loss. However, with proper fine-tuning, we can improve the degraded precision.

C. Performance

We optimized our inference system based on the llama.cpp project. To evaluate the post-optimization performance, we used a 560-tokens text input to generate 50 tokens, measuring the aforementioned performance metrics during this process. We conducted 10 measurements and took the average for each metric, with the results presented in the table V.

TABLE V
LLM PREFILL THROUGHPUT OUTPERFORMS PYTORCH-BASED SOLUTION BY UP TO 4.38X UNDER SHORT5-Q8_0+RW.

Model	PIQA	RSS	VMS	Prefill	Decode	Score
Short2-Q4_0	0.7084	1180	1276	121.69	49.65	60.05
Short2-Q8_0	**0.7255**	1970	2064	140.93	52.01	67.86
Short4-Q4_0	0.6823	**1132**	**1227**	134.34	52.66	47.29
Short4-Q8_0	0.7160	1835	1930	163.67	55.47	64.62
Short5-Q8_0	0.7187	1775	1849	175.55	56.18	66.74
Short5-Q8_0+RW	0.7188	2756	2871	**380.61**	**64.05**	**70.66**

V. SUMMARY AND FUTURE WORK

We presented an end-to-end LLM inference including model compression, low-bit model quantization and efficient kernel optimization. We demonstrated the performance advantage over the open-source solution on ARM CPUs. As our future works, we plan to further improve the ARM CPU tensor library to support LLM inference as part of the contributions to the open-source community. Moreover, leveraging the ubiquity of CPUs, we intend to extend our methodology to edge devices, aiming to satisfy the escalating demand for AI-generated content and augment the capacity for AI computation on ARM CPUs.

ACKNOWLEDGMENT

We extend our sincere thanks to the competition organizers of AICAS 2024 and the Alibaba Cloud Tianchi Competition platform for their exceptional support and contributions.

REFERENCES

[1] X. Miao, G. Oliaro, Z. Zhang, X. Cheng, H. Jin, T. Chen, and Z. Jia, "Towards efficient generative large language model serving: A survey from algorithms to systems," *arXiv preprint arXiv:2312.15234*, 2023. 1

[2] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023. 1

[3] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Codellama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023. 1

[4] J. Bai, S. Bai, Y. Chu, Z. Cui, K. Dang, X. Deng, Y. Fan, W. Ge, Y. Han, F. Huang *et al.*, "Qwen technical report," *arXiv preprint arXiv:2309.16609*, 2023. 1, 3, 5

[5] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin *et al.*, "Opt: Open pre-trained transformer language models," *arXiv preprint arXiv:2205.01068*, 2022. 1

[6] L. Li, S. Qian, J. Lu, L. Yuan, R. Wang, and Q. Xie, "Transformer-lite: High-efficiency deployment of large language models on mobile phone gpus," *arXiv preprint arXiv:2403.20041*, 2024. 1

[7] K. Alizadeh, I. Mirzadeh, D. Belenko, K. Khatamifard, M. Cho, C. C. Del Mundo, M. Rastegari, and M. Farajtabar, "Llm in a flash: Efficient large language model inference with limited memory," *arXiv preprint arXiv:2312.11514*, 2023. 1

[8] X. Men, M. Xu, Q. Zhang, B. Wang, H. Lin, Y. Lu, X. Han, and W. Chen, "Shortgpt: Layers in large language models are more redundant than you expect," *arXiv preprint arXiv:2403.03853*, 2024. 1, 3, 4

[9] S. Ashkboos, M. L. Croci, M. G. d. Nascimento, T. Hoefler, and J. Hensman, "Slicegpt: Compress large language models by deleting rows and columns," *arXiv preprint arXiv:2401.15024*, 2024. 1, 3

[10] I. B. Dibakar Gope, David Mansell, "Large language models on cpus," https://www.pccluster.org/ja/event/pccc23/data/PCCC23_1207_01_arm.pdf, 2023, accessed: Date. 1, 2

[11] H. Shen, H. Chang, B. Dong, Y. Luo, and H. Meng, "Efficient llm inference on cpus," *arXiv preprint arXiv:2311.00502*, Nov 2023. 1, 2

[12] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *Neural Information Processing Systems,Neural Information Processing Systems*, Jun 2017. 1

[13] M. Samragh, M. Farajtabar, S. Mehta, R. Vemulapalli, F. Faghri, D. Naik, O. Tuzel, and M. Rastegari, "Weight subcloning: direct initialization of transformers using larger pretrained ones," *arXiv preprint arXiv:2312.09299*, 2023. 3, 4

[14] ggerganov, "llama.cpp k-quants," https://github.com/ggerganov/llama.cpp/pull/1684, 2024, accessed: Date. 4

[15] A. Saxena, "Prompt lookup decoding," November 2023. [Online]. Available: https://github.com/apoorvumang/prompt-lookup-decoding/ 5

[16] L. Gao, J. Tow, B. Abbasi, S. Biderman, S. Black, A. DiPofi, C. Foster, L. Golding, J. Hsu, A. Le Noac'h, H. Li, K. McDonell, N. Muennighoff, C. Ociepa, J. Phang, L. Reynolds, H. Schoelkopf, A. Skowron, L. Sutawika, E. Tang, A. Thite, B. Wang, K. Wang, and A. Zou, "A framework for few-shot language model evaluation," 12 2023. [Online]. Available: https://zenodo.org/records/10256836 6

[17] Y. Bisk, R. Zellers, J. Gao, Y. Choi *et al.*, "Piqa: Reasoning about physical commonsense in natural language," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 05, 2020, pp. 7432–7439. 6