An Accurate and Efficient Framework to Deploy Qwen2.5 Model on ARM CPU Yitian 710

Team: MyGO!!!!!

Members: Ruchao Fan, Wenyuan Zhao, Zhao Cang, Xiang Ye, Yuhang Gu

Advisers: Yongming Tang, Hao Sun

Affiliations: Southeast University

**Abstract**: With the rapid advancement of AI technology, large language models (LLMs) have become indispensable tools in our daily lives. The proliferation of large language models (LLMs) in edge computing scenarios has intensified the demand for deploying billion-scale parameter models on ARM-based CPU architectures. In this work, we successfully deployed the Qwen2.5-0.5B model on the Yitian 710 server , employing a combination of multi-round fine-tuning, 4-bit quantization, and inference optimization techniques. These approaches significantly enhance the model's throughput and reduce memory footprint without compromising accuracy. Compared to the baseline, we achieve a 90% reduction in memory usage, with the prefill and decode throughput increasing by 373% and 1200%. This work secured the second-place ranking in the AICAS 2025 Grand Challenge.

**Keywords**: Qwen, Multi-round Fine-tuning, Quantization, Inference Optimization.

# **Introduction**

This work presents a systematic optimization methodology for deploying the Qwen2.5-0.5B[1] model on Alibaba's Yitian 710 server, which features 128 ARMv9 cores and a top memory bandwidth of 281GB/s. Our solution addresses three critical challenges in deploying LLMs on ARM architectures:

1. Degradation of large model accuracy after quantization,
2. Memory efficiency under constrained cache,
3. Inference optimization through compute-memory trade-offs.

Existing approaches typically focus on isolated optimization dimensions, either through post-training quantization (e.g., GPTQ[2], AWQ[3]) or hardware-specific operator tuning. However, our experiments reveal that the direct application of these methods leads to suboptimal performance on Yitian 710. To bridge this gap, we propose a co-design framework integrating three novel components:

1. **End-to-end pipeline optimization for Qwen2.5-0.5B deployment**: We establish a cohesive deployment framework that integrates model adaptation using LLaMA-Factory[4], hardware-aware quantization, and ARM-optimized compilation through llama.cpp.
2. **Multi-round fine-tuning for quantization robustness**: To mitigate accuracy degradation caused by quantization, we implement a data-aware fine-tuning strategy using Low-Rank Adaptation (LoRA)[5]. Through multi-round fine-tuning on non-target related domain datasets, we improved the model's generalization, achieving an average accuracy improvement of 5.76% on ARC-Challenge, Hellaswag[6], and C-eval[7] in FP16 format.
3. **Inference optimization for Yitian 710 cache architecture**: We propose the following optimizations:
4. Matrix Tiling: Adjusting block sizes through matrix tiling to reduce memory consumption and enhance the L1/L2 cache hit rate.
5. Instruction Set Acceleration: Utilizing instruction sets such as NEON/SVE2 to accelerate matrix operations.
6. Compile optimization: Incorporating the OpenBLAS inference engine and exploring various compilation options to further improve inference performance.

On the three datasets ARC-Challenge, Hellaswag, and C-Eval specified by the competition organizers, our optimized model achieves an accuracy of 50.75%, surpassing the full-precision (FP) model's accuracy of 50.07%. Compared to the baseline, we achieve a 90% reduction in memory usage, with the prefill and decode throughput increasing by 373% and 1200%, while ensuring accuracy. This demonstrates our method's unique advantage in balancing accuracy preservation, memory efficiency, and throughput.

# **Methods**

1. ***Overview***

In this section, we will detail the optimization methods of Qwen2.5-0.5B model. Our optimization framework is shown in **Fig.1**.

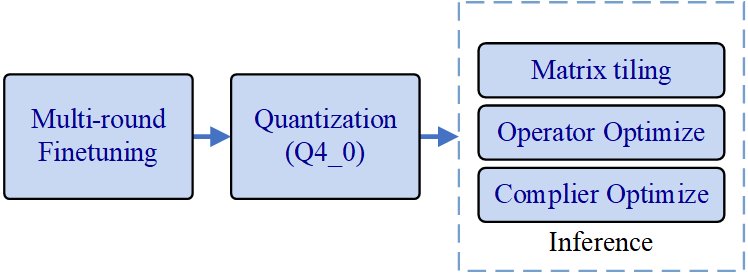


Figure 1: The optimization framework of Qwen deployment

As shown in **Fig.1**, our optimization process can be divided into three parts: multi-round fine-tuning, quantization, and inference optimization. This chapter will elaborate on the above three aspects. Fine-tuning is used to improve the model's accuracy on target datasets, while quantization can achieve model lightweighting. Inference optimization enhances the model's throughput and reduces memory usage, including matrix tiling, operator optimization, and compiler optimization.

1. ***Multi-round Fine-tuning***

We fine-tune the FP16 model to compensate for the accuracy loss caused by quantization. Fine-tuning is a process that adapts a pre-trained model to a specific task by further training it on the smaller, task-specific datasets. This approach leverages the generalized knowledge of the pre-trained model while specializing it for improved performance on the target task. In this work, we employed LoRA (Low-Rank Adaptation), a parameter-efficient fine-tuning algorithm that injects trainable low-rank matrices into the model's layers, reducing computational costs and memory usage during model training. Using LLaMA-Factory, we can fine-tune the model in safetensor format easily.

Table 1: Datasets under test in final round

|  |  |
| --- | --- |
| Datasets Name | Details |
| ARC- Challange | Choice question of science question. |
| Hellaswag | Choice question of continued writing. |
| C-Eval | Chinese choice questions in multi-fields. |

Table 2: Relevant datasets

|  |  |
| --- | --- |
| Datasets Name | Details |
| Alpaca GPT[8] | Logical inference and translation QA, including English and Chinese. |
| Ultra Chat | Multi-round dialogue, context understanding. |
| WinoGrande[9] | Choice question of continued writing. |
| SWAG[10] | Choice question of continued writing. |
| Story Cloze[11] | Choice question of story continued writing. |
| Commonsense QA[12] | Choice question of common sense. |
| CMMLU[13] | Chinese choice questions in multi-fields. |

Different from the competition last year, this competition does not allow the use of the datasets under test for fine-tuning. Therefore, we found 7 kinds of relevant datasets. The under-testing and relevant datasets are shown in **Table 1** and **Table 2**.

In order to maximize the accuracy of the FP model, we employed a multi-round fine-tuning approach. Our fine-tuning process is shown in **Table 3**.

First, we fine-tune the original model Qwen2.5-0.5B on all of 7 datasets to obtain model *LoRA1*. Compared to the original model, *LoRA1* achieves a 0.72% increase in accuracy.

Secondly, we fine-tune the original model on datasets *Alpaca GPT* and *Ultra Chat* to obtain model *LoRA2*. This step aims to enhance the model's logical reasoning and context understanding capabilities, achieving an almost 1% increase in accuracy.

Thirdly, we fine-tune the model *LoRA2* using the remaining five datasets to obtain *LoRA2.1*. This step aims to improve the model's ability to solve multiple-choice questions in various fields, achieving a 1.61% increase in accuracy.

Table 3: Fine-tune process

|  |  |  |  |
| --- | --- | --- | --- |
| Model name | Fine-tuning base model | Fine-tuning datasets | Average accuracy |
| Qwen2.5-0.5B (origin) | / | / | 0.5007 |
| LoRA1 (1-round) | Qwen2.5-0.5B | Alpaca GPT Ultra Chat SWAG WinoGrande Story Cloze CommonsenseQA CMMLU | 0.5079 |
| LoRA2 (1-round) | Qwen2.5-0.5B | Alpaca GPT Ultra Chat | 0.5100 |
| LoRA2.1  (2-round) | LoRA2 | SWAG WinoGrande Story Cloze CommonsenseQA CMMLU | **0.5168** |
| LoRA2.2 (2-round) | LoRA2 | SWAG WinoGrande Story Cloze | 0.5114 |
| LoRA2.2.1 (3-round) | LoRA2.2 | CommonsenseQA CMMLU | 0.5146 |

Finally, to explore whether more rounds of fine-tuning would yield better results, we performed two rounds of fine-tuning on the *LoRA2* model. The first round used datasets *SWAG*, *WinoGrande,* and *Story Cloze* to obtain model *LoRA2.2*. The second round used datasets *CommonsenseQA* and *CMMLU* to obtain the model *LoRA2.2.1*. However, the average accuracy of this method was 0.22% lower than that of *LoRA2.1*.

In addition, **Fig.2** and **Fig.3** show the loss curves of models *LoRA1* and *LoRA2.1*, respectively. It can be observed that the loss curve of the former exhibits significant fluctuations and fails to converge, whereas the latter's loss curve is smoother and converges more rapidly. This also indirectly demonstrates the superiority of multi-round fine-tuning.

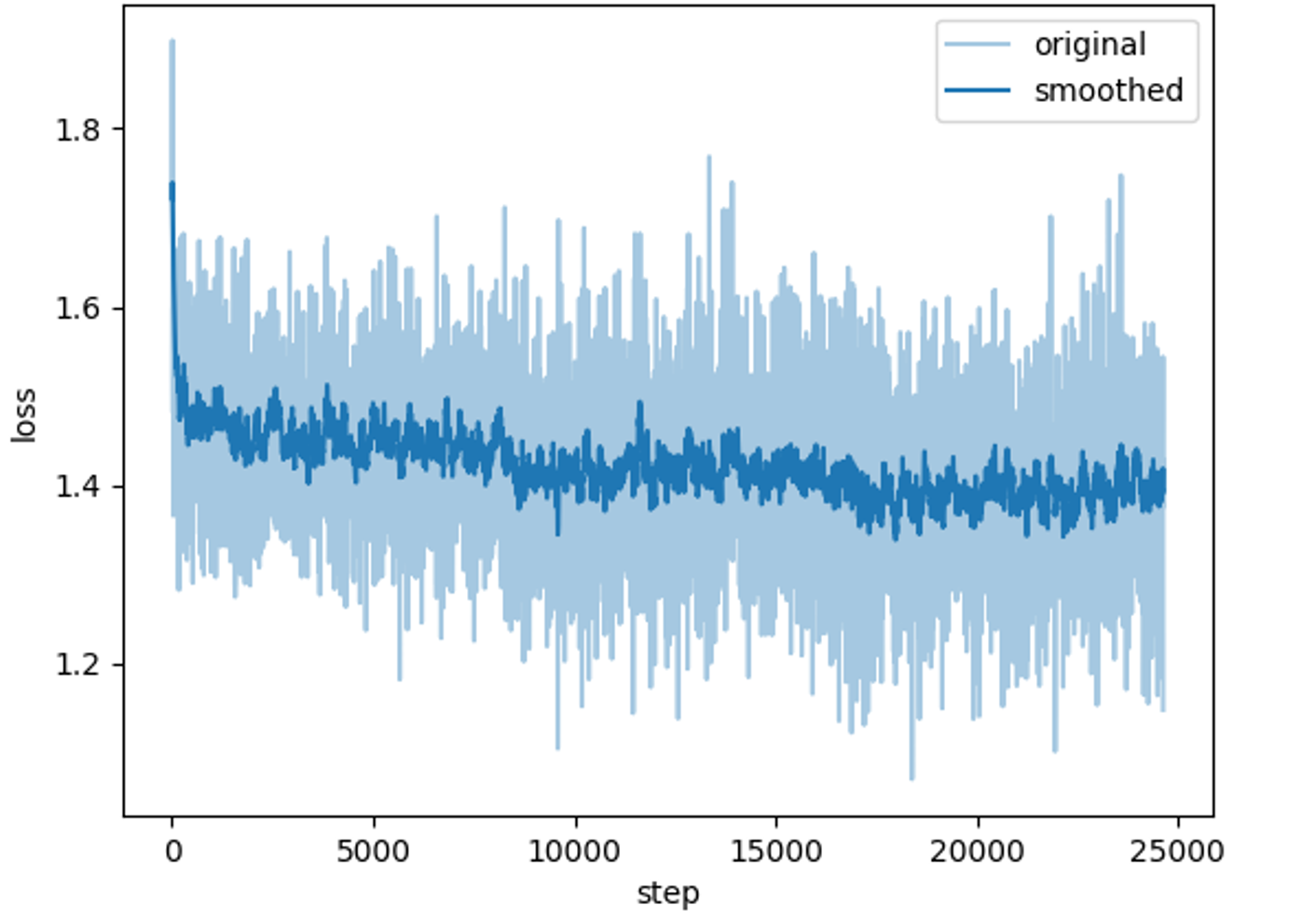


Figure 2: The training loss of *LoRA1* (1-round)

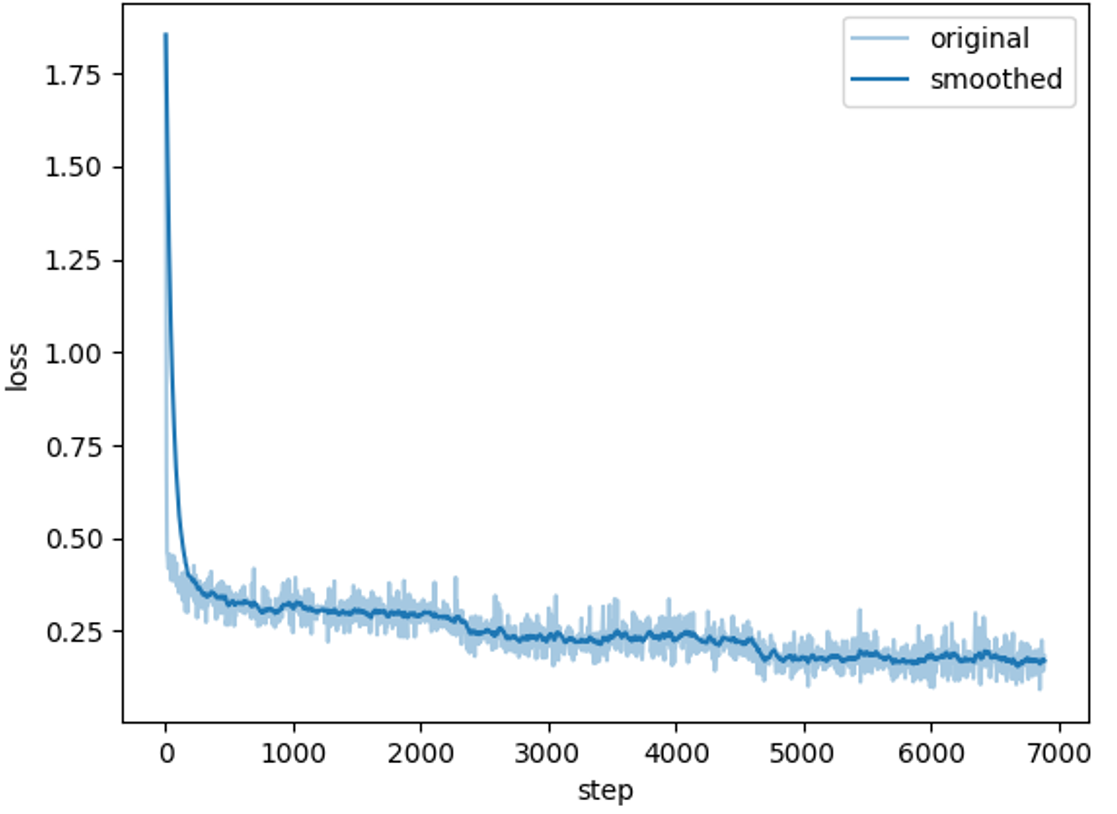


Figure 3: The training loss of *LoRA2.1* (2-round)

We further experiment with QLoRA (Quantized Low-Rank Adaptation)[14], an advanced variant of LoRA that incorporates quantization to reduce memory usage and computational demands. QLoRA applies low-precision quantization (e.g., 4-bit or 8-bit) to the model weights while maintaining trainable low-rank adapters, enabling efficient fine-tuning of large models. We choose the 4-bit QLoRA algorithm, which can reduce the precision loss caused by quantization.

In summary, among the fine-tuning results shown in **Table 3**, the two-round fine-tuning model *LoRA2.1* shows higher accuracy than 1-round and 3-round models, achieving 51.68%. This result demonstrates the potential of multi-round fine-tuning methods in enhancing model performance for specific applications.

1. ***Quantization***

Quantization is a fundamental technique in neural network optimization, reducing the precision of weights and activations to decrease memory footprint and computational demands while maintaining acceptable model accuracy. This approach is particularly critical for deploying large-scale models on resource-constrained devices, such as mobile and embedded systems, where efficiency is paramount. Among various quantization strategies, post-training quantization (PTQ) is widely adopted due to its simplicity, as it compresses pre-trained models without requiring extensive retraining.

1. ***Quantization Methodology***: In this study, we quantized the Qwen2.5-0.5B model, originally stored in the safetensors format, by first converting it to GGUF (Generic Gradient Update Format), a highly optimized storage format designed for efficient inference. We then applied 4-bit quantization (Q4\_0), a method that strikes a balance between compression ratio and computational efficiency. Notably, we modified the original llama.cpp implementation, which employed Q6\_K quantization for the output.weight tensor to ensure uniform Q4\_0 quantization across all tensors. This adjustment was motivated by the need to optimize for weight rearrangement (reordering), a technique that enhances inference speed by improving memory access patterns on modern CPU architectures, particularly those supporting SIMD (Single Instruction, Multiple Data) instructions such as ARM NEON and Intel AVX2.
2. ***Weight Rearrangement and Q4\_0 Optimization***: Weight rearrangement is a critical optimization in quantized models, particularly when leveraging vectorized operations in low-precision arithmetic. The Q4\_0 format was chosen over higher-precision alternatives (e.g., Q4\_K\_M) due to its alignment with efficient SIMD execution. Specifically:

* Q4\_0 uses 4-bit integers with a shared scale factor, allowing for efficient packing into 128-bit or 256-bit SIMD registers.
* The fixed block size (typically 32 or 64 elements per quantized block) ensures that memory accesses remain cache-friendly, reducing latency during matrix multiplication.
* By contrast, Q4\_K\_M, while offering higher precision due to per-block scaling and zero-point adjustments, introduces additional computational overhead, leading to reduced memory throughput and slower inference speeds in our benchmarks.

1. ***Comparative Analysis: Q4\_0 vs. Q4\_K\_M***: To evaluate the trade-offs between different quantization methods, we conduct experiments comparing Q4\_0 and Q4\_K\_M on the same hardware, which is shown in **Table 4**.

Table 4: Model Efficiency Analysis Across Different Quantization Levels

|  |  |  |  |
| --- | --- | --- | --- |
| Quantization bits | Average accuracy | Prefill + Decode throughput (Tokens/s) | Rss + Vms  (MB) |
| Q4\_0 | 0.5075 | 969.63+124.06 | 403.03+  658.33 |
| Q4\_K\_M | 0.5090 | 215.84+87.37 | 445.53+  701.75 |

Our findings revealed:

* **Memory Efficiency**: In terms of memory footprint, Q4\_0 demonstrated clear advantages, with a maximum resident set size (max\_rss) of only 403.03 MB, 9.5% lower than Q4\_K\_M's 445.53 MB. Virtual memory usage (max\_vms) also favored Q4\_0 (658.33 MB vs. 701.75 MB), attributable to its simpler quantization structure. This improvement in memory efficiency is particularly critical for edge devices with limited DRAM bandwidth.
* **Inference Throughput Comparison**: Prefill phase: Q4\_0 achieved a throughput of 969.63 tokens/s, significantly outperforming Q4\_K\_M's 215.84 tokens/s. Decoding phase:Q4\_0 maintained a performance of 124.06 tokens/s, 42% faster than Q4\_K\_M's 87.37 tokens/s. This performance gap primarily stems from Q4\_0's more efficient SIMD instruction utilization and streamlined computation flow.
* **Accuracy Impact Assessment**: In terms of model accuracy, Q4\_0 scored 0.7792, while Q4\_K\_M achieved 0.8611. Although Q4\_K\_M showed an ~10.5% absolute accuracy advantage, Q4\_0's marginal precision loss remains acceptable for most practical applications. Considering its superior memory efficiency and computational speed, Q4\_0 proves to be a more practical choice for resource-constrained environments.  
  These experimental results clearly indicate that the Q4\_0 quantization scheme delivers better system-level performance while maintaining acceptable accuracy. Particularly in edge computing scenarios, its advantages in memory efficiency and processing speed make it the preferred deployment option.

1. ***Inference***

In this task, we employ llama.cpp for backend inference optimization of Qwen2.5. Llama.cpp is a highly efficient C++ implementation, significantly enhancing the performance of LLMs by leveraging optimized computational algorithms and memory management. Its lightweight design ensures minimal resource consumption, making it ideal for deployment in resource-constrained environments. Additionally, llama.cpp supports seamless integration with various hardware accelerators, further boosting inference speed. By utilizing llama.cpp, we achieve a substantial reduction in latency and computational overhead, thereby improving the overall efficiency and scalability of the model in real-world applications. We optimize the backend in three aspects:

1. ***Matrix tiling***: Backend inference can be divided into two parts: prefill and decode. In the prefill stage, the dominant operation is batched matrix multiplication (e.g., QKT). To optimize memory efficiency and throughput, we employ matrix tiling, a technique that decomposes large matrices into smaller tiles processed sequentially. This technique aligns tile sizes with cache capacities, reducing global memory accesses and minimizing cache misses. By partitioning matrices into tiles, each tile is processed by dedicated thread cores, enabling concurrent execution and data locality.

Table 5: Throughput in prefill and decode stage across different chunk sizes

|  |  |  |
| --- | --- | --- |
| chunk\_size | prefill\_  throughput | decode\_  throughput |
| 8\*8 | 924.67 | 133.77 |
| 16\*16 | 974.90 | 141.53 |
| 32\*32 | 987.25 | 152.97 |
| 64\*64 | 952.79 | 141.55 |
| 128\*128 | 924.45 | 141.21 |

As illustrated in **Table 5**, chunk\_size represents the edge length of matrix blocks (e.g., 8\*8 denotes a block of 8 rows and 8 columns), used to partition data for computational efficiency. By strategically adjusting the matrix tiling parameters, we significantly enhanced the model's throughput, achieving a more than threefold improvement. Matrix tiling plays a pivotal role in this optimization by dividing large matrices into smaller, manageable blocks that fit more efficiently into the cache memory. This reduces memory bandwidth requirements and minimizes cache misses, leading to faster data access and computation.

1. ***Operator Optimization***: To maximize decode-stage efficiency on ARM processors, we implemented targeted optimizations using Scalable Vector Extensions (SVE) and NEON SIMD instructions. By leveraging SVE's variable-length vector processing (128–2048 bits) and NEON's fixed-width parallelism, we accelerated matrix-vector operations in autoregressive token generation. Key techniques included SVE-predicated loops for dynamic sequence handling and NEON-optimized fused multiply-accumulate for attention computations. These changes improved throughput to 152.97 tokens/second – a 1.7× gain over the baseline – while reducing latency through register tiling and vectorized post-layer norm operations.
2. ***Compiler Optimization:*** To maximize hardware utilization during compilation, we configured the build system with architecture-specific optimizations. The compilation pipeline employed: *cmake -B build -DCMAKE\_CXX\_FLAGS="-O3 -march=native" cmake --build build --config Release.*

*-march=native*: Automatically detects and enables all instruction sets (e.g., NEON, SVE on ARMv8/v9) supported by the host CPU, ensuring optimal vectorization without manual flags.

*-O3*: Aggressive compiler optimizations, including loop unrolling, inlining, and SIMD parallelization

*Release Mode*: Strips debug symbols, enables link-time optimization (LTO), and prioritizes speed over safety checks.

We also incorporated OpenBLAS for performance gains, but the improvement was marginal due to thread count limitations and llama.cpp's pre-existing optimization of most BLAS operations.

Through targeted compiler optimizations, we achieved significant improvements in model efficiency. By leveraging -march=native and -O3 optimizations, we observed a 10% throughput improvement alongside 20% lower memory consumption, attributable to enhanced vectorization and reduced memory stall cycles. This approach proves particularly effective for ARM-based edge deployments, where hardware heterogeneity demands adaptive optimization strategies.

# **Results**

In this study, we utilized Alibaba Cloud's Yitian 710 server as the primary testing platform, which is equipped with 128 ARMv9 cores and 32GB of memory.

We evaluate our optimization framework from two aspects: model ability and inference efficiency. Model ability refers to the accuracy on *arc\_challenge, hellaswag, and C\_eval* datasets, while inference efficiency refers to the memory usage and throughput, including *max\_rss*, *max\_vms*, *prefill\_throughput,* and *decode\_throughput*.

1. ***Model Ability***

We tabulate the accuracy of Qwen2.5-0.5B model at different stages, as shown in **Table 6**. The fine-tuned model achieves a 0.2 % accuracy improvement. Based on the fine-tuned FP (Full Precision) model, the accuracy of Q4\_0 quantization achieves a 0.05% improvement compared with the original model, which ensures the model ability of the quantized model.

The accuracy evaluation in this competition adopts a weighted average. The weights of datasets ARC-Challenge, Hellaswag, and C-Eval are 0.1, 0.8, and 0.1, respectively. Our optimized model achieves an accuracy of 50.75%, surpassing the full-precision (FP) model's accuracy of 50.07%.

Table 6: Accuracy of Qwen2.5-0.5B

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Model | ARC-Challenge | Hellaswag | C-Eval | average |
| Original | 0.3242 | 0.5214 | 0.5119 | 0.5007 |
| Fine-tuning | 0.3489 | 0.5336 | 0.5501 | 0.5168 |
| Fine-tuning+Q4\_0 | 0.3430 | 0.5238 | 0.5423 | 0.5075 |

1. ***Inference Efficiency***

As shown in **Table 7**, we tabulate the four efficiency metrics of our optimization method, which respectively show 90%, 90%, 373%, and 1200% improvement.

Table 7: Inference efficiency of our framework

|  |  |  |  |
| --- | --- | --- | --- |
| Metrics | Original | Optimized | Improve |
| max\_rss (MB) | 4112.06 | 403.03 | 90% |
| max\_vms (MB) | 6741.13 | 658.33 | 90% |
| prefill\_th (token/s) | 264.74 | 987.25 | 373% |
| decode\_th (token/s) | 12.73 | 152.97 | 1200% |

# **Conclusion**

In summary, we employed a comprehensive approach encompassing multi-round fine-tuning, quantization, and inference optimization to achieve efficient deployment of Qwen2.5-0.5B model on ARM CPUs. By integrating these techniques, we successfully optimized the Qwen model for efficient inference without compromising accuracy. As a result, we significantly improved the model's performance in terms of memory usage and throughput during inference. This achievement demonstrates the effectiveness of our multi-faceted optimization strategy in enabling the high-performance deployment of LLMs on resource-constrained hardware.

**References**

1. Yang A, Yang B, Zhang B, et al. Qwen2. 5 technical report[J]. arXiv preprint arXiv:2412.15115, 2024.
2. Frantar E, Ashkboos S, Hoefler T, et al. Gptq: Accurate post-training quantization for generative pre-trained transformers[J]. arXiv preprint arXiv:2210.17323, 2022.
3. Lin J, Tang J, Tang H, et al. Awq: Activation-aware weight quantization for on-device llm compression and acceleration[J]. Proceedings of Machine Learning and Systems, 2024, 6: 87-100.
4. Zheng Y, Zhang R, Zhang J, et al. Llamafactory: Unified efficient fine-tuning of 100+ language models[J]. arXiv preprint arXiv:2403.13372, 2024.
5. Hu E J, Shen Y, Wallis P, et al. Lora: Low-rank adaptation of large language models[J]. ICLR, 2022, 1(2): 3.
6. Zellers R, Holtzman A, Bisk Y, et al. Hellaswag: Can a machine really finish your sentence?[J]. arXiv preprint arXiv:1905.07830, 2019.
7. Huang Y, Bai Y, Zhu Z, et al. C-eval: A multi-level multi-discipline chinese evaluation suite for foundation models[J]. Advances in Neural Information Processing Systems, 2023, 36: 62991-63010.
8. Peng B, Li C, He P, et al. Instruction tuning with gpt-4[J]. arXiv preprint arXiv:2304.03277, 2023.
9. Sakaguchi K, Bras R L, Bhagavatula C, et al. Winogrande: An adversarial winograd schema challenge at scale[J]. Communications of the ACM, 2021, 64(9): 99-106.
10. Zellers R, Bisk Y, Schwartz R, et al. Swag: A large-scale adversarial dataset for grounded commonsense inference[J]. arXiv preprint arXiv:1808.05326, 2018.
11. Mostafazadeh N, Roth M, Louis A, et al. Lsdsem 2017 shared task: The story cloze test[C]//2nd Workshop on Linking Models of Lexical, Sentential and Discourse-level Semantics. Association for Computational Linguistics, 2017: 46-51.
12. Talmor A, Herzig J, Lourie N, et al. Commonsenseqa: A question answering challenge targeting commonsense knowledge[J]. arXiv preprint arXiv:1811.00937, 2018.
13. Li H, Zhang Y, Koto F, et al. Cmmlu: Measuring massive multitask language understanding in chinese[J]. arXiv preprint arXiv:2306.09212, 2023.
14. Dettmers T, Pagnoni A, Holtzman A, et al. Qlora: Efficient fine-tuning of quantized llms[J]. Advances in neural information processing systems, 2023, 36: 10088-10115.