Qwen Inference Optimization: Leveraging Quantization and Weight Rearrangement

Jinwen Liang
Department of Computer Science and Technology
Nanjing University
Nanjing, China
211220092@smail.nju.edu.cn

Xingyu Luo
Department of Computer Science and Technology
Nanjing University
Nanjing, China
211830073@smail.nju.edu.cn

Lu Shi
Department of Computer Science and Technology
Nanjing University
Nanjing, China
211220049@smail.nju.edu.cn

Yuanze Sun
Department of Computer Science and Technology
Nanjing University
Nanjing, China
211840165@smail.nju.edu.cn

Jie Liu
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing , China
liujie@nju.edu.cn

***Abstract*—This report details our optimization strategies for large language models (LLMs), particularly for Alibaba Cloud's Qwen model. We adopt AWQ method to quantize the model and llama.cpp as the inference backend, which jointly reduce the model's size, improve inference efficiency, and lower memory occupation on CPUs. We organize our optimization scheme into three aspects: Quantization Strategy that balances accuracy and resource usage, Backend Optimization that includes detailed technical adjustment to llama.cpp, and Fine-Tuning that refines model performance for specific tasks. Together, these strategies achieve a balence between accuracy and resource usage, significantly lowering the threshold of practical deployment, making LLMs closer to daily life.**

***Index Terms*—large language model, qwen, llama.cpp, rearrangement**

I. Introduction

Large Language Models (LLMs) have achieved remarkable performance in a substantial number of downstream Natural Language Processing (NLP) tasks, such as text comprehension, text generation, sentiment analysis, machine translation, and interactive QA. However, **tens to hundreds billions of model parameters** present significant challenges for **efficient deployment of LLMs in edge applications**. The increase in model size far outpaces the improvements in hardware performance, urging both the academic and industrial sectors to explore methods of model compression, develop efficient operator, and optimize computation graph in order to deploy and run large models **under limited recourse conditions**. To efficiently employ LLMs, deeply squeezing the hardware performance, we must consider the features of both hardware and the model, which is a hardware-software co-optimization problem.

With the rapidly growing demand for deploying large models on the edge and considering the generality of computing resources, deploying large models on **CPUs** has become one of the future directions. The key issue for efficient deployment of large models on the edge is how to systematically optimize models to fit the limited hardware performance.

Qwen is a large language model officially released by Alibaba Cloud, with size ranging from billions to hundreds of billions parameters. Currently, Qwen shows comprehensive performance across mainstream benchmark evaluation datasets. Our solution, based on Qwen-1.8B/7B open-source model, adopts **AWQ quantization** method and uses **llama.cpp** as the inference backend. With systematic optimization, we significantly reduce the model size, greatly lower the inference latency on CPUs, substantially increase throughput, and markedly reduce inference memory occupation, while **maintaining the original accuracy** of the model. These improvements make the deployment of LLMs on limited hardware like CPUs more accessible.

In conclusion, Our contributions to optimizing large language models are summarized into three pivotal sections, each addressing a distinct aspect of enhancement:

1) **Quantization Strategy** This section details our approach of quantization, which significantly **reduces the model size and computational demands** while preserving accuracy. Here, we outline the techniques and rationale behind selected quantization schemes which

balance accuracy and resource utilization, making models adaptable to diverse hardware environments.

2) **Adjustment of llama.cpp Backend** We discuss the **technical enhancements made to the llama.cpp backend**, including the integration of advanced instruction sets and optimizations in tensor operations to fully leverage hardware capabilities. This section provides insights into the challenges we faced and the innovative solutions we implemented to enhance backend performance.
3) **Fine-Tuning Practices** This part focuses on our fine-tuning practices, which **refine the model's performance on targeted tasks**. It covers the adjustments to model parameters, innovative training approaches, and strategic selection of training datasets, aimed at maximizing the model's accuracy and effectiveness in practical applications.

In the second part of our report, we delve deeply into the Quantization Strategy, outlining the intricacies and rationale behind our approach. Following this, in the third part, we provide a detailed exposition of the optimizations we implemented within the llama.cpp backend, highlighting the technical enhancements and their impacts. The fourth part presents our experiences and outcomes from the Fine-Tuning Practices, sharing insights into how these adjustments have refined the model's performance. To conclude, our report wraps up with a comprehensive summary that reflects on the entirety of our project, underscoring our key findings and achievements.

II. Quantization

Large language models (LLMs) have shown excellent performance on various tasks, but the astronomical model size raises the hardware barrier for serving (memory size) and slows down token generation (memory bandwidth). Quantization **maps a floating-point number into lower-bit integers**, which is an effective method to **reduce the model size and inference costs** of LLMs.

Low-bit weight quantization for LLMs can save memory but is hard. Quantization-aware training (QAT) is not practical due to the **high training cost**, while post-training quantization (PTQ) suffers from large accuracy degradation under a low-bit setting. Works like GPTQ [3] uses second-order information to perform error compensation. It may over-fit the calibration set during reconstruction, **distorting the learned features** on out-of-distribution domains, which could be problematic since LLMs are generalist models.

We choose AWQ [5] (Activation-aware Weight Quantization) as our quantization strategy. AWQ is a hardware-friendly approach for LLM low-bit weight-only quantization. The method is based on the observation that **weights are not equally important**: protecting only 1% of salient weights can greatly reduce quantization error. It is proposed to search for the optimal perchannel scaling that protects the salient weights by observing the activation, not weights. AWQ does not rely on any backpropagation or reconstruction, so it can well preserve LLMs' generalization ability on different domains and modalities, without overfitting to the calibration set.

Our inference backend llama.cpp can only run gguf format models, and the low-bit safetensors format model is not convenient for format conversion. Therefore, instead of directly quantizing the model weights through AWQ into low-bit integers, we just apply the scales from AWQ to the model weights and still store them in floating point format to facilitate format conversion. During the scaling process, we only scale up the weights of the Attention Layer and MLP Layer in QwenBlock to reduce the accuracy loss caused by subsequent low-bit quantization of these weights. Meanwhile, we scale down the weights of RMSNorm Layer in front of the Attention layer and MLP layer to maintain the invariance of the computation. Since these weights account for a very small proportion in the model, they will not be quantized into low bits, but still be stored in **floating point format**. So the weight reduction will not cause losses in the quantization process.

In the preliminary round, we adopted Q4_K_M quantization for the floating-point model after applying AWQ quantization, which is a mix-bit type quantization method provided by llama.cpp. It applies 4-6 bit quantization to weights according to their importance. Compared with the fully 4-bit quantization method, the mix-bit quantization method can reduce the quantization loss and maintain the accuracy of the original model to a greater extent, but with relatively higher memory occupation and lower inference speed. In the semifinal round, we adopted **Q4_0 quantization**, a fully 4-bit quantization method provided by llama.cpp. It improves the inference speed and reduces the memory occupation at the expense of a certain accuracy. Additionally, it can be applied to the weight rearrangement technique we used in the semifinal round, which will be introduced in following sections.

III. llama.cpp

We utilize llama.cpp as the inference backend. Developed by Georgi Gerganov, Llama.cpp implements Meta's LLaMa architecture in **an efficient C/C++ manner and stands** as one of the most vibrant open-source communities focused on LLM inference. Designed as a CPU-prioritized C++ library, llama.cpp features lower complexity and can be seamlessly integrated into other programming environments, which enhances its wide compatibility and accelerates its application across various platforms. Llama.cpp is operable on numerous platforms, offering support for the AVX, AVX2, and AVX512 instruction sets on x86 architectures. Furthermore, llama.cpp supports 2-bit, 3-bit, 4-bit, 5-bit, 6-bit, and 8-bit integer quantization, which can **accelerate inference speed and reduce memory occupation**. In our scheme, we further quantize the AWQ-quantized model using Q4_0 quantization to adapt it to llama.cpp, further boosting the inference speed.

A. Preliminary round

In the preliminary round, a significant enhancement is made by transform the SIMD computation component within the original llama.cpp from NEON instruction set to **SVE**

(Scalable Vector Extensions) instruction set. This adaptation exploits the increased scalability and adaptability offered by SVE, which is especially advantageous for high-performance computing tasks on compatible platforms. SVE instruction set unleashes the potential performance of CPUs, boosting computational efficiency and throughput, allowing for better hardware capability utilization.

Besides, in order to build the evaluation scrips conveniently, we integrate 'llama-cpp-python' into 'optimum-benchmark' and 'lm-eval'. This **Python binding** of llama.cpp provides a user-friendly interface, making it simpler for developers to modify and optimize benchmark scripts. The use of llama-cpp-python facilitates the straightforward implementation and testing of scripts, ensuring the updates are finely tuned to take advantage of the revised llama.cpp backend. This method significantly improves the efficiency of performance evaluations in NLP tasks.

B. Semifinal round

During the performance optimization phase, several key enhancements were adopted to refine the computational efficiency of the model.

First of all, a noteworthy improvement has been implemented as documented in a specific pull request **https://github.com/ggerganov/llama.cpp/pull/5780** on the repository of llama.cpp. This pull request proposes a strategy that involves **rearranging tensor weights** during the model loading phase to better utilize the SIMD features of the ARM instruction set. This rearrangement is particularly optimized for tensors stored in Q4_0 and Q8_0 formats, which are 4-bit and 8-bit respectively, without decimals, as these formats are most easily adapted to this method.

The newly arranged weights are designed to facilitate the use of SIMD instructions within both the ARM NEON and ARM SVE instruction sets for performing matrix and vector operations (vector-matrix multiplication). This adjustment has significantly improved inference efficiency. However, it's important to note that this technique is still at development stage, and the rearranged weights need to be stored, which leads to **an increase in memory consumption**.

Additionally, we made a strategic decision to set the number of threads and batch threads to eight, which corresponds to the eight of physical cores available in our hardware setup.

Aligning the thread count with the number of physical cores is essential for **achieving maximum computational efficiency**. When fewer than eight threads are used, not all physical cores are utilized, leading to **underutilization of the available processing power**. Conversely, setting the thread count higher than the number of physical cores introduces significant inefficiencies, there are more active threads than the available cores, causing threads to compete for CPU time. This leads to frequent context switching, **incurring performance costs due to the overhead associated with these switches**. Therefore, using eight threads ensures each core is utilized optimally—each thread runs on a dedicated core, minimizing unnecessary context switches and cache usage inefficiencies.

A significant portion of computational overhead was identified to be due to the use of 'logf' operations, particularly in the context of rotating position encodings. Using 'perf' for profiling, we optimized these calculations by **substituting 'logf' with 'log2'**, **reducing the number of required 'logf' operations** from four to three. The final solution integrated these approaches, replacing four instances of 'logf_fast' with three, further boosting the efficiency by reducing computational complexity.

The computational optimizations also extended to the arithmetic operations involving dequantization to INT8, which is integral to the vector operations mentioned in the first point.

To achieve more accurate performance metrics, we decide to directly invoke the compiled './main' file from the llama.cpp repository. This change enables a more precise measurement of prefill and decode throughput, ensuring that performance assessments were grounded in the actual capabilities of the software.

To support the rearrangements detailed in the first point, **all tensors previously quantized to f16 were converted to Q4_0** which is slightly different from the official implementation of llama.cpp that the output tensor weight was Q6_K. This change is crucial for aligning the tensor formats with the optimized vector operations, thereby enhancing the model's overall computational efficiency.

These strategic modifications collectively aim at enhancing the processing speed and accuracy of the model, ensuring that it remained capable of handling complex computations efficiently while maintaining high throughput in real-world applications.

C. Future work

During the semifinal experiments, on the provided platform, we encountered unexpected results when we replace the NEON instruction set with the SVE instruction set, as there is no observed improvement in computational speed. This is surprising, especially since similar conditions in the preliminary round (albeit with a 16-core cpu and Qwen7B model) show different outcomes. After various attempts to rectify this, including switching compilers and utilizing the Arm Compute Library, we conclude that hardware limitations might be impacting the efficiency of SVE instruction set execution on this specific machine. Considering different, possibly more capable hardware in future experiments could offer contrasting results.

Additionally, in the pull request **https://github.com/ggerganov/llama.cpp/pull/5780**, a promising approach is suggested where the author proposed completing the rearrangement of tensor weights **during the quantization stage**. This method aims to reduce the number of tensor weights retained during computation, which could lead to lower memory consumption. Implementing this could streamline processing and enhance operational efficiency by minimizing memory management overhead.

IV. Fine-tuning

While model training is extremely resource-expensive, fine-tuning technique is widely applied to large scale pre-trained models to transfer the knowledge from pre-trained models and **fit the desired downstream application domain**. It's done under the premise that pre-trained models learns a wealth of features and patterns from diverse data sources. By following this paradigm, the training cost can be greatly reduced and the performance on specific area is boosted. Besides, fine-tuning can also be used for **recovering the model performance after pruning or quantization**(RFT, recovery fine-tuning).

Even with fine-tuning, it can still be expensive to tune the model if we train all the parameters of the model. Some methods are proposed to solve the problem, among which Lora [4] is most widely used. Lora is based on the hypothesize that the change in weights during model adaptation has a low "intrinsic rank", thus we can train **a low rank adaption** rather than the original full rank parameter matrix, which can significantly save memory and training time. Specifically, suppose the original weight matrix is of $R^{a\times b}$, two new matrix A and B for adaption which are of $R^{a\times r}$ and $R^{r\times b}$ respectively are trained during fine-tuning instead of original weights. After training, we merge $A \times B$ to the original weight matrix via element-wise addition.

We adopt llama-factory [8] to fine-tune the model with Lora. Several datasets are tested and we find Alpaca-gpt4 [6] and UltraChat [2] can slightly improve the model performance(from 73.12 to about 73.8), but other datasets may cause the performance to deteriorate. We attribute this to the dataset scale. It's worth mentioning that English or multi-language datasets are better because the test dataset piqa is in English. We also test some different hyper-parameters such as learning rate, but it makes little difference. We evaluate Lora checkpoints every 1000 iterations and pick the best one, for there is fluctuation during training.

Many follow-up works emerges trying to improve Lora, such as qlora [1] and qalora [7]. The former introduces a number of innovations to further save memory while the latter tries to combine fine-tuning and quantization together to mitigate the secondary accuracy loss in quantization after fine-tuning.

However, we meet difficulties when merging the weights and adapting these methods to llama.cpp. So, we just use naive Lora in the grand challenge and we hope we can adapt qlora and qalora in future works.

V. Conclusion

In this report, we comprehensively detail our endeavors to enhance large language models through a tripartite approach, each focusing on a critical aspect of model optimization. Our quantization strategy successfully minimized model size and computational requirements without degrading accuracy, demonstrating a balance between accuracy and resource occupation, which is crucial for deployment in varied hardware scenarios.

Moreover, our extensive work on llama.cpp has culminated in substantial improvements in inference efficiency. By integrating advanced instruction sets and optimizing tensor operations, we have maximized the utilization of hardware capabilities, overcoming significant technical challenges with innovative solutions.

Lastly, our fine-tuning practices refine the models' accuracy on specific tasks, with strategic adjustments to model parameters and training methods. These efforts ensure that the model not only meets but exceeds expectations in practical applications.

Together, these contributions underscore our commitment to advancing the field of large language models, making them more scalable, efficient, and applicable across a broader range of computational environments. This work not only enhances current model capabilities but also sets a foundation for future innovations in the realm of artificial intelligence.

References

[1] Tim Dettmers et al. "QLoRA: Efficient Finetuning of Quantized LLMs". In: *Advances in Neural Information Processing Systems*. Ed. by A. Oh et al. Vol. 36. Curran Associates, Inc., 2023, pp. 10088–10115. URL: https://proceedings.neurips.cc/paper_files/paper/2023/file/1feb87871436031bdc0f2beaa62a049b-Paper-Conference.pdf.

[2] Ning Ding et al. *Enhancing Chat Language Models by Scaling High-quality Instructional Conversations*. 2023. arXiv: 2305.14233 [cs.CL].

[3] Elias Frantar et al. "GPTQ: Accurate Post-training Compression for Generative Pretrained Transformers". In: *arXiv preprint arXiv:2210.17323* (2022).

[4] Edward J. Hu et al. "LoRA: Low-Rank Adaptation of Large Language Models". In: *CoRR* abs/2106.09685 (2021). arXiv: 2106.09685. URL: https://arxiv.org/abs/2106.09685.

[5] Ji Lin et al. "AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration". In: *arXiv* (2023).

[6] Baolin Peng et al. *Instruction Tuning with GPT-4*. 2023. arXiv: 2304.03277 [cs.CL].

[7] Yuhui Xu et al. *QA-LoRA: Quantization-Aware Low-Rank Adaptation of Large Language Models*. 2023. arXiv: 2309.14717 [cs.LG].

[8] Yaowei Zheng et al. *LlamaFactory: Unified Efficient Fine-Tuning of 100+ Language Models*. 2024. arXiv: 2403.13372 [cs.CL].