

MNN-LLM: A Generic Inference Engine for Fast Large Language Model Deployment on Mobile Devices

Zhaode Wang

Alibaba Group

Beijing, China

zhaode.wzd@taobao.com

Jingbang Yang

Alibaba Group

Hangzhou, China

jingbang.yjb@taobao.com

Xinyu Qian

Alibaba Group

Hangzhou, China

qianxinyu.qxy@taobao.com

Shiwen Xing

Alibaba Group

Hangzhou, China

tianbu.xsw@taobao.com

Xiaotang Jiang

Alibaba Group

Hangzhou, China

xiaotang.jxt@taobao.com

Chengfei Lv*

Alibaba Group

Hangzhou, China

chengfei.lcf@taobao.com

Shengyu Zhang

Zhejiang University

Hangzhou, China

sy_zhang@zju.edu.cn

Abstract

Large language models (LLMs) have demonstrated exceptional performance across a variety of tasks. However, their substantial scale leads to significant computational resource consumption during inference, resulting in high costs. Consequently, edge device inference presents a promising solution. The primary challenges of edge inference include memory usage and inference speed. This paper introduces MNN-LLM, a framework specifically designed to accelerate the deployment of large language models on mobile devices. MNN-LLM addresses the runtime characteristics of LLMs through model quantization and DRAM-Flash hybrid storage, effectively reducing memory usage. It rearranges weights and inputs based on mobile CPU instruction sets and GPU characteristics while employing strategies such as multicore load balancing, mixed-precision floating-point operations, and geometric computations to enhance performance. Notably, MNN-LLM achieves up to a 8.6x speed increase compared to current mainstream LLM-specific frameworks.

1 Introduction

In recent years, large language models (LLMs) have rapidly evolved, becoming one of the most revolutionary technologies in the field of natural language processing. Prominent models like ChatGPT-4o [18] and QwenMax [7], due to their substantial parameter scales, are typically deployed in the cloud using GPU inference. However, the large parameter size and computational demands lead to high user costs, and utilizing cloud services may involve handling sensitive user data, raising privacy and security concerns. In response to these issues, the trend of deploying LLMs on mobile devices is gaining momentum [19]. While mobile devices are widely used, their memory and computational limitations make it challenging to run large-scale LLMs directly.

To address this challenge, many open-source pre-trained LLMs have released smaller-scale models tailored for edge environments. These models share the same architecture as their cloud counterparts but have fewer parameters, such as Qwen2-1.5B [27]. According to the Scaling Law [16], smaller models exhibit limited capabilities and often can only perform simple or specific tasks. However, with the advent of models like phi-1 [11], it has been

shown that the quality of training data can significantly enhance the capabilities of smaller parameter models. Additionally, reinforcement learning algorithms in models like ChatGPT-01 [18] can further improve LLM performance without increasing the parameter count. As a result, the gap between smaller and larger models is gradually narrowing; for instance, the 3B parameter Qwen2.5-3B [24] achieved a score of 64 on the MMLU benchmark, surpassing many 30B parameter predecessors. This rapid enhancement in LLM capabilities offers greater feasibility for deploying LLMs on mobile devices.

Despite the reduction in parameter scale, the computational demands of LLMs remain substantial compared to traditional computer vision models used for edge inference. Running LLMs smoothly on edge devices poses significant challenges due to hardware, memory, and computational constraints. To address this issue, several frameworks for deploying LLMs on mobile devices have emerged. Some frameworks, such as PowerInfer-2 [26] and LLM in Flash [1], require modifications to the model for compatibility. Others, like llama.cpp [10], MLC-LLM [23], and fastllm [29], can be used directly with a focus on LLMs.

This paper introduces MNN-LLM, a generic mobile inference framework that supports LLM inference as well as the deployment of various deep learning models. MNN-LLM is developed based on MNN [15], a general framework designed for executing deep learning model inference on mobile devices. It addresses the challenges posed by large-scale LLMs through targeted optimizations in model export, quantization, and computational graph optimization. Furthermore, MNN-LLM analyzes the inference process and employs various forms of combined quantization, utilizing DRAM-Flash hybrid storage to reduce runtime memory usage. By optimizing high-computation operators and rearranging data according to the characteristics of different hardware, MNN-LLM ensures optimal utilization of edge computing resources.

2 Background and Motivation

2.1 LLM Model and Inference

Currently, mainstream LLMs primarily adopt a Decoder-Only architecture, with the main parameters located in the Embedding and Linear operators. During inference, the operators that consume the most time are Linear and Attention.

*Chengfei Lv is the corresponding author.

The inference process can be divided into two phases: prefill and decode. The prefill phase refers to the computation of the input text, processing the user’s input text sequence to generate the first token. The decode phase involves generating text, where each decode operation produces one token until a termination token is generated. These two phases exhibit different computational characteristics; specifically, the prefill phase tends to be computation-bound, while the decode phase is memory-bound due to the typical computational throughput and memory bandwidth of edge devices.

In the inference phase of the Attention mechanism, there are three inputs: query, key, and value. During the decode phase, only the query generated from the current input token is needed; however, all previously computed keys and values are required. To reduce computational load and avoid redundant calculations, a key-value (KV) cache is typically employed to store the keys and values from prior computations.

2.2 Mobile Devices Analysis

Mobile devices often have limited memory, and the large parameter sizes of LLMs can lead to significant memory usage, especially as context length increases, resulting in memory shortages that may terminate processes. Although Flash memory has much slower read speeds than DRAM, its higher capacity makes it crucial for LLM inference.

Mobile devices utilize CPUs and GPUs, typically featuring multiple cores and often following a big.LITTLE [2] architecture, necessitating concurrent optimization during CPU development. Variations in instruction sets across CPUs require tailored optimizations for optimal performance. General-purpose computing on mobile GPUs typically employs Vulkan and OpenCL standards, allowing developers to parallelize tasks using these APIs. Hardware drivers manage instruction dispatch and task scheduling, enhancing code portability across platforms.

3 MNN-LLM Overview

MNN-LLM, built on the deep learning framework MNN, leverages MNN’s extensive operator set and model supported versatility to enhance flexibility and adaptability. Unlike LLM-specific inference engines, MNN-LLM supports a wider range of models, boosting usability and developer friendliness in edge computing contexts.

MNN provides robust support for computer vision (CV) models like MobileNet [13] and YOLO [22], which are typically exported to ONNX [8] before conversion to MNN format. While MNN-LLM uses these export and conversion processes, the large parameter sizes of LLMs can result in high memory usage and longer conversion times. To mitigate this, optimizations have been introduced: Linear operators are replaced with custom operators during graph export, allowing ONNX export to focus on the computation graph without parameters. After model export, conversion, and optimization, parameters can be handled separately, streamlining the process and leveraging MNN’s model format. Additionally, during model conversion, optimizations such as RMSNorm [28] fusion and Attention fusion are applied. The computation graph also supports the runtime loading of LoRA [14] weights, enabling seamless integration of LoRA models without requiring external implementations in the inference framework.

MNN-LLM provides robust quantization capabilities, supporting both integer (int) and floating-point (fp) quantization during the model conversion phase, as well as quantization of activation values and KV cache during runtime. Additionally, it supports other quantization algorithms, such as GPTQ[9], and allows for the import of quantized weights.

MNN-LLM performs extensive optimizations for memory and computation at runtime. To address the significant memory usage of LLMs, it employs methods such as DRAM-Flash hybrid storage and combined quantization. For the high computational load, strategies like data reorder tailored for hardware, multicore load balancing, mixed-precision floating-point operations, and geometry computations are utilized. Additionally, specific optimizations are implemented for multi-LoRA scenarios.

4 Memory Optimization

4.1 DRAM-Flash Hybrid Storage

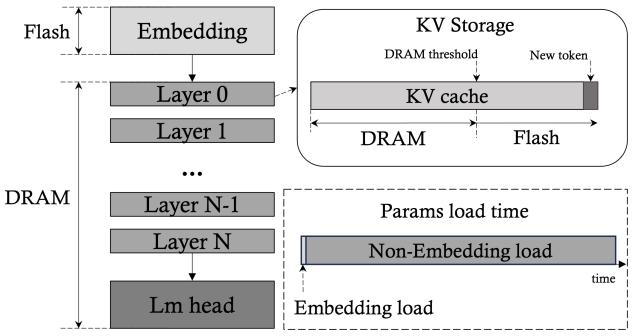


Figure 1: DRAM-Flash Hybrid Storage for LLM model parameters and KV cache.

The primary bottleneck for deploying large LLM models on mobile devices lies in the limitations of DRAM. MNN-LLM employs a DRAM-Flash hybrid storage strategy to mitigate memory usage, ensuring minimal memory occupancy while maintaining the usability of LLM inference under constrained memory conditions. Although Flash storage has a larger capacity than DRAM, its read speeds are significantly slower; for instance, LPDDR5X achieves approximately 58 GB/s, while UFS 4.0 ranges from about 450 MB/s to 3 GB/s [26]. This means that DRAM can be 19 to 130 times faster than Flash. While hybrid storage can reduce memory demands and enhance usability, it may compromise inference performance. As shown in Figure 1, MNN-LLM’s hybrid storage strategy is tailored to the operational characteristics of the model: for parameter storage, it assesses utilization rates and allocates low-utilization parameters to Flash to minimize speed impact. For the KV data, prefetching techniques are employed to reduce the latency of Flash reads, thereby mitigating their effect on performance.

The large parameter scale of LLM models is a primary reason for their high memory consumption. Structurally, the parameters can be divided into three categories: *Embedding*, *Layer*, and *Lm head*. The size of the *Embedding* and *Lm head* parameters is generally calculated as *vocabulary size* \times *hidden size*, and since the *vocabulary size* is usually large, the *Embedding* parameters do not

Table 1: Qwen2 7B Model Params

Configuration	Size	Params	Size
vocabulary size	151646	Embedding	1.09 B
hidden size	3584	Layers	4.89 B
intermediate size	18944	Lm head	1.09 B
layers	28	Total	7.07 B

participate in calculations like other parameters do. *Layer* refer to the parameters in each Decoder Layer, including the Attention and MLP Linear layers, typically sized at *hidden size* × *hidden size* or *intermediate size* × *hidden size* with consistent parameter scales across layers. As shown in Table 1, in the Qwen2 7B [27] model, the non-computational *Embedding* parameters account for about 15% of the total parameters.

In the decode phase, each input consists of the previously generated token. Leading to a computational process that necessitates loading 1/vocabulary size of the *Embedding* parameters, along with full *Layer* and *Lm head* parameters. Thus, *Layer* and *Lm head* parameters should be prioritized for DRAM storage, while the *Embedding* parameters can be stored in Flash. Taking Qwen2 7B as an example, with Embedding data read in bfloat16 format, the decode phase only requires the Embedding value for one token, resulting in a data size of 7 KB for each decode. The UFS 4.0 read speed is approximately 15 μ s slower than LPDDR5X. In contrast, loading non-Embedding parameters from memory takes about 103 ms. In typical mobile devices, the compute characteristics during the decode phase are Memory Bound, making the memory access time roughly equivalent to the parameter access time. Therefore, storing *Embedding* parameters in Flash adds only about 1.4 % to the total inference time. Consequently, utilizing Flash for storing Embedding layers allows for a 15% reduction in DRAM usage without significantly impacting inference performance. For example, Qwen-7B can reduce DRAM usage by approximately 2.18 GB when using bfloat16 storage, greatly enhancing the feasibility of model inference on memory-constrained mobile devices.

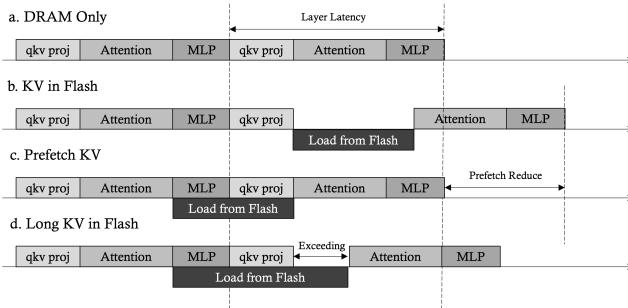


Figure 2: Comparison of KV loading times for DRAM, DRAM-Flash, Prefetching, and Exceeding.

In scenarios with long input texts or extensive generation lengths, the continuous growth of the KV cache can lead to significant memory usage. MNN-LLM addresses this challenge by employing a hybrid storage strategy, utilizing Flash to hold part of the KV

cache, thus ensuring LLM inference remains feasible under long-context conditions. Initially, all KV cache values are stored in DRAM, but as the context expands and the KV cache size increases, any portion exceeding a certain threshold is transferred to Flash. Since each computation produces only one set of new KV values, the total number of KV values for the Qwen2 7B model amounts to approximately 1 KB, minimizing storage overhead.

As the number of KV cache values stored in Flash rises, the time required to load them from Flash will gradually increase, which can slow down inference speed, as illustrated in Figure 2b. To mitigate the impact of KV cache loading from Flash on inference time, we implement prefetching: during the MLP phase of the current layer and the qkv projection phase of the next layer, KV cache values are prefetched from Flash into memory. When the prefetching time is less than or equal to the computation time, the LLM inference speed remains unaffected.

For instance, in the Qwen2 7B model, the parameter size for a single layer's qkv and MLP is 178.83 MB, and the decode phase is Memory Bound. Given that LPDDR5X incurs about 3 ms of loading time for this data, we assume a loading speed of 1 GB/s for Flash due to its larger continuous memory blocks, allowing approximately 3 MB of KV values to be loaded within the computation time. Therefore, when the length of the KV cache stored in Flash is under 3072 K, the overhead from Flash loading is effectively masked by the computation time, as shown in Figure 2c. However, once the length of the KV cache in Flash exceeds 3072 K, as depicted in Figure 2d, prefetching cannot completely offset the Flash loading overhead; each additional 1 K of length adds approximately 1 ms of delay. It is important to note that DRAM also holds a substantial length of KV cache, meaning that only in scenarios with exceedingly long contexts will the inference speed of the LLM be impacted. Nevertheless, storing KV cache in Flash ensures that LLM inference remains viable even with long contexts.

4.2 Combined Quantization

The large parameter size of LLM models is the primary reason for their high memory consumption, and quantization can significantly reduce the parameter size, thereby lowering memory usage. However, quantization can affect the model’s inference accuracy; generally, lower bit counts result in greater information loss and a larger impact on accuracy. There are various methods, data types, and bit counts for quantization, making it crucial to choose an appropriate method to balance memory usage, runtime performance, and model accuracy.

For the parameters of the Embedding, Layer, and Lm head, MNN-LLM employs a combination quantization strategy to balance accuracy and computational overhead. The weights of the embedding layer account for approximately 15% of the total model weight. Since only a small portion of these weights is utilized during each decoding step, they are stored in Flash memory, which does not occupy DRAM. This allows for the use of bfloat16 storage, ensuring computational accuracy. Non-embedding parameters, which include the weights of the layers and the LM head, must be fully loaded for each computation, making their size significantly impactful on inference performance. In particular, during the decoding

phase, which is memory-bound, the inference time is directly proportional to the size of these parameters. Therefore, it is crucial to use low-bit quantization for these weights. Taking both precision and hardware computation instructions into account—where edge CPUs are particularly friendly towards int8 computation—these parameters are quantized using int4 or int8. During calculations, activation values are quantized to int8, enabling the use of W4A8 or W8A8 computation methods on CPUs to leverage int8 instructions. On GPUs, W4A16 or W8A16 methods are used to take advantage of floating-point capabilities. To maintain model accuracy, all these parameters employ asymmetric quantization. Asymmetric quantization as below:

$$w_{asy} = \text{round} \left(\frac{\frac{w_{float} - w_{min}}{w_{max} - w_{min}}}{\frac{clip_{max} - clip_{min}}{clip_{max} + clip_{min}}} \right) + clip_{min} \quad (1)$$

Additionally, because the LM head has a greater impact on model accuracy than the layers, it is prioritized for int8 quantization to enhance overall precision.

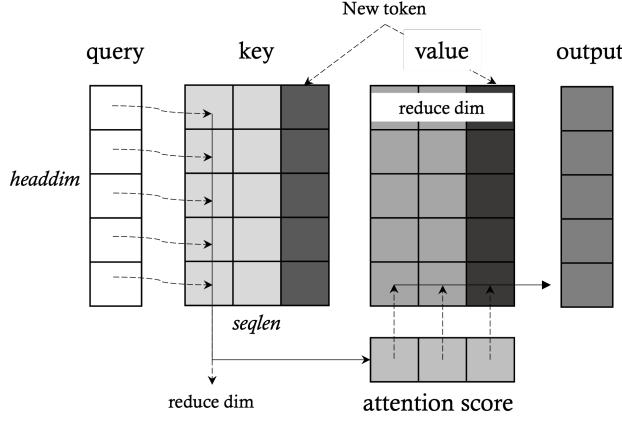


Figure 3: The reduction dimensions in the computation of Attention query, key, and value.

When dealing with long contexts, the memory usage of the KV cache continues to grow, and quantization strategies can effectively reduce this memory consumption. MNN-LLM provides different quantization methods for keys and values based on their computational roles. During attention calculations, the shapes for query, key, and value are $[headdim, seqlen, headdim]$. When performing matrix multiplication between key and query, the dimension being reduced is *headdim*, which is a fixed value. Therefore, int4/int8 quantization can be applied to keys, allowing new key values to be quantized and stored directly. In contrast, during the matrix multiplication of *attention score* with values, the dimension being reduced is *seqlen*. Using int4/int8 quantization for values can affect the data distribution of existing values when new ones are added, necessitating updates to their quantization values and incurring additional overhead. To address this, MNN-LLM employs fp8 quantization for values, allowing new values to be quantized directly without impacting the existing ones.

Table 2: Tile Sizes for Different CPU Architectures

Architecture	e_p	h_p	l_p
ARM i8sdot	12	8	4
ARM i8mm	10	8	8
X86 AVX2	4	8	4
X86 AVX512	4	64	4

5 Compute Optimized

5.1 Hardware-Driven Data Reorder

Analysis of the inference process in LLMs shows that the primary time-consuming operations are Linear and Attention, both of which fundamentally rely on matrix multiplication. Therefore, optimizing matrix multiplication for these two operators is crucial for improving LLM performance. Loop Tiling is a common optimization technique that enhances memory access locality, significantly impacting performance. The optimal tile size for Loop Tiling [12] greatly affects the final matrix multiplication performance and is influenced by the device’s memory, cache, and computational hardware. Thus, it is essential to select the most suitable data reorganization and computation method based on hardware and data scale to achieve peak performance. MNN-LLM employs a Hardware-Driven data reorder strategy tailored to the computational characteristics of these two operator types to determine the tiling method and size, optimizing LLM inference performance.

The matrix multiplication for the Linear operator involves the activation values and weight values, where the activation values are computed during inference and the weights are determined when the model is loaded. In MNN-LLM, weights are generally quantized to int4 or int8. Assuming the activation value matrix size is $[e, l]$, and the weight size is $[h, l]$, the resulting size will be $[e, h]$. After data tiling on the mobile CPU, the input matrices are rearranged as: $\left[\frac{e}{e_p}, \frac{l}{l_p}, e_p, l_p \right]$ for the activation values and $\left[\frac{h}{h_p}, \frac{l}{l_p}, e_p, l_p \right]$ for the weights. This tiling allows for value reuse within the registers during kernel computations, enhancing memory locality and reducing memory access frequency. The memory access count is optimized from $2ehl + eh$ to $\frac{e}{e_p} \frac{h}{h_p} (lep + lh_p + h_p e_p)$. By using memory access frequency as the optimization objective and hardware parameters as constraints, we can compute the values for e_p, h_p, l_p under different hardware conditions. Let R be the number of vector registers, and $instruction_width$ be the data size computed in a single instruction along the l-dimension, e_p, h_p, l_p as given by the following formulas:

$$\min \quad \frac{e}{e_p} \frac{h}{h_p} (lep + lh_p + h_p e_p) \quad (2)$$

$$\text{s.t. } e_p + h_p + h_p e_p \leq R \quad (3)$$

$$l_p = instruction_width \quad (4)$$

Based on the above strategy, the block sizes calculated for various CPU instruction sets are shown in Table 2. By employing a Hardware-Driven data rearrangement strategy tailored to different CPU architectures, MNN-LLM can better utilize CPU computational power. For instance, the throughput of the *smla* instruction on ARM i8mm [5] is twice that of *sdot* [4]. When MNN-LLM detects that the CPU supports i8mm, it rearranges the weights with

$l_p = 8$ during the model loading phase. This arrangement format enhances performance compared to the data layout in llama.cpp, thereby improving the efficiency of the prefill stage.

GPUs support hardware loading/storing merging, which allows them to combine a certain number of memory access instructions if the memory addresses accessed by consecutive work items are contiguous. This capability minimizes the number of memory access instructions. Additionally, GPUs can load/store up to 128 bits of data at a time. To maximize memory loading efficiency, each work item should utilize vectorized loading/storing functions. In OpenCL, GPU memory objects are categorized into Buffers and Images. Images can automatically handle boundaries and return appropriate out-of-bounds values based on settings. Certain devices, such as Qualcomm’s Adreno GPUs [20], possess powerful texture engines and dedicated L1 caches, enabling efficient loading of data from Image objects. Compared to ordinary buffer objects, Images offer higher bandwidth, making them the preferred choice for storage. To leverage these memory loading advantages, MNN-LLM rearranges GPU weight data and uses Image objects for storage. The rearranged data structure is $[\frac{l}{l_p}, h, l_p]$ with $l_p = 32$. Each work item loads 4-bit weights at once, totaling 128 bits, which meets the GPU’s maximum loading bandwidth and corresponds to the size of four floating-point values in the CL_RGBA Image memory object. Additionally, each work item accesses data contiguously along the h dimension, ensuring continuous memory reads between work items. Finally, the runtime dynamically adjusts the parallelism based on actual dimensions, allocating a reasonable number of computational tasks to each work item.

For the Attention operator, a similar rearrangement strategy as used for Linear is applied. The key and value are stored directly in the rearranged data layout, ensuring that there is no need to rearrange the historical KV during each computation.

5.2 Multicore Workload Balancing

Modern CPUs typically have multiple cores, so effectively utilizing multicore computing capabilities is crucial when optimizing performance. MNN-LLM leverages the multicore parallelism of CPUs to parallelize operations along the $seqlen$ and $\frac{h}{h_p}$ dimensions. Considering the big.LITTLE [2] architecture of mobile CPUs, MNN-LLM specifies the computing load for different cores at startup based on their actual computational capabilities. During parallel computation, MNN-LLM allocates the computational workload according to the load rates of the cores. This balancing workload distribution strategy, can enhance multithreaded computing performance compared to the uniform workload strategy.

Mainstream mobile SoCs typically feature one prime core and three performance cores, such as the Snapdragon 8 Gen 3 [21]. High-load computations generally utilize the prime core and performance cores. When the number of threads exceeds one, parallel computing between the prime core and performance cores occurs, as shown in Figure 4. In this scenario, workload balancing significantly improves the multithreaded speedup compared to uniform workload distribution.

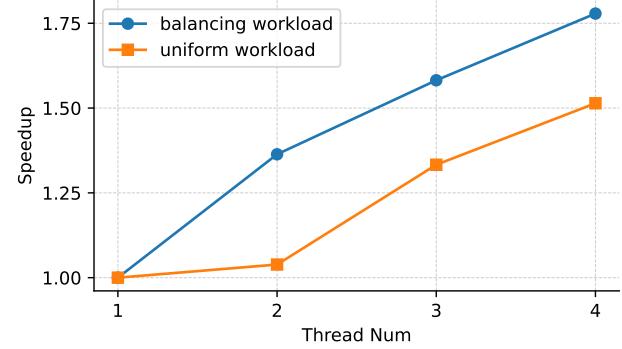


Figure 4: Parallel computing between 1 prime cores and 3 performance cores, speedup achieved through balancing workload and uniform workload.

5.3 Mixed Float Precision

In the previous discussion on matrix operations, low-bit quantization methods were employed to accelerate computations. For non-matrix multiplication operations, MNN-LLM also supports mixed precision for results, ensuring accuracy while enhancing inference performance. ARMv8.2 [6] and newer CPUs support float16 calculations, which can save half the memory compared to float32, and the throughput of float16 NOEN [3] instructions is twice that of float32. However, float16 has some precision limitations; for calculations requiring a higher precision range, significant errors may occur, especially when values exceed 65,504. To address this, MNN-LLM adopts a mixed precision strategy to maintain inference accuracy. During LLM inference, the Softmax calculation in Attention is particularly sensitive to data precision, so MNN-LLM ensures that *Softmax* uses float32. In the matrix multiplication of query and key, the query values may be large, potentially causing overflow after accumulation. To mitigate this, the division by $\sqrt{d_k}$ [25] can be applied directly to the query, reducing its value range and preventing overflow in the final result. This approach optimizes overall memory usage and inference performance while maintaining accuracy.

5.4 Geometry Compute

The computation graph of LLMs also includes long-tail operators such as *Transpose*, *Gather*, and *Concat*. Although these operators may not significantly contribute to overall execution time, they can result in substantial memory access when data sizes are large. To address these long-tail operators, MNN-LLM employs geometric computation [17] methods, abstracting all data rearrangement operations as linear mappings of addresses.

$$f(\vec{x}) = \vec{offset} + \vec{stride}\vec{x} \quad (5)$$

By taking the \vec{offset} and \vec{stride} with a length of 3, we can construct a fundamental mapping relationship described as a Region. This allows us to represent any data rearrangement operators using one or more Regions.

Table 3: Computation and Memory under Different LoRA Computation Orders.

Type	$(LoRA_A \cdot LoRA_B) \cdot A$	$LoRA_A \cdot (LoRA_B \cdot A)$
Computation	$rh^2 + h^3$	$2rh^2$
Memory	$2(rh^2 + h^2 + h^3)$	$4rh^2 + hh + rh$

For consecutive data rearrangement operators in the computation graph, this abstraction generates numerous contiguous Regions. MNN-LLM implements an automatic Region Fusion algorithm based on rules like *loop unrolling*, *loop interchange*, *loop tiling*, and *loop fusion*. This algorithm can automatically merge compatible Regions, thereby reducing the number of read and write operations for data rearrangement operators and enhancing performance. By utilizing geometric computation for LLM model inference, the overhead of long-tail operators can be reduced, improving performance by approximately 3%.

5.5 LoRA Optimization

On mobile devices, different tasks may require different LLM models. Due to the large number of model parameters, directly utilizing multiple models can lead to excessive bandwidth and storage usage. Thus, using a base model in conjunction with multiple LoRA models is a more efficient solution for multitasking.

MNN-LLM supports the deployment of merged LoRA models and the online loading of multiple LoRA models. When employing multiple LoRA models, MNN-LLM first loads the base model, followed by the computation graph and weights of the LoRA models, with LoRA models sharing the weights of the base model. Given that LoRA weights are generally small, the memory overhead is minimal. Online loading of LoRA models is more flexible than pre-merged approaches, making it suitable for multitasking scenarios, although it incurs additional computational costs. The computation graph for LoRA adds a bypass for the layers involving LoRA, transforming the original computation $A' = W \cdot A$ into $A' = W \cdot A + (LoRA_A \cdot LoRA_B) \cdot A$, where the additional computations may slow down model inference. Analyzing the characteristics of LoRA weights reveals that the size of R is relatively small compared to the original parameters, allowing us to leverage the associative property of matrix multiplication to alter the computation order, transforming $(LoRA_A \cdot LoRA_B) \cdot A$ into $LoRA_A \cdot (LoRA_B \cdot A)$. Assuming the size of matrix A is $[h, h]$ and that of $LoRA_A$ and $LoRA_B$ is $[h, r]$, the computation memory access before and after optimization is shown in Table 3. Given that R is relatively small [14], rearranging the computation order significantly reduces the memory access volume. For instance, using Qwen2 7B as an example, if $h = 3584$ and $r = 8$, the optimized memory access volume is only 0.5% of the original, thereby effectively improving computational efficiency.

6 Evaluation

The experimental design hinges upon quantized models, namely Qwen2 1.5B, Qwen2 7B, and Llama3 8B, utilizing the Xiaomi 14 as the test apparatus. Comparative evaluations of inference efficacy are conducted across CPU (harnessing 4 threads) and GPU (via OpenCL)

architectures, employing inference engines such as llama.cpp, MLC-LLM, and fastllm. Given that MLC-LLM does not accommodate CPU-based inference and fastllm lacks GPU compatibility, pertinent experiments are excluded for these engines. Extensive trials were executed with prompts of varying lengths (64, 256, and 1024 tokens), with a restrictive upper limit of 16 tokens imposed on the decoding phase.

Owing to the poor performances of MLC-LLM in handling asymmetric quantization models, the reported results of MLC-LLM are based on symmetric quantized models but competing engines were explicitly engaged in inference tasks using asymmetric models. The performance results are reflected in terms of the prefill and decode speed, graphically represented in Figure 5.

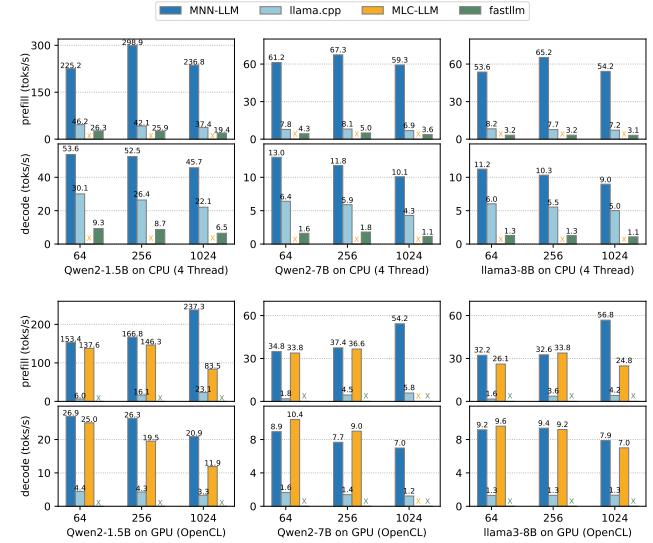


Figure 5: Prefill and decode speeds of MNN-LLM, llama.cpp, MLC-LLM, and fastllm under different prompt lengths on Xiaomi14’s CPUs and GPUs.

In CPU benchmarking, MNN-LLM excels, achieving prefill speed boosts of 8.6x over llama.cpp and 20.5x over fastllm, complemented by decoding speeds that are 2.3x and 8.9x faster, respectively. In GPU-based assessments, MNN-LLM’s performance slightly declines compared to MLC-LLM, particularly when using Qwen2-7B with shorter prompts, due to MLC-LLM’s advantageous symmetric quantization technique. MNN-LLM excels, achieving up to 25.3x faster prefill and 7.1x faster decoding than llama.cpp, and 2.8x and 1.7x improvements over MLC-LLM, respectively.

7 Conclusion

This paper introduces MNN-LLM, a high-performance general-purpose inference framework tailored for LLM inference on mobile devices. The framework enhances memory usage through DRAM-Flash Hybrid Storage and Combined Quantization, while improving inference speed with Hardware-Driven Data Reordering, Multicore Workload Balancing, Mixed Float Precision, and Geometry Compute. When compared to leading mainstream frameworks, MNN-LLM achieves up to an 8.6x performance improvement.

References

- [1] Keivan Alizadeh, Iman Mirzadeh, Dmitry Belenko, Karen Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. 2024. LLM in a flash: Efficient Large Language Model Inference with Limited Memory. arXiv:2312.11514 [cs.CL] <https://arxiv.org/abs/2312.11514>
- [2] ARM. 2024. Arm Big.LITTLE. <https://www.arm.com/zh-TW/technologies/big-little>.
- [3] ARM. 2024. Arm NEON. <https://www.arm.com/technologies/neon>.
- [4] ARM. 2024. Dot Product. <https://developer.arm.com/documentation/100069/0609/A64-SIMD-Vector-Instructions/SDOT-vector->.
- [5] ARM. 2024. Matrix Multiplication extension. <https://developer.arm.com/documentation/101754/0622/armclang-Reference/Other-Compiler-specific-Features/Supported-architecture-features/Matrix-Multiplication-extension>.
- [6] ARM. 2024. The Armv8.2 architecture extension. <https://developer.arm.com/documentation/109697/latest/Feature-descriptions/The-Armv8-2-architecture-extension>.
- [7] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen Technical Report. *arXiv preprint arXiv:2309.16609* (2023).
- [8] Junjie Bai, Fang Lu, Ke Zhang, et al. 2019. ONNX: Open Neural Network Exchange. <https://github.com/onnx/onnx>.
- [9] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2023. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. arXiv:2210.17323 [cs.LG] <https://arxiv.org/abs/2210.17323>
- [10] Georgi Gerganov. 2024. ggerganov/llama.cpp: Port of Facebook's LLaMA model in C/C++. <https://github.com/ggerganov/llama.cpp>.
- [11] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. 2023. Textbooks Are All You Need. (2023). arXiv:2306.11644 [cs.CL] <https://arxiv.org/abs/2306.11644>
- [12] Emna Hammami and Yosr Slama. 2017. An Overview on Loop Tiling Techniques for Code Generation. In *2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA)*. 280–287. <https://doi.org/10.1109/AICCSA.2017.168>
- [13] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv:1704.04861 [cs.CV] <https://arxiv.org/abs/1704.04861>
- [14] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. arXiv:2106.09685 [cs.CL] <https://arxiv.org/abs/2106.09685>
- [15] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, Chengfei Lv, and Zhihua Wu. 2020. MNN: A Universal and Efficient Inference Engine. *CoRR* abs/2002.12418 (2020). arXiv:2002.12418 <https://arxiv.org/abs/2002.12418>
- [16] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alex Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. *CoRR* abs/2001.08361 (2020). arXiv:2001.08361 <https://arxiv.org/abs/2001.08361>
- [17] Chengfei Lv, Chaoyue Niu, Renjie Gu, Xiaotang Jiang, Zhaode Wang, Bin Liu, Ziqi Wu, Qiulin Yao, Congyu Huang, Panos Huang, Tao Huang, Hui Shu, Jinde Song, Bin Zou, Peng Lan, Guohuan Xu, Fei Wu, Shaojie Tang, Fan Wu, and Guihai Chen. 2022. Walle: An End-to-End, General-Purpose, and Large-Scale Production System for Device-Cloud Collaborative Machine Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 249–265. <https://www.usenix.org/conference/osdi22/presentation/lv>
- [18] OpenAI. 2023. ChatGPT. <https://openai.com/chatgpt> Available at: <https://openai.com/chatgpt>.
- [19] Qualcomm. 2023. The future of AI is “on device”. https://cms.tinyml.org/wp-content/uploads/ew2023/Kyuwooong-Hwang_tinyML-Asia-2023.pdf.
- [20] qualcomm. 2024. Adreno Graphics Processing Units. <https://www.qualcomm.com/products/features/adreno>.
- [21] qualcomm. 2024. Snapdragon 8 Gen 3 Mobile Platform. <https://www.qualcomm.com/products/mobile/snapdragon/smartphones/snapdragon-8-series-mobile-platforms/snapdragon-8-gen-3-mobile-platform>.
- [22] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhad. 2016. You Only Look Once: Unified, Real-Time Object Detection. arXiv:1506.02640 [cs.CV] <https://arxiv.org/abs/1506.02640>
- [23] MLC team. 2024. MLC-LLM. <https://github.com/mlc-ai/mlc-llm>.
- [24] Qwen Team. 2024. Qwen2.5: A Party of Foundation Models. <https://qwenlm.github.io/blog/qwen2.5/>
- [25] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. arXiv:1706.03762 [cs.CL] <https://arxiv.org/abs/1706.03762>
- [26] Zhenliang Xue, Yixin Song, Zeyu Mi, Le Chen, Yubin Xia, and Haibo Chen. 2024. PowerInfer-2: Fast Large Language Model Inference on a Smartphone. arXiv:2406.06282 [cs.LG] <https://arxiv.org/abs/2406.06282>
- [27] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zhihao Fan. 2024. Qwen2 Technical Report. *arXiv preprint arXiv:2407.10671* (2024).
- [28] Biao Zhang and Rico Sennrich. 2019. Root Mean Square Layer Normalization. arXiv:1910.07467 [cs.LG] <https://arxiv.org/abs/1910.07467>
- [29] ztxz16. 2023. fastllm. <https://github.com/ztxz16/fastllm>.