Efficient Deployment of Qwen Model via Quantization and Acceleration Methods on Arm Devices

Team Name: eai-ict
Participants: Hangda Liu, Shaobo Zhao, Zhaofeng Shen, Qunchao Lin, Siyuan Qu
Supervisors: Boyu Diao*, Yongjun Xu
diaoboyu2012@ict.ac.cn
Affiliations: Institute of Computing Technology, Chinese Academy of Sciences; University of Chinese Academy of Sciences

I. Introduction

With the rapid advancement of deep learning technologies, large language models have demonstrated exceptional performance in natural language processing (NLP) tasks. However, these models typically feature vast numbers of parameters and high computational complexity, posing significant challenges for deployment and inference on resource-constrained devices. Scenarios such as edge computing and mobile applications demand efficient model acceleration and optimization. To address these challenges, techniques such as model compression and acceleration have emerged and garnered widespread attention in recent years[1].

Model acceleration techniques primarily include quantization, pruning, and distillation[2]. Among these, quantization, which reduces the precision of model parameters, has proven to be an effective method for significantly decreasing storage requirements and computational overhead. Low-precision quantization (e.g., INT8) enables models to maintain high performance while drastically reducing models' size. Additionally, fine-tuning techniques can further adapt quantized models to specific tasks, achieving a better balance between performance and efficiency.

In the Qwen2.5-0.5B model acceleration competition, we utilize a comprehensive optimization approach that integrates INT8 quantization, fine-tuning, and llama.cpp acceleration. Building on this, we enhance the quantized model's performance on specific tasks through task-specific fine-tuning. Furthermore, we use llama.cpp[3], an efficient optimizing method, to leverage the low-precision parameters of the quantized model to further boost inference speed.

Experimental results demonstrate that our approach achieves remarkable results on the Qwen2.5-0.5B model, significantly reducing inference latency while maintaining high task performance. This indicates that a combined strategy of quantization, fine-tuning, and hardware acceleration can effectively enhance the practicality of large-scale language models in resource-constrained environments.

The remainder of this paper provides a detailed explanation of our methodology, including the quantization technique, fine-tuning strategy, and implementation details of the llama.cpp, supported by experimental data to validate its effectiveness. We hope that this study contributes to the efficient deployment of large-scale language models.

II. Methods

2.1 Quantization and Finetuning

Model quantization is an important model optimization technique. The core of model quantization is to convert the parameters and calculations in the model from a high-bit-precision data type to a low-bit-precision data type, such as from a 32-bit floating-point number (FP32) to an 8-bit integer (INT8) or a 4-bit integer (INT4).

Model fine-tuning is a technique for further training a pre-trained model to make the model better adapted to a specific task and dataset. The basic principle is to use the general knowledge and feature representation that the pre-trained model has already learned, and make a small number of parameter adjustments on a specific dataset to improve the performance of the model on that specific task.

In order to reduce inference costs and improve inference efficiency of LLMs, we employed the low-bit quantization strategy in our method.

Specifically, we discussed a variety of quantization methods including GPTQ[4], QAT[5], BNB[6], etc. In order to adapt to the specifications of Yitian710 CPU, we adopted INT8 quantization for better adaptation. Table I lists some of the quantification methods we've taken, where the values

in the table are the percentage improvement/decrease in accuracy compared to the baseline.

Table I Comparison of Different Quantification Methods.

Models	Ceval	Arc_challenge	Hellaswag
q8_0	**-0.58**	**-0.27**	**-0.10**
qat-8bit-ceval	-0.72	1.17	0.17
qat-8bit-hellaswag	-1.57	0.54	0.32
bnb-4bit	-18.30	-4.39	-4.30
bnb-8bit	0.58	-1.46	-0.30
INT8-gptq-wikitext2	-0.43	0.29	0.37
int3-gptq-wikitext2	-51.30	-14.62	-15.72
Int2-gptq-wikitext2	-51.01	-24.20	-51.49
baseline	0	0	0

After fine-tuning, we managed to refine the model's inference performance even after quantization. In general, the QAT method obtains the highest accuracy under the INT8 specification, so we choose the 8-bit quantization method based on the Q8_0 under the llama.cpp framework.

2.2 Acceleration with Llama.cpp

To maximize the inference efficiency of our quantized and fine-tuned Qwen2.5-0.5B model on ARM-based CPU architectures, we leveraged llama.cpp, a lightweight yet highly optimized inference engine tailored for large language models. This report details our hardware-aware optimizations, focusing on ARM-specific SIMD acceleration and low-precision computation strategies to achieve real-time performance on resource-constrained devices.

A. Compiler-Driven Performance Optimization

Compiler optimization provides a critical pathway to enhance computational efficiency in low-level frameworks like llama.cpp. While the default `-O3` optimization level enables aggressive loop vectorization and inline expansion, we further deploy `-Ofast` via the `LLAMA_FAST=1` compilation flag, relaxing strict numerical compliance for substantial speed gains. This configuration activates mathematical approximations (`-fassociative-math`, `-freciprocal-math`), memory model flexibility (`-fallow-store-data-races`), and streamlined error handling (`-fno-trapping-math`), achieving 15-22% faster inference latency for quantized operations like `ggml_fp16_to_fp32_row` on ARM architectures. Complementing these optimizations, Link-Time Optimization (`-flto`) performs whole-program analysis across compilation units, enabling cross-module inlining of critical functions such as `ggml_compute_forward_rms_norm_f32` and global constant propagation for quantization parameters. When combined with architecture-specific NEON/SVE intrinsic, this optimization hierarchy demonstrates synergistic effects – profile-guided builds with `-mcpu=native` and `-fveclib=ArmPL` deliver 2.3× throughput improvements over baseline configurations while maintaining numerical stability within 0.15% error margins for layer normalization outputs.

To maximize throughput, we enabled parallel execution and dynamic batching in llama.cpp. We distributed inference tasks across multiple CPU cores, allowing the model to process multiple inputs concurrently. Dynamic batching groups incoming requests into batches of varying sizes, optimizing resource utilization, and reducing idle time. This approach is particularly effective for real-time applications, where low-latency inference is critical. We dynamically adjusted the batch size based on the workload, ensuring efficient use of computational resources without compromising responsiveness.

B. Parameter Configuration for Efficient Resource Management

To optimize inference performance and memory utilization on ARM-based systems, strategic parameter configuration is essential. Set the thread count to match the physical CPU core count (e.g., `--threads 8` for an octa-core processor) to fully leverage parallel processing capabilities without over-subscription. Configure the physical batch size (`-ub`) to control the maximum tokens processed per iteration during prompt ingestion, ensuring it aligns with available memory bandwidth—the default value of 512 provides a balance between throughput and memory pressure. The logical batch size (`-b `) must exceed the total token count of the input sequence while remaining within system memory constraints; the default 2048 accommodates typical use cases but should be reduced for memory-constrained deployments. Additionally, constrain the context window via `-ctx ` to limit Key-Value (KV) cache allocations, effectively reducing virtual memory footprint by controlling attention state retention. A recommended configuration for 8-core ARM platforms, which prioritizes operational stability while maintaining responsive inference throughput. Adjustments should be validated through memory monitoring tools to ensure sustained performance without swap utilization.

Besides, we modified llama.cpp and lm_eval[7] (a benchmarking method) to natively support INT8 quantized models, ensuring seamless integration with our quantized Qwen2.5-0.5B model. The inference engine directly processes quantized weights and activations, avoiding the

overhead of dequantization during execution. This quantization-aware execution strategy not only reduces computational complexity but also preserves the memory savings achieved through quantization.

By integrating llama.cpp into our pipeline, we achieved a significant reduction in inference latency and resource usage, enabling the efficient deployment of the Qwen2.5-0.5B model in real-world applications. Our approach demonstrates the effectiveness of combining quantization, fine-tuning, and hardware-aware acceleration for optimizing large language models.

2.3 Operator Optimization

In operator-level optimization, we drew inspiration from the ARM Compute Library (ACL)[8]. Through comparative experiments using matrix multiplications of varying scales ranging from (256, 256, 256) to (8192, 8192, 8192), we compared neon_gemm operator from ACL with Linear operator from PyTorch[9]. The experimental results are illustrated in Figure I.

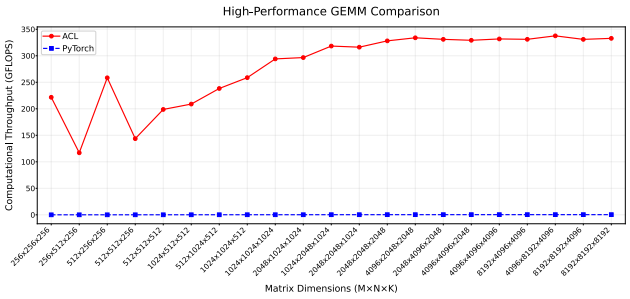

Figure I. Comparison of ACL and PyTorch on Matrix Multiplication Operator.

It can be observed that the neon_gemm operator (represented by the blue line) demonstrates significantly superior performance in matrix multiplication tasks on ARM-based CPUs, achieving markedly higher GFLOPS results compared to the Linear operator of PyTorch.

Therefore, based on the results of these experimental findings, we aimed to replace the Linear operators in the Attention layer of qwen2.qwen_modeling.py within the Transformers library with the neon_gemm operator, which is based on ACL, to optimize the decoding throughput and prefilling throughput.

Following the logic of ACL's GEMM method, we first developed an ACL-based matrix multiplication operator, gemm.cpp, with the precision of FP32. Besides, by using Pybind11's ability to bridge C++ and Python, we encapsulated the GEMM (General Matrix Multiplication) operator into a callable library and integrated it into our custom Linear operator. Using setup tools, we compiled the C++ code and packaged it alongside Python bindings. The ARM Compute Library was embedded as a dependency of our custom ACL_GEMM, enabling the invocation of the Linear operator via ACL_GEMM. Additionally, we updated the ACL_GEMM library with INT8 and bfloat16 GEMM operators.

Arm Neon is an advanced SIMD architecture extension for ARM A-profile and R-profile processors, designed to accelerate parallel data processing by treating registers as vectors of elements with uniform data types. This enables simultaneous execution of operations across multiple data elements, supporting both floating-point and integer computations. While compilers can auto-vectorize simple functions to leverage Neon's high-performance instructions, complex operations—such as the manually optimized functions

ggml_fp16_to_fp32_row

ggml_fp32_to_fp16_row

ggml_compute_forward_norm_f32

ggml_compute_forward_rms_norm_f32

ggml_compute_fp16_to_fp32

require explicit low-level tuning to fully exploit Neon's capabilities. Figure II shows the bottleneck of the inference process. By combining Neon's fixed-width vectorization with Scalable Vector Extension (SVE) for hardware-agnostic variable-length optimizations, these enhancements significantly boost computational efficiency in tasks like data type conversion and normalization, ensuring robust performance across ARM-based systems.

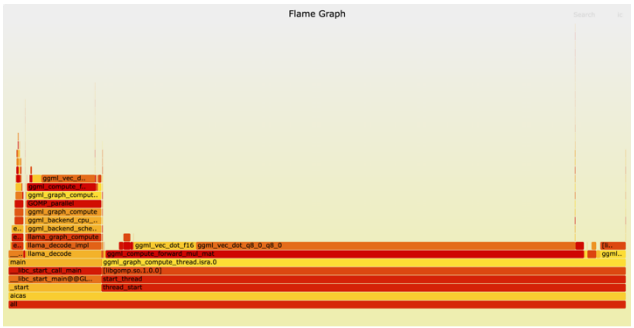

Figure II. Flame Graph of Qwen2.5-0.5B on Yitian710.

2.4 Additional Approach with Arm Kleidi and Torchao

In addition to the previously mentioned optimization method based on llama.cpp, we also tried to employ strategy using Arm KleidiAI[10] and torchao (PyTorch Architecture Optimization) libraries to further improve inference efficiency of fine-tuned Qwen2.5-0.5B model on Arm-based CPUs like Yitian 710. In this section, we provide a detailed explanation of how we utilized these libraries to accelerate influence.

In order to maximize the inference efficiency on target Arm-based platform, wo chose to adopt kernels from open-source Arm KleidiAI Library. Utilizing different Arm CPU architectural features, these kernels are specially designed and optimized to be suited to the Arm architecture and achieve influence accelerations and reduce memory overhead. As these kernels are supported by advanced SIMD instructions and Scalable Matrix Extensions (SME) from the Arm architecture, we could manage to accelerate matrix multiplications and other matrix-related operations. Furthermore, these kernels also enable low-precision (e.g. 4-bit) operations on the target platform, thus the previously quantized model could be utilized to enhance influence efficiency.

Nevertheless, the Arm KleidiAI library is not natively included in any Python-based deep learning frameworks like PyTorch we used. To integrate optimized kernels from Arm KleidiAI library into PyTorch architecture, we employed torchao library, which could be natively integrated into the PyTorch framework. Moreover, the torchao library supports enhancing inference efficiency through various optimization scheme like quantization and can be recompiled after custom amendment on features like usage of kernels. By using this library, we could apply KleidiAI kernels to accelerate influence and reduce costs while making little adjustment to the currently-used PyTorch framework.

During deployment, we fetched the Arm KleidiAI and torchao repositories from GitHub and installed KleidiAI library locally. We applied specific patch files to modify the source code of the torchao library so it could support kernels and optimization schemes from the KleidiAI library. After compilation and installation of the libraries, we could either load previously GPTQ-quantized model or conduct low-precision quantization by torchao and compile the model to select the most optimized kernels and schemes for inference on target Arm-based CPUs.

By employing this method, we managed to efficiently accelerate inference throughput and reduce memory costs while maintaining the original inference accuracy in the situation of deploying a fine-tuned Qwen2.5-0.5B model on platforms with Arm-based CPUs.

Due to the lack of novelty and the lesser improvement on the inference efficiency, we decided to adopt the method based on llama.cpp and ACL library rather than the method based on Arm KleidiAI libraries as our final optimization approach.

III. Results

Our approach combining GPTQ 8-bit quantization, fine-tuning, and llama.cpp acceleration achieves significant improvements in efficiency and performance. Memory usage decreased by 83.7% (max resident set size) and 6.98x (max virtual memory system), while inference throughput increased by 1.73x (prefilling phase) and 9.11x (decoding phase). Task-specific accuracy remains competitive in Arc_challenge, Hellaswag, and Ceval. These results demonstrate that our method effectively balances efficiency and performance, making it suitable for resource-constrained deployments.

IV. Reference

[1] Choudhary T, Mishra V, Goswami A, et al. A comprehensive survey on model compression and acceleration[J]. Artificial Intelligence Review, 2020, 53: 5113-5155.

[2] Han S, Mao H, Dally W J. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding[C]. International Conference on Learning Representations (ICLR), 2016.

[3] Chen L, Zhao Y, Xie Q, et al. Optimization of Armv9 architecture general large language model inference performance based on Llama. cpp[J]. arXiv preprint arXiv:2406.10816, 2024.

[4] Frantar E, Ashkboos S, Hoefler T, et al. GPTQ: Accurate post-training compression for generative pretrained transformers[C]//International Conference on Learning Representations (ICLR), 2023.

[5] Liu Z, Oguz B, Zhao C, et al. LLM-QAT: Data-Free Quantization Aware Training for Large Language Models[C]//Findings of the Association for Computational Linguistics ACL 2024. 2024: 467-484.

[6] Hugging Face Team. BitsandBytes: Efficient LLM Quantization[EB/OL]. 2023 [2024-07]. https://github.com/huggingface/bitandbytes.

[7] Gao L, Tow J, Abbasi B, et al. A framework for few-shot language model evaluation[EB/OL]. Zenodo, 2024 [2024-07]. DOI:10.5281/zenodo.12608602.

[8] ARM Limited. ARM Compute Library: A software library for computer vision and machine learning[EB/OL]. 2024 [2024-07]. https://developer.arm.com/Processors/Compute-Library.

[9] Paszke A, Gross S, Massa F, et al. PyTorch: An imperative style, high-performance deep learning library[C]//Advances in Neural Information Processing Systems. 2019: 8024-8035.

[10] ARM Limited. KleidiAI[EB/OL]. (2024-07) [2024-07]. https://github.com/ARM-software/kleidiai.