Software and hardware collaborative optimization of Qwen large language model for ARM CPU

1st Xingyu Zhu 2nd Pengcheng Yang 3rd Tingjie Yang 4th Jialong Wang

***Abstract*—With the emergence of ChatGPT based on Transformer architecture, artificial intelligence has returned to the public's view, and large language model has received widespread attention. However, due to its very large parameter size as well as arithmetic requirements, it often requires a high hardware threshold, such as multi-GPU clusters. In order to deploy and implement LLMs inference on the end-side, and to lower the threshold of using LLMs, we optimize the hardware level of Qwen model for CPUs. First, we use the LLM low privilege weighting method to quantize AWQ and GGUF for the Qwen model, which reduces the amount of weight storage and computation of the LLM. Second, we replace the inference backend with LLaMA.cpp and optimize the matrix multiplication kernel. Finally, we use the Arm instruction set optimization, which is more suitable for the Yitian 710 CPU, to improve the instruction parallelism. After testing, our optimisations with 4-bit quantized model improve the prefill throughput by 312.05% and decode throughput by 1598.13% compared to the original model, reduce the memory footprint to 25% of the original, and reduces the accuracy test for piqa by only 1.89%.**

***Index Terms*—LLM, CPU, quantization, matrix multiplication acceleration, instruction set optimization**

I. Introduction

In the past, natural language processing has undergone different paradigm changes. Originally, natural language processing relied on hand-written rules to process text [1], but with the improvement of computing power, methods using statistical information [2] in large-scale corpora have gradually become mainstream. Machine learning methods such as support vector machines, decision trees and naive Bayes have been widely used in natural language processing. In recent years, the rise of deep learning methods such as recurrent neural networks [3] and long-term memory networks has led to a new wave of natural language processing. The emergence of Transformer [4] architecture makes pre-trained models become an important development direction of natural language processing. Large language models based on Transformer architecture have excellent performance in text translation, text content creation, dialogue generation, code development and other fields.

The Transformer structure abandons the traditional CNN and RNN structure and adopts a self-attention mechanism to realize sequence modeling. This mechanism allows the model to process sequential data by weighting information at different locations without relying on a fixed window size or sequential order. In Transformer, the input sequence is stacked with multiple self-attention layers and feedforward neural network layers, then processed through residual linking and layer normalization, and finally passed through the output layer to generate a final prediction or representation. The design of this structure brings many advantages. First, thanks to the introduction of a self-attention mechanism, Transformer can process all locations in the input sequence in parallel, speeding up training and reasoning. Second, Transformer is excellent at modeling long distance dependencies, unlike traditional RNN structures that are subject to disappearing gradients or explosions. Transformer has the flexibility to handle sequences of different lengths without having to input the entire sequence at once like an RNN. However, for very long sequences, due to the $O(n^2)$ computational complexity of the self-attention mechanism, it may lead to limitations in memory and computational resources. In response to this problem, various improved Transformer variants have been proposed, including local attention-based models [5] and layered attention mechanisms [6]to reduce computational complexity.

LLM based on the Transformer architecture, such as GPT [7] and BERT [8], typically have up to billions of parameters, which makes them difficult to deploy directly on resource-constrained end-side devices such as smartphones and IoT devices. These devices often lack sufficient processing power, memory, and storage to support such large models, and their operation also leads to high energy consumption and slow response times. In order to run these models efficiently on end-side devices, model size and computational complexity need to be reduced through model optimisation techniques such as pruning and quantisation. Through these optimisation measures, large models can not only be deployed on end-side devices, but also maintain reasonable inference speed and accuracy, thus broadening the application scenarios of AI technologies and making end-side AI applications smarter and more convenient.

The rest of this paper is organised as follows. Section II describes in detail the Yitian 710 CPU, the model we use, Qwen. Section III describes our optimisation scheme and results, and section IV concludes.

II. Background and Motivation on Yitian 710 CPU

Qwen-1.8B is a 1.8 billion parameter large language model developed by Alibaba Cloud. Based on the Transformer architecture, it is trained with a diverse and massive pre-training dataset, including internet text, professional books, and code among others. A significant highlight of this model is its low-cost deployment capability; it can operate directly on hardware platforms such as the Yitian 710 CPU, using minimal VRAM resources while maintaining efficiency during inference and fine-tuning processes. Qwen-1.8B possesses an extensive vocabulary covering various languages, and it performs exceptionally well in multiple downstream tasks in both Chinese and English.

The Yitian 710 CPU, a domestic high-performance processor, utilizes the Armv9 architecture and supports the SVE2 instruction set, offering powerful computational strength and an excellent power efficiency ratio. This makes it an effective support for deploying large models on CPU platforms.

In competition, we deploy the Qwen-1.8B model on the Yitian 710 CPU. Leveraging the model's low memory footprint and our in-depth optimization on the Yitian 710 CPU, our goal is to demonstrate that this model can still achieve high inference speeds and accuracy in a CPU environment. This showcases its powerful capabilities and adaptability under limited computational resources, further breaking the confines of traditional GPU deployments and broadening the application horizons of large models.

III. Hardware and software optimisation solutions

We first derive the AWQ scale of the model using AWQ quantization techniques, then quantize the model into different bits using the gguf quantization model, and use llama.cpp to inference the model. Second, we refine llama.cpp using GEMM and GEMV optimization methods to optimize the operator kernel.

A. Model compression

In this competition, we employed an effective quantization technique — Activation-aware Weight Quantization [9] (AWQ), a collaborative development between MIT, SJTU, and Tsinghua University for large model quantization. The essence of the AWQ approach is to identify and preserve the top 1% of salient weights in large models, significantly reducing the performance degradation commonly associated with the quantization process. This unique method circumvents the need for backward propagation or model reconstruction, ensuring outstanding universality of the Large Language Model (LLM) across various tasks and domains, avoiding overfitting to calibration datasets, and maintaining hardware execution efficiency without the necessity of reordering data layouts.

The AWQ technique selects salient weights by analyzing the interplay between weight and activation distributions, a significant departure from traditional weight-based selection methods. Instead, it opts for selections based on activation magnitude, which has proven to significantly enhance the outcomes of model quantization. However, merely preserving salient weights in high-precision formats such as FP16 can be conducive to performance retention but may compromise overall model efficiency.

To safeguard salient weights while diminishing quantization loss, AWQ employs an activation-aware scaling mechanism. It adjusts the weight W and the activation X in conventional matrix multiplication as demonstrated in Eq. (1), opting for Eq. (2) to balance significant parameters within weights and activations [10].

$$Y = X \cdot W \tag{1}$$

$$Y = (X\frac{1}{s}) \cdot (sW) = \tilde{X} \cdot \tilde{W} \tag{2}$$

By utilizing heuristic rules and automatically searching for the optimal scaling factor, the AWQ method strives to minimize the quantization error of the output as defined by Eq. (3). This search process encompasses both activation magnitude and weight magnitude, selecting the hyper parameters through grid search.

$$s^* = \arg\min_s L(s), \quad L(s) = \|Q(W \cdot s)(s^{-1} \cdot X) - W \cdot X\| \tag{3}$$

By performing inference on a targeted dataset, we captured the activation outputs across the model's layers in real time and recorded the AWQ quantization scale factors. These factors were subsequently utilized in further quantization processes [11] to improve the model's quality. Utilizing the PIQA test set, we evaluated the model pre- and post-AWQ implementation. As demonstrated in Table I, quantizing the model to an int8 format incurred a precision loss; however, the use of AWQ mitigated this loss, restoring a degree of the model's accuracy.

TABLE I
MODEL ACCURACY BEFORE AND AFTER AWQ

model	piqa
Qwen1.8B-F16	0.7252
Qwen1.8B-INT8	0.72198
Qwen1.8B-INT8-AWQ	0.7227

B. Inference back-end optimization

After obtaining the AWQ scales of the model, we convert the model weight files into GGUF (GPT-Generated Unified Format) files and apply quantization at various bit precisions as required. Then, to maximize CPU computational efficiency, we migrate the inference framework to llama.cpp for optimization and configuration.

GGUF, initiated by Georgi Gerganov, is a binary file format designed for efficient storage and exchange of large language models. Its main goal is to speed up the loading and saving of models while maintaining the ease of file readability. This format supports rapid loading through memory mapping (mmap), making it suitable for model inference across various execution environments. Typically, models developed in PyTorch or other frameworks are converted to GGUF format to optimize inference efficiency. This conversion involves optimizing data structures and applying memory mapping to reduce resource consumption. The core advantages of GGUF include its single-file deployment capability, high extensibility, and comprehensive inclusion of all necessary model loading information, which greatly simplifies the model's deployment and sharing processes. The file structure of GGUF is meticulously defined,

TABLE II
PERFORMANCE OF DIFFERENT BIT COUNT MODELS

Model	RSS(MB)	VMS(MB)	Prefill	Decode
Q8_0	1929.11	2251.12	82.54	45.88
Q6_K	1638.62	1889.41	42.12	32.89
Q5_K_S	1438.50	1649.84	44.71	35.08
Q5_K_M	1516.09	1728.41	44.62	34.86
Q5_0	1420.76	1633.72	47.37	36.94
Q4_K_S	1311.59	1486.14	60.27	43.83
Q4_K_M	1393.31	1568.50	56.75	41.52
Q4_1	1348.48	1542.80	45.05	36.68
Q4_0	1275.80	1451.88	49.39	39.67
Q3_K_S	1154.89	1292.93	44.72	35.74
Q3_K_M	1240.73	1376.86	49.75	38.65
Q3_K_L	1266.73	1404.31	47.14	37.19
Q2_K	1072.50	1181.14	47.84	38.79
F32	6209.32	7396.15	85.67	15.69
f16	3297.02	3893.15	158.63	29.33

including the file header, metadata key-value pairs, tensor information, and weight data, and supports both little-endian and big-endian formats, ensuring compatibility across different computing platforms.

To perform inference with GGUF models, we also use llama.cpp. This is another pure C/C++ project developed by Georgi Gerganov, designed to allow large language models (LLMs) to run efficiently on consumer-grade hardware. llama.cpp does not rely on external libraries, supports ARM NEON and AVX2 instruction sets, and implements mixed-precision computing with F16 and F32 as well as 4-bit quantization, significantly reducing the demands on storage and memory. llama.cpp is capable of running across platforms, including on Android devices, offering a resource-efficient and straightforward deployment solution that makes the application of large models more widespread and feasible.

We quantize the GGUF models to the following bit levels and perform inference using llama.cpp, while recording performance metrics such as MAX RSS and MAX VMS memory usage, and throughput during the prefill and decode phases, as shown in TableII

As can be seen from the data in Table II, the fact that llama.cpp does not yet support int4/8 results in a throughput that fails to reach the level of f16 for the bit number settings of q8 and q4. Therefore, we will optimise llama.cpp, specifically tuning GEMV (General Matrix-Vector Multiplication) and GEMM (General Matrix-Matrix Multiplication) for int4/8, and will add some ARM instruction sets to speed up the process.

C. Optimization of General Matrix-Matrix Multiplication

The inference process of a large language model based on the Transformer architecture is divided into two phases: prefill and decode. In the prefill phase, the input sequence undergoes processing through N identical layers, each of which performs word embeddings and positional prefill on the input. Each prefill layer comprises two sub-layers: a multi-head attention layer and a feedforward network layer.

Similarly, in the decode phase, there are N identical layers that apply word embeddings and positional prefill to the output sequence. Each decode layer consists of three sub-layers: a masked multi-head attention layer, a multi-head attention layer, and a feedforward network layer. The query, key, and value vectors for both the multi-head attention and masked multi-head attention layers in both the prefill and decode phases are derived from the input vectors through distinct fully connected layers.

Distinctively, in the decode phase, the multi-head attention layer takes as input the output vector KV from the prefill phase combined with the output vector Q from the decode phase's masked multi-head attention layer. The attention mechanism maps queries and key-value pairs into a higher-dimensional space, where weight values are computed based on the interaction between the query and keys. The corresponding mathematical formula for this computation follows.

$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{4}$$

The two-dimensional matrix QK^T is obtained through the multiplication of the query vector and the transpose of the key vector. After undergoing scaled dot-product and normalization operations, this matrix is then multiplied by the value vector. This process involves substantial matrix computations, hence an efficient matrix multiplication kernel has an immeasurable impact on the inference time of a large language model.

Algorithm 1 General Matrix-Matrix Multiplication

```
1: for $m = 0; m < H; m = m + 1$ do
2:     for $n = 0; n < H; n = n + 1$ do
3:         $QK[m][n] = 0;$
4:         for $w = 0; w < W; w = w + 1$ do
5:             $QK[m][n] += Q[h][w] * K[w][h];$
6:         end for
7:     end for
8: end for
```

Assuming the size of the input after multiplication with a weight matrix is $H*W$, taking the QK^T matrix multiplication as an example, code segment Primitive describes the completion of the QK^T matrix multiplication operation using three nested loops. The innermost loop contains two computational operations, requiring four memory accesses per result: two accesses to elements within the QK^T matrix, and one access each to elements in the Q and KT matrices.

Treating the $I*J$ submatrix within the QK^T matrix as a unit for computation, according to code segment Post-optimization, if the reusable operands in the computational operations are stored as temporary variables by unrolling the first two loops, the number of memory accesses can be reduced from the original $4H^2$W to (2 + 1/I + 1/J)H^2W. When I and J are sufficiently large, the memory access count can be halved compared to the original. If the $I*J$ submatrices within the QK^T matrix are themselves stored as temporary variables, the number of memory accesses to the QK^T matrix reduces from $2H^2$W to HW, which is negligible relative to HW. When $I = J = N_{opt}$, the resulting speedup factor is $2N_{opt}$ times.

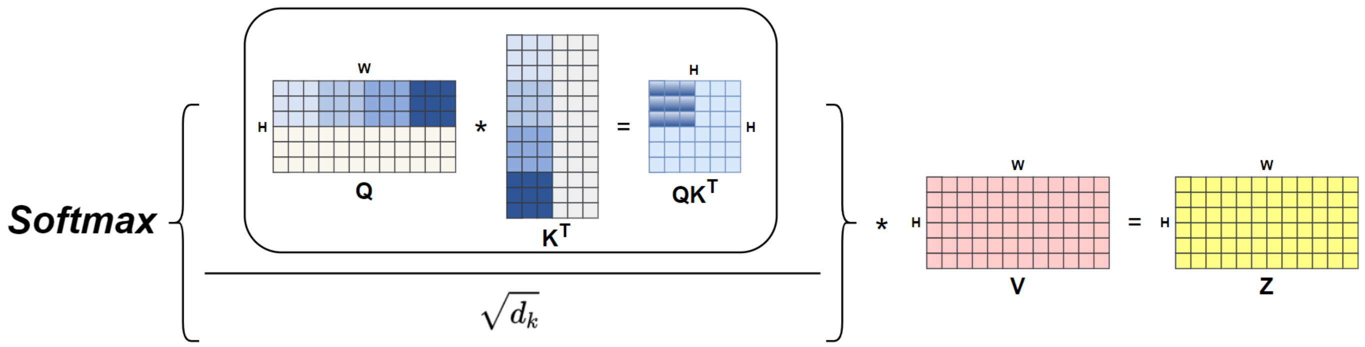

Fig. 1. Attention matrix multiplication diagram

Algorithm 2 GEMM Optimization

```
1: for $m = 0; m < H; m = m + I$ do
2:     for $n = 0; n < H; n = n + J$ do
3:         for $i = 0; i < I; i = i + 1$ do
4:             for $j = 0; j < J; j = j + 1$ do
5:                 $QK[m + i][n + j] = 0;$
6:             end for
7:         end for
8:         for $w = 0; w < W; w = w + K$ do
9:             for $k = 0; k < K; k = k + 1$ do
10:                for $x = 0; x < I; x = x + 1$ do
11:                    for $y = 0; y < J; y = y + 1$ do
12:                        $QK[m+x][n+y] += Q[m+x][w+k] * K[w + k][n + y];$
13:                    end for
14:                end for
15:            end for
16:        end for
17:    end for
18: end for
```

D. Optimization of General Matrix-Vector Multiplication

Efficiency of linear layer computations significantly impacts the overall throughput of Large Language Models during inference. This process can be divided into two main phases: prefill and decode. In the prefill phase, throughput is crucially influenced by the computational speed of the GEMM (General Matrix-Matrix Multiplication) kernel. However, during decode phase, the GEMV (General Matrix-Vector Multiplication) kernel becomes more critical. While GEMV has lower computational intensity compared to GEMM, it still significantly influences model inference throughput during decode phase. Therefore, optimization efforts of decode phase throughput primarily focus on minimizing memory access overhead and accelerating the GEMV kernel. This work explores weight rearrangement and the utilization of the SVE (Scalable Vector Extension) instruction set on Armv9 CPU to optimize the GEMV kernel and memory access.

Within the Arm architecture, the sdot (Single-precision Dot Product) instruction set facilitates efficient vector and matrix multiplications for single-precision floating-point numbers. By performing multiply-accumulate operations within a single instruction cycle, sdot significantly enhances processor data throughput. However, leveraging the sdot instruction set for acceleration needs specific weight rearrangement to improve memory locality and, consequently, memory access efficiency. Due to the row-major order storage format employed by the LLaMA.cpp inference backend framework, tensor elements are not stored contiguously in memory. This non-contiguous layout hinders the continuous reading of elements within the same kernel. To address this, weight rearrangement into contiguous memory is required to enable sdot to accelerate GEMV kernel. Figure 2 illustrates the data rearrangement process with an example of channel = 4 and kernel size = 3x3.

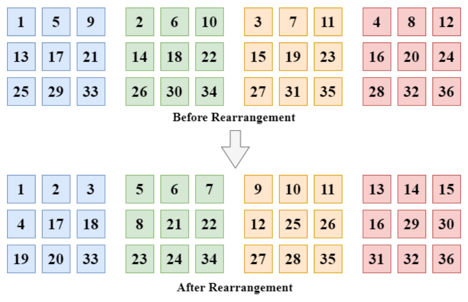

Fig. 2. Weight Rearrangement Diagram

In the original data layout, the 4 kinds of colors represent 4 channels, with each channel containing 9 squares corresponding to the elements within a 3x3 kernel. The numbers within the squares indicate their order in memory. After rearrangement, elements belonging to the same channel and kernel are stored contiguously in memory, enhancing memory access efficiency and providing the foundation for sdot instruction set acceleration.

ARM SVE is a next generation of instruction set extension of AArch64 architecture. Unlike other SIMD sets that only support fixed vector register sizes, such as NEON with its 128-bit limitation, SVE supports flexible vector widths. This flexibility allows scaling based on computational requirements, ranging from 128 bits up to 2048 bits with a unit of 128-bit.

TABLE III
Performance of INT4/8 Models with Different Optimization Methods

Optimization	Model	RSS(MB)	VMS(MB)	Prefill	Decode
None	Original Model	10627.37	12228.18	86.37	3.75
None	Q4_0	1275.80	1451.88	49.39	39.67
	Q8_0	1929.11	2251.12	82.54	45.88
fp16	Q4_0	1269.03	1418.42	76.20	45.20
	Q8_0	2068.29	2217.67	97.33	48.56
fp16+GEMM	Q4_0	1306.87	1558.56	265.45	47.63
	Q8_0	2469.66	2628.53	289.86	50.96
fp16+GEMV	Q4_0	2908.44	3059.24	147.42	59.69
	Q8_0	5793.04	5943.84	136.18	50.98
fp16+GEMM +GEMV	Q4_0	2913.04	3063.84	269.52	59.93
	Q8_0	5792.89	5943.84	306.70	49.91

This paper leverages SVE instructions to implement GEMV kernels for INT8 and INT4 data types, enabling parallel processing of multiple data elements during matrix-vector multiplications. Additionally, the GEMV kernels use matrix blocking techniques to improve cache utilization and further reduce memory access overhead, ultimately enhancing GEMV kernel computational speed.

Furthermore, automatic search and code optimization for the Armv9 architecture can be achieved by appending additional gcc / g++ compiler options like fp16 and i8mm instructions.

IV. Result and Evaluation

Testing based on the aforementioned optimization methods yielded the results presented in Table III. As observed, for both INT4 and INT8 quantized models (Q4_0 and Q8_0), After GEMM kernel optimization, the prefill phase exhibited a 5.4x and 3.51x speedup, respectively, compared to the data generated by the unoptimized LLaMA.cpp inference framework. This highlights the significant improvement in prefill phase throughput achieved through GEMM kernel optimization. Additionally, after GEMV kernel optimization, the decode phase demonstrated a 1.50x and 1.11x speedup, respectively, indicating the positive impact of GEMV kernel optimization on decode phase throughput. However, a moderate increase in RSS and VMS memory usage was observed, attributed to the additional computational resources required for weight rearrangement. Through a series of optimizations targeting the GEMM and GEMV kernels, along with appending compiler options, the final results for the INT4 and INT8 quantized models, compared to the original models without software and hardware optimizations, exhibited a 3.12x and 3.55x improvement in prefill phase throughput, respectively. Additionally, the decode phase demonstrated a 15.98x and 13.31x improvement in throughput, respectively. Furthermore, RSS memory usage was reduced to 0.27x and 0.55x, and VSS memory usage was reduced to 0.25x and 0.49x, respectively. These findings demonstrate the effectiveness of our optimization methods in achieving substantial LLMs inference acceleration.

V. Conclusion

This study effectively demonstrates the feasibility and benefits of a software and hardware collaborative optimization approach for the Qwen large language model on ARM CPUs. By employing techniques such as LLM low-precision weighting, inference backend replacement, matrix multiplication kernel optimization, and Arm instruction set optimizations, the optimized model exhibits substantial improvements in performance, reduced memory footprint, and maintained accuracy. These achievements enable the deployment of large language models on resource-constrained end-side devices, challenging traditional GPU reliance and expanding their application potential in diverse contexts.

References

[1] Winograd, Terry, "Understanding natural language." Cognitive psychology, 1972, 3(1): 1-191.

[2] Jelinek, Frederick, "Statistical methods for speech recognition" MIT press, 1998.

[3] Elman J L, "Finding structure in time." Cognitive science 14.2 (1990): 179-211.

[4] Vaswani, Ashish, et al., "Attention is all you need." Advances in neural information processing systems, 2017, 30.

[5] Child, Rewon, et al. "Generating long sequences with sparse transformers." arXiv preprint arXiv:1904.10509, 2019.

[6] Kitaev, Nikita, Łukasz Kaiser, and Anselm Levskaya. "Reformer: The efficient transformer." arXiv preprint arXiv:2001.04451, 2020.

[7] Radford, Alec, et al.,"Improving language understanding by generative pre-training.", 2018.

[8] Devlin, Jacob, et al.,"Bert: Pre-training of deep bidirectional transformers for language understanding." arXiv preprint arXiv:1810.04805, 2018.

[9] Lin J, Tang J, Tang H, et al., "AWQ: Activation-aware weight quantization for llm compression and acceleration," in The Seventh Annual Conference onMachine Learning and Systems, 2024, (accepted).

[10] Xiao, Guangxuan, et al. "Smoothquant: Accurate and efficient post-training quantization for large language models." International Conference on Machine Learning. PMLR, 2023.

[11] QWEN Documentation, "GGUf quantization," QWEN Documentation, [Online]. Available: https://qwen.readthedocs.io/zh-cn/latest/quantization/gguf.html.