AICAS 2024 Grand Challenge Technical Document

Hyungmin Kim
Department of Electronic Engineering
Hanyang University
Seoul, Korea
kong4274@hanyang.ac.kr

Jiwoong Park
Department of Electronic Engineering
Hanyang University
Seoul, Korea
pjw9703@hanyang.ac.kr

Sihwa Lee
Department of Artificial Intelligence
Hanyang University
Seoul, Korea
macto94@hanyang.ac.kr

Geonho Lee
Department of Electronic Engineering
Hanyang University
Seoul, Korea
thisisho@hanyang.ac.kr

Sungwan Ryu
Department of Electronic Engineering
Hanyang University
Seoul, Korea
ssk05118@hanyang.ac.kr

Minsoo Kim
Department of Electronic Engineering
Hanyang University
Seoul, Korea
minsoo2333@hanyang.ac.kr

Jungwook Choi
Department of Electronic Engineering
Hanyang University
Seoul, Korea
choij@hanyang.ac.kr

Wonyong Sung
Department of Electronic Engineering
Seoul National University
Seoul, Korea
wysung@snu.ac.kr

***Abstract*—This document is a technical report that includes an introduction to the techniques used in the competition and the deployment processes used in the preliminary and final rounds.**

***Index Terms*—Qwen, Optimization, Quantization, Pruning, etc.**

I. Introduction

This document serves as the technical report for the AICAS 2024 Grand Challenge. The report is divided into two main sections: the preliminary round and the final round. Each section comprises two parts: algorithms aimed at enhancing the efficiency of the model, and optimizations designed to increase throughput on the designated devices. Our finetuned models are available via the HuggingFace Hub [1][2].

We used the following hardware and software environment:

- **Device**: NVIDIA RTX A6000
- **Environment**: CUDA 11.8 / torch 2.0+cu118 / transformers 4.31.0

II. Preliminary Round

A. Introduction

At the preliminary round, we need to optimize Qwen-7B. Participants conduct a performance evaluation, comparing the abilities and efficiencies of the deployed LMMs performance before and after optimization.

B. Algorithm Optimization

We adopted a two-phased approach: First, we used the training dataset samples from benchmarks (ARC-Challenge [1], HellaSwag [2], PIQA [3]) for fine-tuning the Full-Precision (FP) Qwen-7B model to improve its performance. Second, to boost both efficiency and accuracy, we utilized the fine-tuned FP model as a starting point for applying INT4 quantization through Quantization-aware Training (QAT) [4] [5].

1) FP Fine-Tuning: We explored various settings for epochs and learning rates to find the optimal fine-tuning recipe.

2) INT4 Weight Quantization-Aware Fine-Tuning: **Introduction to QA-LoRA:** To enhance the accuracy of the INT4 model through Quantization-Aware Fine-Tuning (QAFT) we employed the QA-LoRA [6]. This involved freezing the base model in INT4 format while updating the LoRA adapters in FP16. The QA-LoRA uniquely allows the LoRA adapter to adjust the zero point of the INT4 base model, ensuring that merging the FP16 adapter with the base weight maintains the INT4 quantization integrity.

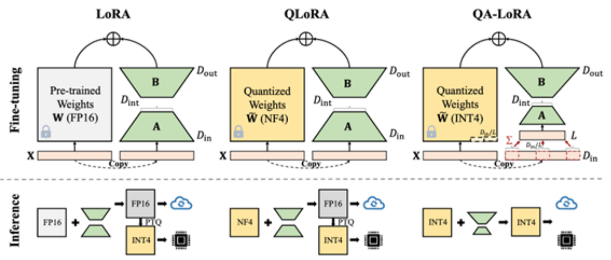

Fig. 1. Illustration of the goal of QA-LoRA

Integrating QA-LoRA Adapters for INT4 Inference: Integrating the QA-LoRA adapters with the base model's quantization framework involved a critical step of merging the adapter weights (LoRA-A and LoRA-B) into the zero points of the INT4 base weights, as shown in Algorithm 1. This process

[1]https://huggingface.co/Ho97/Qwen-7B-FF
[2]https://huggingface.co/Ho97/Qwen-1_8B-FF-E3

TABLE I
QWEN-7B LATENCY AND THROUGHPUT FOR EACH KERNEL ON A100

Kernel Method	Decoding Latency (sec /128 tokens)	Throughput (tokens/sec)
torch FP16 (CPU)	30.7483	4.130
torch FP16 (GPU)	3.0410	52.287
bitsandbytes w4a16 – g128	3.7193	34.415
AutoGPTQ w4a16 - g128	3.3210	38.542
AutoAWQ w4a16 - g128	**2.2760**	**56.239**
QUICK w4a16 - g128	2.3812	53.754
exllama w4a16 - g128	2.2824	56.081

ensured the updated zero points maintained the quantization integrity without distorting the integer arithmetic framework of the base model.

Algorithm 1 LoRA adapter merge

```
 1: Input: $model$
 2: Output: $zeros$ of Linear layers

 3: for module in model do
 4:     if module is Linear then
 5:         Find LoRA adapter $adapter\_A$
 6:         Find LoRA adapter $adapter\_B$
 7:         Find $zeros$ of Linear layer
 8:         $adapter\_weight$ = ($adapter\_A$ @ $adapter\_B$)
 9:         Merge $zeros$ and $adapter\_weight$ of the module
10:     end if
11: end for
12: return Updated $zeros$
```

C. Software Optimization

1) Evaluate Kernels for Optimal Performance: To identify the most efficient method, we have tested all the options available in optimum-benchmark (including a few kernels that were recently released). Since memory usage is expected to be nearly identical at the same bit-precision, we focused our testing primarily on latency and throughput. Based on Table I, we decided to use the AWQ [7] kernel. The AWQ kernel performs linear operations with 4-bit weights and 16-bit activations, using packed weights, packed zeros, and FP16 scales as inputs. Therefore, we have performed post-processing on the fine-tuned model according to the QA-LoRA we described before, ensuring it is compatible with the AWQ kernel.

2) Post-Processing of QA-LoRA Model for AWQ Kernel Compatibility: The QA-LoRA fine-tuned model consists of INT4 base weights, INT4 zero points, FP16 scales, and FP16 adapters. To enable efficient inference using AWQ kernel, it's necessary to merge the adapter with the base weight for single forward path. However, the adapter obtained through QA-LoRA updates the zero point of the quantized base weight, necessitating the post-processing for AWQ kernel compatibility. The post-processing step consists of 3 steps, as shown in Algorithm 2.

Algorithm 2 Post-Processing of $zeros$

```
 1: Input: $zeros$
 2: Output: Updated $qzeros$

 3: for module in model do
 4:     # Unpack zeros
 5:     Right shift $zeros$ to unpack
 6:     Apply bitwise AND to isolate unpacked values
 7:     Convert unpacked $zeros$ to FP16 format
 8:     # Update zeros
 9:     Update $zeros$ with fine-tuned adapter
10:     # Repack updated zeros
11:     for each column in the packed zeros array do
12:         Generate $qzeros$ by repacking $zeros$
13:     end for
14: end for
15: return Updated $qzeros$
```

D. Results

1) Algorithm Optimization: **FP Fine-Tuning:** The fine-tuning experiments revealed variations in accuracy with different learning rates and epochs as shown in Table VI. Training for one epoch at a learning rate of 1E-5 achieved an optimal average accuracy of 62.36%. As shown in Table II, extending training to three epochs with a learning rate of 9E-6 improved the average accuracy to 64.10%.

TABLE II
FP QWEN-7B FINE-TUNING RESULTS WITH LEARNING RATE AND EPOCH EXPLORATION

Fine-Tuning Dataset	Learning rate	Epochs	PIQA	HellaSwag	ARC Challenge	Average Accuracy
PIQA + HellaSwag + ARC Challenge	6E-6	1	79.60	58.03	47.01	61.54
	7E-6	1	79.54	58.10	47.78	61.80
	9E-6	1	79.87	58.08	48.63	62.19
	1E-5	1	79.98	58.06	49.06	62.36
	9E-6	**3**	**79.54**	**62.16**	**50.6**	**64.10**
	8E-6	3	79.76	61.94	50.51	64.07
	7E-6	3	80.03	61.09	50.00	63.70

INT4 Weight Quantization-Aware Fine-Tuning: The application of 4-bit weight quantization on the fine-tuned FP model resulted in a significant initial accuracy drop of 3.32%. However, employing the QA-LoRA mitigated this loss, reducing the accuracy drop to just 1.07%(Table III). This not only maintained higher accuracy relative to the pre-trained model but also reduced the overall model size through quantization.

2) Software Optimization: Through the post-processing steps described above, we were able to create a model with 4-bit weights and enable inference using the AWQ kernel. The effects of quantization and kernel optimization on decoding latency, memory usage, and throughput can be seen in Table IV. We achieved an approximate 25% speed-up and a 32% increase in throughput. Regarding memory usage, through 4-bit weight quantization, we achieved a 50% reduction in memory. The reason we could not achieve a 4 times reduction in the overall memory usage during inference

TABLE III
QA-LoRA EXPERIMENTAL RESULTS AND COMPARISON BETWEEN PRE-TRAINED / FINE-TUNED FP QWEN-7B, AND PTQ 4 BIT QWEN-7B

Method	Learning rate	Epochs	PIQA	HellaSwag	ARC Challenge	Average Accuracy
Pre-Trained FP Qwen-7B	-	-	77.15	57.44	44.11	59.57
Fine-tuned FP Qwen-7B	9E-6	3	79.54	62.16	50.6	64.10
Fine-tuned Qwen-7B + **PTQ(4bit)**	-	-	78.78	57.67	45.9	60.78
Fine-tuned Qwen-7B + **QA-LoRA(4bit)**	9E-6	3	79.49	60.03	49.57	63.03

TABLE IV
COMPARING THE EFFICIENCY OF QWEN-7B BEFORE AND AFTER OPTIMIZATION

	Decoding Latency (sec / 128 tokens)	**Memory** (MB)	**Throughput** (tokens/sec)
Before (torch-FP16)	3.041	21047	41.756
After (AWQ-INT4)	2.296 (x1.24)	10412 (x0.49)	55.309 (x1.32)

(FP16 vs INT4) is due to the requirements for activation memory and the presence of unmanageable elements. (fragmented memory, torch-CUDA initialization memory, etc.). To gain a deeper understanding of memory usage in model inference, we conducted a memory breakdown for the same experimental setting, which is illustrated in Figure 2. Based on our approach, we were able to reduce the model size by approximately 62%. Given the small batch sizes typically employed in model inference serving tasks, the memory allocated for activation does not constitute a major portion. Consequently, decreasing the model size through weight quantization significantly mitigates memory constraints.

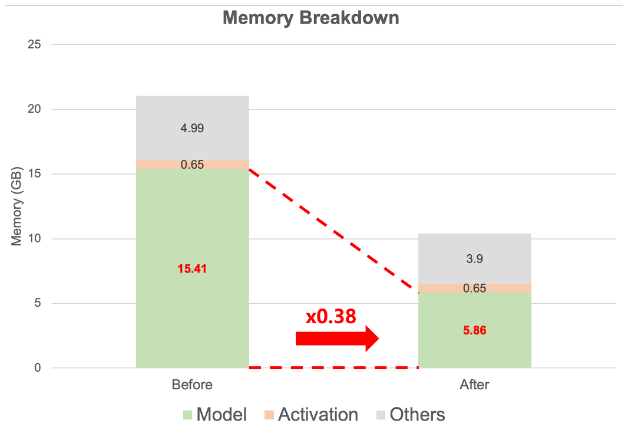

Fig. 2. Inference memory breakdown for before and after optimization

III. Final Round

A. Introduction

During the final stage, the organizing committee provides a specified hardware, the unified YiTian 710 CPU. In the final round, the model participants need to optimize is changed to Qwen-1.8B model. The goal of the final round is optimizing this smaller model in the CPU based platform.

B. Algorithm Optimization

To enhance the inference efficiency of the Qwen-1.8B model while maintaining accuracy, our strategy incorporates a three-phased approach:

a. **Fine-tuning the Full-Precision Model:** We full fine-tuned model with PIQA dataset to optimize the baseline accuracy of the model before any further modifications.
b. **Pruning layers:** With accuracy enhanced Qwen-1.8B model, we adopted layer pruning to reduce model size and increase throughput. With several layer pruning methods, we found best method to prune layers.
c. **Quantization via QAT:** By applying INT4 quantization through QAT, we sought to strike a balance between computational efficiency and model accuracy. Also, we applied knowledge distillation(KD) with QAT to further reduce degradation of accuracy during quantization.

1) Fine-tuning full-precision Model: To increase the the base model's accuracy, we tried to find best fine-tuning recipe by learning rate (LR) sweeping. Unlike conventional methods, we tried LR sweep at each epoch to further increase accuracy.

2) Layer pruning on fine-tuned FP model: Layer pruning is an effective method for reducing memory consumption and improving throughput by removing layers from a model. We explored following layer pruning methods to find optimal method with minimum accuracy drop. As Qwen-1.8B model is composed of 24 decoder layers, removing too many layers can significantly decrease performance. Therefore, we observed the accuracy trends after removing 4, 6, and 8 layers.

3) INT4 Weight Quantization-aware Fine-Tuning: **Quantization-Aware Fine-Tuning for FP Fine-Tuned Qwen-1.8B:** INT4 quantization can reduce the model size and increase throughput without huge degradation of accuracy. To further reduce the degradation of the accuracy drop, we used QAT. From the quantization configuration of the llama.cpp, we use symmetric INT4 quantization with only scale value. We prepared similar quantization code in python to do QAT with same quantization configuration of the llama.cpp. When we trained our model with our QAT code and tested the trained model with llama.cpp quantization, we found that the difference between result was affordable.

Knowledge Distillation after Pruning: Knowledge distillation is commonly used after pruning or quantization because we have both teacher and student model. We used our full fine-tuned FP model with all 24 layers as a teacher model. Student model is layer pruned and fine-tuned model. We referenced the code for QAT and KD from TSLD [8].

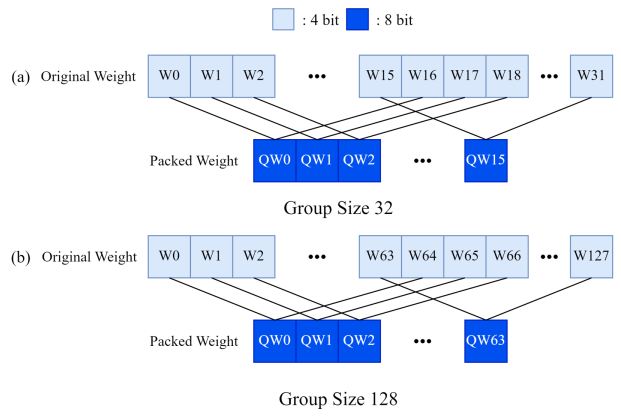

Fig. 3. Quantized Weight Packing Scheme (a) Group Size: 32 (b) Group Size: 128

C. Software Optimization

1) The Importance of Hardware-Specific Development: The necessity of tailoring software development specifically for target hardware in CPU optimization arises from the different performance characteristics they possess. This approach is most important when aiming to leverage the full potential of off-the-shelf CPUs, particularly for complex and resource-intensive tasks such as generative inference in large language models(LLMs). The computational landscape is marked by a diversity of processors, each with distinct capabilities, memory hierarchies, and instruction sets, such as the ARM Advanced SIMD NEON intrinsics. Consequently, generic algorithms, while broad in applicability, often fail to exploit the unique optimizations possible on specific hardware, leading to suboptimal performance and efficiency. For instance, the task of executing generative LLMs on user devices demands not only high accuracy but also high efficiency, given the substantial memory and computational requirements of these models.

2) Optimizing LLM Inference on ARM with llama.cpp: In our pursuit to efficiently deploy the Qwen-1.8B model on ARM-based devices featuring octa-core processors, we leveraged the llama.cpp [9] library. The reason why we utilize llama.cpp [9] is that its sophisticated support for ARM's SIMD intrinsics as shown in Table V, specifically NEON, which are instrumental in maximizing computational throughput and efficiency. By exploiting these intrinsics, llama.cpp [9] significantly optimizes the execution of vectorized operations, a critical factor in enhancing the performance of LLMs on ARM architectures.

Moreover, llama.cpp's [9] architecture is designed to adeptly manage thread-level parallelism, allowing for the concurrent execution of tasks across all 8 cores. This capability is most important in optimizing the computational load distribution, thereby reducing execution times and increasing the throughput of generative inference tasks. The llama.cpp's [9] utilization of manually tailored quantized kernels not only facilitate the reduction of model size without compromising accuracy but also provide a foundation for incorporating additional optimization techniques, such as dynamic scaling factors which can lead to further computational savings.

TABLE V
CONCISE DESCRIPTION OF VECTOR INSTRUCTIONS USED IN A 4-BIT QUANTIZED MATRIX MULTIPLY KERNEL.

Instruction	**Description**
`vld1q_s8(address)`	Loads 8-bit integers into SIMD.
`vdupq_n_s32(a)`	Duplicates 32-bit integer in SIMD.
`vandq_u8(a, b)`	Bitwise AND on 8-bit unsigned.
`vshrq_n_u8(a, n)`	Right shifts 8-bit unsigned by `n`.
`vreinterpretq_s8_u8(a)`	Reinterprets unsigned as signed.
`vsubq_s8(a, b)`	Subtracts 8-bit signed integers.
`vdotq_s32(a, b, c)`	Dot product of 8-bit, adds to 32-bit.
`vmlaq_n_f32(a, b, c)`	Multiplies floats by scalar, adds to register.

3) Enhancing Quantized Matrix Multiply Kernel Efficiency: To maximize the utilization of ARM NEON's SIMD Registers and to enhance memory efficiency, we employ 4-bit quantized weights and leverage meticulously crafted kernels for this purpose. Furthermore, to amplify the efficiency of these optimizations, we expand the Matrix Multiply Kernel, enabling faster data processing. This optimization is feasible due to the utilization of kernels with a large group size, which allows loading more data per memory access, thereby reducing overhead and processing more data in a single instruction loop. Additionally, to maximize the efficiency of the SIMD Registers, we apply packing to INT4 Quantized Weights into an 8-bit data format as illustrated in Fig. 3. These compression result in approximately 4x reduction in memory consumption compared to FP16 model.

The key lies in the balance between hardware capabilities and algorithmic adjustments. By packing quantized weights, not only do we significantly reduce the memory footprint, but we also ensure that the data is more amenable to the SIMD processing paradigms, which are crucial for achieving high throughput. This balance between computational efficiency and memory optimization is central to pushing the boundaries of what is possible in high-performance computing applications, particularly in the context of LLM inference.

D. Results

1) Algorithm Optimization: **Fine-tuning Recipe:** The comparison between conducting single LR sweep for 3 epochs and sweep for each epoch three times is shown in the Table VI. When we swept LR epoch by epoch, it already outperformed the single LR sweep at the second epoch. At third epoch, we could increase the piqa accuracy to 76.06%

Layer Pruning: The results for different pruning methods and settings are shown in the Table VII. The experimental configuration shown in the table below represent the optimal

TABLE VI
LEARNING RATE SWEEP RESULTS

Epoch	Initial PIQA	LR	Trained PIQA
LR Sweep for Each Epoch			
1	73.12	6E-6	74.92
2	74.92	3E-6	75.95
3	**75.95**	**1.9E-7**	**76.06**
Single LR Sweep			
3	73.12	9E-6	75.68

results for pruning. Upon comparing the two methods, we observed that SLEB [10] tends to preserve accuracy better than ShortGPT [11], so we used SLEB for layer pruning. Additionally, we applied fine-tuning to recover the accuracy loss due to layer pruning. Below Table VIII are the best results of our fine-tuning experiments. As indicated in the Table VIII, there is not a significant difference in accuracy between removing 6 layers and 8 layers. Therefore, to maximize the benefits in memory and throughput, we decided to use the model with 8 layers removed.

TABLE VII
LAYER PRUNING RESULTS

Method	The number of deleted layers	The number of remaining layers	PIQA accuracy
-	0	24	75.95
SLEB	4	20	70.62
	6	18	66.70
	8	16	62.89
ShortGPT	4	20	67.63
	6	18	65.51
	8	16	58.79

TABLE VIII
FINE-TUNING RESULTS OF PRUNED MODEL

Method	The number of deleted layers	The number of remaining layers	Learning rate	Epoch	PIQA accuracy
-	0	24	-	-	75.95
SLEB	4	20	1E-5	3	73.07
	6	18	1E-5	3	71.87
	8	16	8E-5	2	71.38

QAT + KD: The results of QAT is shown in Table IX. QAT improves the accuracy of model then just using post training 4bit quantization. KD could further increase the accuracy.

TABLE IX
QAT & KD RESULTS IN W4A8 QUANTIZATION

Baseline	Groupsize	PTQ	QAT	QAT + KD
71.11	32	70.73	71.22	71.33
	128	70.73	71.22	71.33

2) Software Optimization: Considering various factors, we adopted a comprehensive optimization strategy that encompasses algorithmic aspects as well as software considerations, targeting throughput, latency, and model accuracy. Consequently, we selected a model characterized by an exceptionally high compression ratio, achieving low resource requirements without compromising accuracy. Furthermore, while applying layer pruning, we set a cap of eight layers for potential adjustments, allowing for a comparative analysis of model performance under these conditions. Our findings indicate that the application of eight layers of pruning on a W4A8 model utilizing a group size of 128 yielded the most significant performance improvement as illustrated in Fig. 4

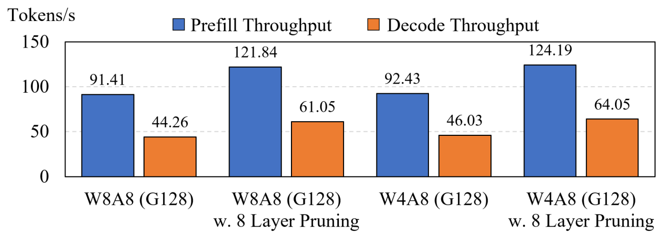

Fig. 4. Throughput performance comparison between quantized models with and without layer pruning

This approach highlights the importance of a balanced optimization strategy that does not solely focus on one aspect of performance. By carefully examining the interaction between model complexity, computational efficiency, and accuracy, we were able to devise a methodology that significantly enhances model performance while maintaining practical resource usage. The adoption of layer pruning, in particular, demonstrates that the potential for targeted adjustments to model architecture can lead to substantial improvements in efficiency without negatively affecting model accuracy. Considering tradeoffs among these factors is crucial for the development of high-performance computing applications where resource constraints are a significant consideration.

REFERENCES

[1] Peter Clark et al. Think you have solved question answering? try arc, the ai2 reasoning challenge. *ArXiv*, abs/1803.05457, 2018.
[2] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence?, 2019.
[3] Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. Piqa: Reasoning about physical commonsense in natural language, 2019.
[4] Wei Zhang et al. TernaryBERT: Distillation-aware ultra-low bit BERT. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *EMNLP*, Online, 2020. Association for Computational Linguistics.
[5] Chaofan Tao et al. Compression of generative pre-trained language models via quantization. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, *ACL*. Association for Computational Linguistics, May 2022.
[6] Yuhui Xu et al. Qa-lora: Quantization-aware low-rank adaptation of large language models, 2023.
[7] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for llm compression and acceleration, 2023.
[8] Minsoo Kim et al. Token-scaled logit distillation for ternary weight generative language models. *Advances in Neural Information Processing Systems*, 36, 2024.
[9] G. Gerganov. llama.cpp: Low-latency llm inference library for pure c/c++., 2024.
[10] Jiwon Song, Kyungseok Oh, Taesu Kim, Hyungjun Kim, Yulhwa Kim, and Jae-Joon Kim. Sleb: Streamlining llms through redundancy verification and elimination of transformer blocks, 2024.
[11] Xin Men, Mingyu Xu, Qingyu Zhang, Bingning Wang, Hongyu Lin, Yaojie Lu, Xianpei Han, and Weipeng Chen. Shortgpt: Layers in large language models are more redundant than you expect, 2024.