*Mixed Quantization of LLM Inference*

*on Arm Devices*

Zhaode Wang   
AlibabaBeijing, China  
zhaode.wzd@alibaba-inc.com

Jianhao Zhang   
Kunlun IncBeijing, China  
zhjh123@mail.ustc.edu.cn

Jin Yao

TsingmicroNanjing, China  
yaojin@tsingmicro.com

*Abstract*—This paper delves into the deployment and optimization of Large Language Models (LLMs) on edge devices, addressing the challenges posed by the massive number of model parameters and hardware limitations. Practical applications on ARM platforms demonstrate that these optimization measures effectively reduce computational resource consumption and enhance inference speed. Additionally, operator optimizations, such as loop tiling and the utilization of high-throughput computing instructions, further boost model performance. Lastly, this paper addresses the issue of accuracy loss due to model quantization by introducing a heuristic search method to compute the optimal mixed quantization scheme. This approach aims to balance model accuracy, memory usage, and performance, achieving a harmonious effect.

Keywords—Large Language Models (LLM), Model Optimization, Mixed Quantization

# Introduction

Large Language Models (LLMs) have achieved remarkable performance in most downstream tasks of natural language processing, such as text comprehension, text generation, sentiment analysis, machine translation, and interactive question answering. However, the challenge of efficiently deploying LLMs on the edge is significant due to the billions or even trillions of model parameters. Moreover, the growth rate of model parameters far exceeds the rate of hardware performance improvement. Therefore, both academia and industry have begun exploring methods of model compression, data flow optimization, and operator invocation to deploy and run large models efficiently under limited hardware conditions.

# Base model evaluation

Qwen2.5-0.5B was selected as the evaluation model, and we compared several versions provided by the official source to choose the one with higher performance as the base model. Qwen offers pre-trained base models, including: Qwen2.5-0.5B, and a dialogue-optimized version fine-tuned with instructions: Qwen2.5-0.5B-Instruct. Additionally, the PAI team provides a distilled version based on Qwen2.5-0.5B-Instruct, known as DistilQwen2.5-0.5B-Instruct. This version uses well-known open-source collections such as Magpie, Openhermes, and Mammoth 2 to distill Qwen2.5-0.5B-Instruct to enhance its performance. We conducted evaluations on these three models using the hellaswag, arc\_challenge, and ceval datasets, with scores shown in the TABLE 1.

From the table, it can be seen that the instruction fine-tuned version shows an average improvement of 1.8% over the base model across the three test sets; the distilled version further enhances the performance by an average of 0.5% compared to the instruction fine-tuned version. Therefore, we have chosen the distilled version as the base model.

| Model | DataSet | | |
| --- | --- | --- | --- |
| hellaswag | arc\_challenge | ceval |
| Qwen2.5-0.5B | 0.5226 | 0.3242 | 0.5171 |
| Qwen2.5-0.5B-Instruct | 0.5237 | 0.3345 | 0.5312 |
| DistilQwen2.5-0.5B-Instruct | 0.5305 | 0.3387 | 0.5275 |

# performance optimization

## performance analysis

When delving deep into the performance of Large Language Models (LLM), it is crucial to understand the composition of the backbone network. The backbone of an LLM consists of a series of consecutive decode blocks. As illustrated in the figure, the execution of a decode block involves two main computational operations: Linear (represented in green) and MatMul (represented in yellow). These operations not only perform complex mathematical computations but are also accompanied by memory operations such as split, concat, and transpose, collectively known as Memory operators. Thus, we can categorize the core operations during LLM model inference into three types: Linear, MatMul, and Memory (Split, Concat, Transpose).

Through testing on ARM CPUs, we are able to conduct a detailed analysis of the time consumption of the aforementioned three core operators during these two stages. For example, an analysis of the time taken by a single block reveals:

* During the prefill stage, the time consumption of the Linear operators is relatively stable, accounting for over 93% of the total time.
* During the decode phase, as the sequence length increases, the proportion of time spent on Linear operators decreases, while that of MatMul and Memory operators increase. MatMul and Memory operations are mainly associated with the computation of Attention.

Based on the above analysis, our primary optimization focus should be on the Linear operators, followed by the computation processes pertaining to Attention(contain MatMul and Memory).

## Optimizing Linear Operators

Identify applicable funding agency here. If none, delete this text box.

During the inference process of Large Language Models (LLM), the computational efficiency of Linear layers plays a critical role in the overall performance. This computation is divided mainly into two stages: during the prefill stage, where a large amount of input data is processed, Linear layers perform matrix multiplication (GEMM), which is a compute-intensive process; during the decode stage, Linear layer computations shift to matrix-vector multiplication (GEMV), making the efficiency of memory access increasingly crucial. To enhance performance across these two stages, we have implemented the following strategies:

* Quantization: We have adopted a W4/8A8 quantization scheme, quantizing the model's weights (W) to 4/8 bits and the activations to 8 bits. This approach significantly reduces the model's memory footprint and improves performance because data with a lower bit-width requires less memory bandwidth for reading and writing.
* Loop Tiling: We employ loop tiling to specifically reorder data to improve memory locality and enhance memory access efficiency. This technique involves organizing the operations in the compute process in a manner that minimizes the distance and frequency between memory accesses, thereby reducing cache misses and improving execution speed.
* Choosing Higher Throughput Computing Instructions: We select SIMD instruction sets that offer higher throughput for carrying out the core computations, coupled with the use of assembly to develop multi-sized kernels. This strategy is aimed at accelerating matrix multiplication operations by making more efficient use of the processor's capabilities.

## Quantization

In modern Large Language Models (LLMs), linear layers often include billions of parameters, posing challenges for deployment on memory-limited mobile devices. With 7 billion parameters (7b) requiring roughly 14GB of memory even in fp16, compressing these weights is crucial. Quantization offers a solution by reducing memory requirements while preserving model performance. The relative similarity of linear layer weights makes them suitable for low-bit quantization, resulting in minimal performance impact and effectively doubling the theoretical performance of 4/8-bit quantization compared to traditional 8-bit.

Using asymmetric quantization, models like the 7b can operate within 3.5~7GB, enabling devices with 8GB of memory to handle such models. This approach decreases memory access fourfold during computation, thereby enhancing performance. Traditionally, mixed precision computing combined 4-bit quantized weights with floating-point inputs, demanding significantly more memory access and using less efficient floating-point SIMD instructions.

To improve this, a dynamic W4/8A8 quantization scheme was implemented, converting inputs into 8-bit integers and utilizing more efficient integer instructions such as sdot. This technique reduces memory access volume and enhances computational efficiency on ARM platforms, where integer instruction peak performance significantly surpasses that of floating-point calculations. The W4/8A8 strategy effectively improves both computational performance and memory access efficiency.

## Loop Tiling

For the matrix multiplication operation [e, l] @ [l, h] -> [e, h], the number of memory accesses is: 2ehl + eh. In practice, there is redundancy in memory access since both weights and inputs are accessed redundantly h and e times, respectively. Applying loop tiling to e and h can reduce the number of redundant memory accesses. For a matrix multiplication with a tiling size of ep, hp, the number of memory accesses is: e/ep \* h/hp. Therefore, by performing tiling on inputs and weights and implementing computation kernels for ep, hp, we can reduce memory access redundancy.

The selection of ep and hp is constrained by the number of physical registers available, so we can solve the following formula to obtain the optimal ep and hp values, thereby realizing the corresponding kernel:

To enhance memory locality and reduce the cost of memory access, we integrate the W4A8 computation paradigm with device-supported computational instructions for specific data rearrangement of inputs and weights. Specifically, considering that the shape of input data is usually [e, l], and the shape of weights is [h, l], we rearrange these data into [l/pack, e/ep, ep, pack] and [h/pack, l/pack, pack, pack]. Here, "pack" refers to a block size carefully chosen based on the computation instructions supported by the hardware. For example, when the system supports the smmla instruction, we choose pack = 8; when the system supports the sdot instruction, we choose pack = 4. This rearrangement strategy allows the computation kernel to perform more compact matrix multiplication operations: multiplying matrices of sizes [ep, pack] and [pack, pack] to produce an output matrix of size [ep, pack]. Furthermore, we have implemented various compute kernels for the batch dimension, taking into account the number of available registers, which enables us to fully exploit the computational capabilities to improve memory access efficiency. In the linear layers of Large Language Models (LLM), the number of output channels (h) is often large. This provides us an opportunity to execute parallel computations across the [h/pack] dimension, fully leveraging modern multicore capabilities to boost multicore performance.

Considering that merging 4-bit weights into 8-bit weights and loading them using vector instructions results in non-contiguous original data, this can be addressed during weight rearrangement. The data in [pack, pack] can be reshaped into [n, 16, 2] and then transposed into [n, 2, 16]. Thus, when extracting 4-bit data from 8-bit data, direct computation can proceed without the need for reordering.

## Higher Throughput Computing Instructions

The smmla instruction on ARM platforms is designed to perform multiplication and accumulation operations on int8 matrices. It multiplies elements from two 128-bit registers, each holding 2 rows of 8 int8 elements and accumulates the results into a 128-bit register containing 2 rows of 2 int32 elements. Essentially, it performs the operation [2, 8] @ [8, 2] -> [2, 2], executing 32 multiplications and 32 additions in total; smmla theoretically offers double the performance of sdot. Since we have implemented W4A8 quantization and rearranged the data in packs of 8, we can use smmla for matrix multiplication implementation. In GEMM computations, its theoretical performance is double that of sdot, and for GEMV computations, it is the same as sdot.

## Optimizing Attention Operators

The computation time of the attention operator increases with the growing key-value (kv) cache due to the expanding scale of past-key-value tensors in the attention operation, which adds to the computational load. This results from data-intensive operations like concatenation, transposing, and scaling matrix multiplication within the attention mechanism. In ONNX implementations, past key-value tensors as inputs and outputs prompt substantial memory operations. To mitigate this, operators can be fused to treat the kv-cache as internal state, reducing memory data processing to writing key-value tensors of length 1 per invocation. Memory for this state can be pre-allocated in chunks based on pre-fill requirements, with reallocations aligned with the chunk size or maximum length to minimize overhead.

Matrix multiplication, especially the query @ key and qk @ value operations, is time-intensive within attention mechanisms. Data rearrangement along e and h dimensions can enhance computational efficiency, despite incurring memory copying costs. After operator fusion, query rearrangement can merge with the transpose operation, and key-value rearrangements can fuse with concatenate-transpose operations, optimizing to a single memory copy for all preparations. For matrix multiplication, float16 balances performance and memory use, while float32 ensures precision for softmax operations. Bfloat16 offers higher throughput compared to float16 but demands careful optimization and validation for stability and cross-platform compatibility.

# Mixed quantization

Earlier, it was mentioned that quantization is an effective method for optimizing model memory and performance. However, quantization can also lead to a decrease in model accuracy. Generally, the lower the number of bits used in quantization, the greater the impact on accuracy. Therefore, we need to balance model accuracy, memory usage, and performance.

## Comparison of Different Quantization

When quantizing weights, the range of quantization bits typically falls between 2 to 8. Moreover, based on the method of quantization, there are classifications such as per-tensor, per-channel, and per-group. Considering computational accuracy and instructions, quantization methods for LLMs generally favor per-channel and per-group quantization, with the bit number commonly set at 4 or 8. Utilizing ARM CPUs, we compare these conventional quantization methods.

Initially, we conduct theoretical analysis. Per-channel quantization utilizes a single set of scale and zero-point for the data within each channel, resulting in fewer extra parameters after quantization. During computation, int8 calculations can be performed across the entire channel, followed by dequantization, yielding high computational efficiency. Per-group quantization divides channels into multiple groups, each possessing its own set of scale and zero-point, leading to more parameters post-quantization. Computation requires dequantizing results for each group, offering higher precision but lower efficiency. Weights after 4-bit quantization are smaller, occupy less memory, and provide higher performance during the memory-access-constrained decode phase. In contrast, 8-bit quantization produces larger weights with higher memory usage, providing better performance in the prefill phase due to reduced shifting operations but lower performance albeit higher precision during the decode phase.

Based on these characteristics, we analyzed different quantization methods, with results shown in TABLE 2. It can be observed that 4-bit quantization, whether per-channel or per-group, results in significant accuracy reduction compared to floating-point models. Conversely, models with 8-bit per-channel quantization exhibit minimal accuracy loss relative to floating-point models. Therefore, if maintaining accuracy is the goal, 8-bit quantization should be employed.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | avg score | memory | Prefill t/s | Decode t/s |
| 4bit group | 0.42198 | 266M | 1082.12 | 138.28 |
| 4bit channel | 0.34452 | 240M | 1227.19 | 183.61 |
| 8bit group | 0.46331 | 501M | 1102.00 | 101.47 |
| 8bit channel | 0.46540 | 475M | 1230.48 | 130.89 |

## Mixed Quantization

While considering accuracy, we must also take memory usage and decode speed into account. Mixed precision quantization can be employed to reduce memory consumption and enhance decode speed. Mixed quantization involves using different levels of quantization precision for operators in different parts of the model, applying 4-bit quantization to operators with minimal impact on model accuracy, and 8-bit quantization to the parts with significant influence. This combination achieves a balance in accuracy, memory usage, and inference speed.

Selecting the optimal mixed quantization scheme is a combinatorial optimization problem. Given the unpredictable interactions between different operator combinations, this represents an NP-complete problem. For instance, the Qwen2.5-0.5B model has 169 quantization partitions based on operators, resulting in a search space of , which is incredibly large. For a selected quantization combination, datasets can be used to evaluate its accuracy. To achieve a precise assessment, the test data must be substantial, with each evaluation potentially time-consuming. Given the immense search space, exhaustive search is impractical, and heuristic algorithms are necessary.

To address the NP-complete problem, a **Regressive Genetic Algorithm** is used. First, two quantization methods are selected, and the one with higher precision serves as the base quantization method. The search aims to apply the lower-precision quantization method to certain Linear operators as extensively as possible while maintaining high accuracy.

1. **Quantize layers and types and calculate accuracy**: Start by using low-precision quantization for each layer and calculate the corresponding accuracy score. Then, do the same for each type of operator, calculating its accuracy score. These scores will guide the heuristic algorithm with probabilities in further steps. For layers or types with accuracy below P, set the guiding probability to 0 for pruning, reducing the search space size.
2. **Layer combination search**: Perform the search using a regressive genetic algorithm, where gene encoding length equals the total number of layers, with 0 indicating the use of high-precision quantization and 1 indicating low-precision. Limit the number of layers using low precision to be less than N and accuracy greater than P to ensure quantization precision. During gene mutation, use the normalized values from step 1 as mutation probabilities to guide the evolutionary direction and ultimately identify a combination of layers for low-precision quantization.
3. **Operator combination search**: Fix the low-precision layers identified in step 2 and use regressive genetic algorithms for other layers at the operator level. The gene encoding length is (remaining layers \* number of types), with the gene bit indicating method similar to step 2. Also, limit the total number of low-precision operators to be less than N and ensure accuracy greater than P. In the mutation phase of the genetic algorithm, use normalized combinations of the layer and type accuracy from step 1 as mutation probabilities to guide the evolution path, eventually finding a low-precision quantization operator combination.

Next, we present a result obtained for the DistilQwen2.5-0.5B-Instruct model using the aforementioned genetic algorithm search. This result utilizes a mixed quantization approach combining 4-bit and 8-bit per-channel quantization. By default, the entire model uses 8-bit per-channel quantization, with certain Linear operators utilizing 4-bit per-channel quantization. Firstly, in the initial step, 4-bit quantization was applied according to layer and type, resulting in scores shown in Table 3. To reduce the search space, parts with layer scores below 0.49 and type scores below 0.5 were pruned. Then, based on the scores in the table, a second genetic algorithm search was performed. The quantized layer combination obtained in this search includes: Lm head, and layers 21 and 22. Following these results, a third search was conducted, which identified gate\_proj for layers 8, 18, 19, 20, and 23. This mixed quantization method reduces model weight size by 18% compared to int8 quantization, enhances decode performance by 10%, with only a 0.08% decrease in average accuracy.

|  |  |  |
| --- | --- | --- |
|  | 4 bit quant | ceval score |
| Layer-wise | Lm head | 0.5170876671619614 |
| Layer 0 | 0.5 |
| Layer 1 | 0.5148588410104011 |
| Layer 2 | 0.48588410104011887 |
| Layer 3 | 0.5022288261515602 |
| Layer 4 | 0.5170876671619614 |
| Layer 5 | 0.5156017830609212 |
| Layer 6 | 0.5096582466567607 |
| Layer 7 | 0.5104011887072808 |
| Layer 8 | 0.512630014858841 |
| Layer 9 | 0.49925705794947994 |
| Layer 10 | 0.49777117384843983 |
| Layer 11 | 0.49108469539375926 |
| Layer 12 | 0.5104011887072808 |
| Layer 13 | 0.5104011887072808 |
| Layer 14 | 0.5089153046062407 |
| Layer 15 | 0.512630014858841 |
| Layer 16 | 0.5148588410104011 |
| Layer 17 | 0.5044576523031203 |
| Layer 18 | 0.5148588410104011 |
| Layer 19 | 0.5096582466567607 |
| Layer 20 | 0.5148588410104011 |
| Layer 21 | 0.5141158989598811 |
| Layer 22 | 0.49777117384843983 |
| Layer 23 | 0.5118870728083209 |
| Type-wise | q\_proj | 0.49925705794947994 |
| k\_proj | 0.4985141158989599 |
| v\_proj | 0.4903417533432392 |
| o\_proj | 0.5 |
| gate\_proj | 0.5089153046062407 |
| up\_proj | 0.4725111441307578 |
| down\_proj | 0.40936106983655274 |

## AWQ Quantization

AWQ quantization is a simple and effective method that adjusts the weight distribution based on the distribution of activations to mitigate the accuracy loss caused by quantization. After determining a mixed quantization scheme, applying AWQ can further enhance quantization accuracy. In this context, we used a portion of the CMMLU dataset for calibration to perform AWQ quantization on the derived mixed quantization scheme. After implementing AWQ quantization, the average accuracy improved by 0.9%.

# Conclusion

We designed a W4/8A8 quantization computation method specifically for ARM CPUs and optimized matrix multiplication implementation using ARM instructions to achieve optimal performance. A heuristic algorithm was utilized to determine a mixed quantization scheme for the model, ultimately achieving a balanced performance in terms of accuracy, memory usage, and inference speed