Blazingly Fast LLM Inference Framework on Arm Devices

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

Hui Ni
Tencent
Shanghai, China
nihuini@tencent.com

Xiao Zhang
Cambricon
Hefei, China
zhangxiao1@cambricon.com

Song Lin
Kunlun-inc
Beijing, China
song.lin@kunlun-inc.com

Jin Yao
Tsingmicro
Nanjing, 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. It introduces various optimization strategies, including model finetuning, Quantization Aware Training (QAT), the use of the ONNX format, constant folding, and operator fusion. Finetuning and quantization training significantly enhance model performance across various datasets. Additionally, the paper discusses methods for optimizing neural network models using ONNX, including shape inference and operator fusion, which significantly improve model efficiency and hardware utilization. Practical applications on ARM platforms demonstrate that these optimization measures effectively reduce computational resource consumption and enhance inference speed. Lastly, the paper highlights detailed operator optimizations, such as loop tiling and the use of high-throughput computing instructions, further boosting model performance.**

Keywords—Large Language Models (LLM), Model Optimization, Quantization Training, ONNX, Operator Fusion

I. 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.

II. FINETUNING AND QUANTIZATION

To improve accuracy of LLM, Finetuning and Quantization Aware Training (QAT) is employed. Testing revealed that finetuning and QAT could improve the accuracy of the piqa dataset by 1.9%. More detailed results showed in TABLE 1.

Finetuning utilized the method recommended by QWen. The model was finetuned on the datasets of piqa, arc_challenge, and hellaswag. The data generation for finetuning referred to the usage of lm-eval, resulting in a total of 57,137 generated data entries. Finetuning was conducted using the deepspeed zero3 configuration, with adjustments made to parameters such as max_length and batch_size.

After finetuning, the QWen 1.8B model showed significant improvements on the datasets. The model was then quantized using GPTQ, and the GPTQ results still exceeded those of the original bf16 model.

To adapt to the quantized inference on ARM platforms, we conducted Quantization Aware Training on the model using BitDistiller. BitDistiller used the original model as the teacher and the quantized model as the student, training the student model with data produced by the teacher model.

BitDistiller Algorithm adopts the asymmetric clipping to alleviate outliers in w, prior to the training loop. When training, BitDistiller forwards the model with the quantized weights, computes the loss with the CAKLD objective, and updates the full-precision weights.

TABLE I. ACCURACY OF MODEL

Model	**DataSet**		
	piqa	arc_challenge	hellaswag
QWen 1.8B	0.7280	0.3200	0.4543
FineTuned	0.7514	0.3413	0.5046
FineTuned+GPTQ	0.7476	0.3370	0.4898
Pseudo Quantization	0.7242	0.3234	0.4779
QAT+ Pseudo Quantization	0.7432	0.3294	0.4700

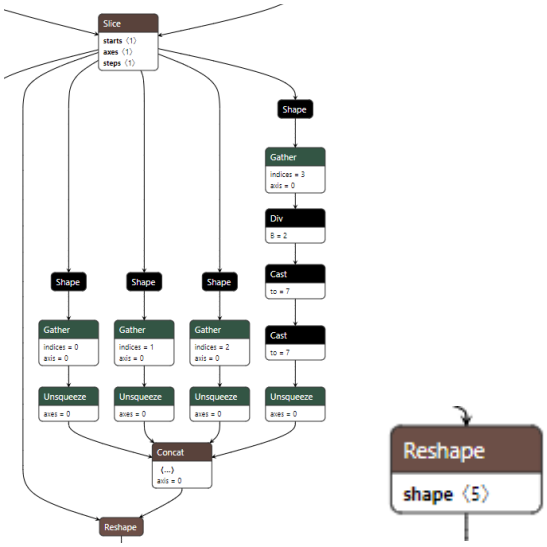
III. MODEL OPTIMIZATION

A. model analysis

ONNX (Open Neural Network Exchange) is an open format for representing deep learning models, designed to facilitate seamless model interchangeability across different deep learning frameworks. The design philosophy of ONNX is simple and generic, defining an intermediate representation format that describes the structure and parameters of neural network models. This intermediate representation format is graph-based, containing the hierarchical structure of the model and the connections between its components. ONNX supports various mainstream deep learning frameworks such as PyTorch, TensorFlow, MXNet, etc., allowing developers to freely migrate models between different frameworks without worrying about compatibility issues. However, it is precisely because of compatibility that ONNX graphs can be excessively redundant. For example, the implementation of the LayerNorm operator varies among different frameworks. To consider compatibility, each framework will convert its respective LayerNorm implementation into a series of small operators within ONNX. However, LayerNorm is typically treated as a single operator in inference frameworks, necessitating a tool to perform IR (Intermediate Representation) level operator fusion, redundancy elimination, and more optimization techniques.

B. Shape inference

Shape inference refers to the process of deducing the output shape of an operator based on the input shape. There are two main approaches to handle this. One approach is to generate random data based on the input shape, run the operator using ONNX Runtime, and obtain the output shape. Another approach is to directly calculate the output shape based on prior knowledge. Once the output shape of the operator is inferred, specific optimizations can be performed based on the obtained shape. The left image below illustrates this process.



When the output shape of the reshape operator is inferred as (1, dynamic_dim, 16, 2, 64), all the computations on the right side of the reshape operator can be removed. Instead, a constant node can be used to replace them, with the value (1, -1, 16, 2, 64). The optimization result is illustrated in the upper right image.

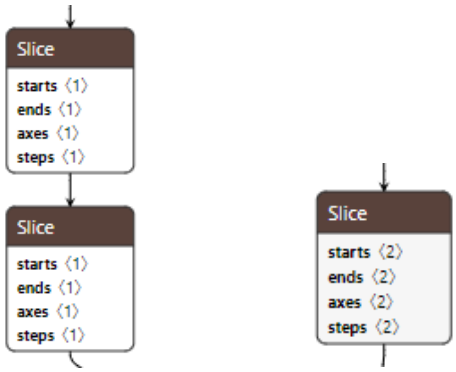
C. Constant folding

Constant folding in ONNX refers to the process of simplifying a computational graph by evaluating constant nodes and replacing them with their resulting values. This optimization technique is commonly used to improve the efficiency of inference by reducing the computational overhead associated with unnecessary constant calculations. When constant folding is applied, the ONNX graph is traversed, and any subgraphs containing sequences of operations with constant inputs are identified. These subgraphs are then evaluated, and the constant nodes are replaced with their computed values. This process continues recursively until no further constant folding can be performed. Constant folding can significantly improve the performance of inference, especially when dealing with models containing a large number of constant operations or redundant computations. By precomputing constant values during graph optimization, the computational overhead during runtime inference can be reduced, leading to faster execution times and improved efficiency.

when all the output shapes of shape operators are constant, we can perform inference on the model once using ONNX Runtime. Then, the output of the concat operator can be turned into a constant node, serving as the second input to the reshape operator. This allows us to remove the subnetwork from the shape operators to the concat operator.

D. Operator fusion

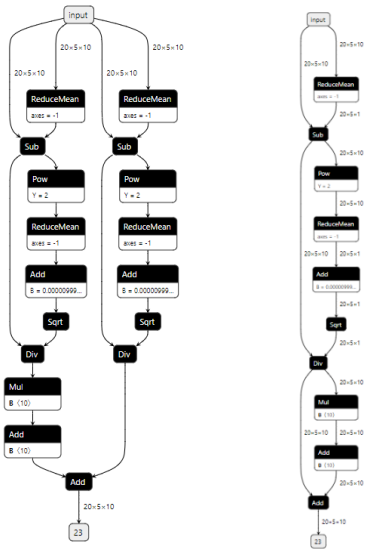
Operator fusion is a powerful optimization technique that combines multiple operators into a single operator. This consolidation reduces memory usage, computational overhead, and the number of memory accesses, thereby improving hardware utilization and reducing inference latency. By fusing compatible operators, such as consecutive element-wise operations or convolutions followed by activation functions, redundant memory transfers and computations can be avoided. This not only optimizes the execution flow but also enhances cache efficiency, leading to faster inference times and better utilization of available hardware resources.



When two cascading Slice operators meet certain conditions (such as having different axes for slicing), they can be fused into a single operator.

E. *Common subexpression elmination*

Common Subexpression Elimination (CSE) is a powerful optimization technique commonly employed in compilers to improve the efficiency of code execution. In many onnx models, certain operators are computed multiple times within a given scope, even though their results remain constant across these computations. Common subexpressions refer to these redundant expressions. CSE identifies such common subexpressions and replaces subsequent occurrences with references to the original computation result. This process effectively reduces the number of computations required during program execution.

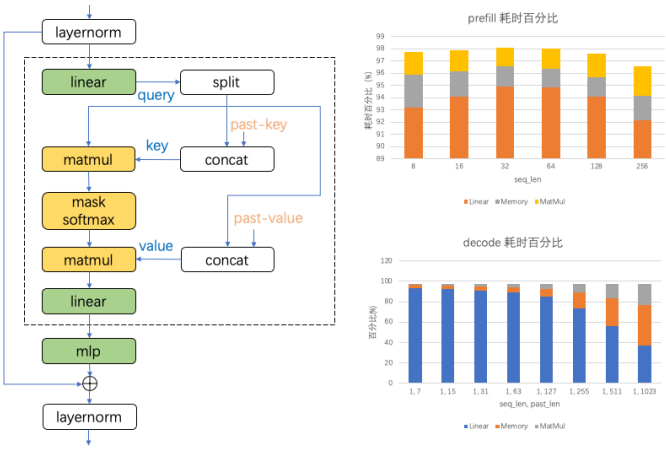


there are two ReduceMean operators with identical inputs and attributes. This means they can be merged into a single operation using a common subgraph algorithm. This process can continue incrementally until all Div operators can be merged into one.

IV. PERFORMANCE OPTIMIZATION

A. *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).

B. *Optimizing Linear Operators*

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 W4A8 quantization scheme, quantizing the model's weights (W) to 4 bits and the activations (A—i.e., the model's inputs and outputs) 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

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

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.

C. Quantization

In today's popular Large Language Models (LLM), the number of weights in linear layers has reached an astonishing scale, often comprising billions of parameters. For example, a model with 7 billion parameters (7b), even when stored using 16-bit floating-point (fp16) numbers, requires approximately 14GB of memory space. Deploying such models on memory-constrained mobile devices poses a significant challenge. To address this issue, we must take measures to compress these weights to occupy less memory. Quantization offers a viable path. Fortunately, the differences between the weights of linear layers in LLM are relatively small, making them highly suitable for low-bit quantization—that is, representing each weight value with fewer bits. Even after quantization, the computational results can maintain high fidelity with the original floating-point calculations, indicating minimal impact on model performance. As the GEMV operations during the decode phase are constrained by memory bandwidth, the theoretical performance of 4-bit quantization is twice that of 8-bit quantization. Additionally, experimental results show that the precision loss from 4-bit quantization compared to 8-bit is not substantial. Therefore, we have chosen to proceed with 4-bit quantization as our solution. With this approach, the 7b model now only requires approximately 3.5GB of memory to operate, meaning devices with 8GB of memory can run such a large model. We adopted an asymmetric quantization scheme, and a considerable advantage of this approach is that it can reduce the memory access for the model's weights by four times during computation, thereby effectively improving memory access performance.

While quantizing the weights, we also reviewed how the model inputs are handled. In the past, we utilized mixed-precision computing, which combined 4-bit quantized weights with 16-bit or 32-bit floating-point inputs. This required us to dequantize the quantized weights back into floating-point numbers before computation, load the floating-point input data from memory, and then perform floating-point multiply-accumulate operations. This method demands more memory access when processing inputs and, since the weights are stored in a row-major order, further increases the memory access volume. Additionally, since it involves floating-point computation, it uses floating-point SIMD instructions, which have lower peak performance than integer instructions.

To address these issues, we implemented a dynamic quantization scheme for inputs, specifically the W4A8 scheme. First, we need to account for the distribution of inputs at runtime, using the abs-max quant scheme to convert inputs into 8-bit integer numbers as the input to the linear layer. Then, we load the 4-bit quantized weights and convert them into 8-bit integers to execute the multiply-accumulate operations of matrix multiplication, using 32-bit integers to save the accumulation results. Finally, after the entire computation process, we dequantize these accumulated results back into floating-point numbers. Adopting the W4A8 technique not only significantly reduces the memory access volume but also allows us to utilize more powerful integer computing instructions, greatly enhancing the model's computational performance.

Adopting the W4A8 quantization strategy allows us to significantly improve the computational and memory access efficiency of the model while maintaining computational precision. In previous practices, schemes like W4A16 or W4A32 used floating-point operations, where the weights were stored as int4 and the input data was kept at 16-bit or 32-bit floating-point precision. On ARM architecture systems, floating-point calculations use the fmla instruction, while 8-bit integer calculations can utilize instructions such as sdot or smmla. Performance tests have shown that on ARM platforms, the peak performance of the smmla instruction is four times that of the fmla instruction. This significant boost reflects the computational gains that can be realized by optimizing instruction selection.In addition to enhancing computational performance, improved memory access efficiency is an essential component of model optimization. Indeed, alongside computational performance, memory access efficiency is a vital aspect of model optimization. The W4A8 quantization scheme not only enhances the computational throughput but also significantly reduces the memory access volume.

D. 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:

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

$$s.t.\ e_p h_p\ + e_p + h_p \leq R$$

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.

E. 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.

Below is an example kernel code for matrix multiplication using the smmla instruction on ARM:

```
LoopSz_TILE_8:
    // src    : 4 x [2 x 8] : v4-7
    // weight : 4 x [2 x 8] : v0-3
    // dst    : 4 x 4 x [4] : v16-31
    ld1 {v8.16b, v9.16b}, [x25], #32                 // weight
    ld1 {v4.16b, v5.16b, v6.16b, v7.16b}, [x24], x15 // src
    ushr v0.16b, v8.16b, #4
    and  v1.16b, v8.16b, v10.16b
    ushr v2.16b, v9.16b, #4
    and  v3.16b, v9.16b, v10.16b
    .inst 0x4e80a490 // smmla v16.4s, v4.16b, v0.16b
    .inst 0x4e81a491 // smmla v17.4s, v4.16b, v1.16b
    .inst 0x4e82a492 // smmla v18.4s, v4.16b, v2.16b
    .inst 0x4e83a493 // smmla v19.4s, v4.16b, v3.16b
    .inst 0x4e80a4b4 // smmla v20.4s, v5.16b, v0.16b
    .inst 0x4e81a4b5 // smmla v21.4s, v5.16b, v1.16b
    .inst 0x4e82a4b6 // smmla v22.4s, v5.16b, v2.16b
    .inst 0x4e83a4b7 // smmla v23.4s, v5.16b, v3.16b

    .inst 0x4e80a4d8 // smmla v24.4s, v6.16b, v0.16b
    .inst 0x4e81a4d9 // smmla v25.4s, v6.16b, v1.16b
    .inst 0x4e82a4da // smmla v26.4s, v6.16b, v2.16b
    .inst 0x4e83a4db // smmla v27.4s, v6.16b, v3.16b
    .inst 0x4e80a4fc // smmla v28.4s, v7.16b, v0.16b
    .inst 0x4e81a4fd // smmla v29.4s, v7.16b, v1.16b
    .inst 0x4e82a4fe // smmla v30.4s, v7.16b, v2.16b
    .inst 0x4e83a4ff // smmla v31.4s, v7.16b, v3.16b
    subs x26, x26, #1
    bne LoopSz_TILE_8
```

F. Optimizing Attention Operators

The computation time of the attention operator increases with the growth of the key-value (kv) cache, primarily because an expanding kv cache enlarges the scale of the past-key-value tensors in the attention operation. With the increase in size, operations such as concatenating and transposing key-value with past-key-value tensors involve more data, which directly contributes to the computational load. Furthermore, the matrix multiplication (MatMul) operations within the attention mechanism also grow in scale, which adds to the time complexity, especially when kv cache sizes become significantly large, and memory-related computations become more prominent.

In ONNX implementations, having past key-value tensors as both inputs and outputs results in substantial memory-related operations. To counteract this overhead, we can consider fusing these operators and treat the kv-cache as internal state within the merged attention operator. By doing this, we only need to write key and value tensors of length 1 on each invocation without having to process data relating to the entire past length. The required memory for this state can be pre-allocated and managed in chunks, where a chunk size equal to the pre-fill requirements would be pre-allocated to store kv tensors. When the number of decode iterations reaches the chunk size, the memory can then be reallocated (realloc). If the maximum length can be determined ahead of time, it's optimal to set the chunk size equal to this maximum length to avoid the overhead of reallocating memory.

Matrix multiplication, particularly the query @ key and qk @ value operations, tends to be one of the most time-intensive computations within the attention mechanism. Similar to elsewhere, data rearrangement can be used to improve computational efficiency; you can refer to the rearrangement strategies outlined in the aforementioned discussions. For the attention mechanism, this involves rearranging data along the e and h dimensions. These rearrangements do come with the cost

of memory copying. However, after fusing the operators, the rearrangement for the query can be merged with the original transpose operation, and for key and value with the concatenate plus transpose operations, all of which can be optimized to just a single memory copy for all computational preparations.

For matrix multiplication, float16 precision has been used to balance performance and memory footprint. However, for the softmax part, which requires higher numerical precision, float32 is used to ensure the accuracy of the operation. Bfloat16 could be used as an alternative to float16 for matrix multiplication, offering higher computational throughput, but its implementation might take additional time to optimize and validate because of the careful consideration needed for numerical stability and portability across different hardware platforms.

V. Final Performance Analysis

In this section, we will analyze the theoretical peak performance of the model on the device, the performance improvements resulting from our optimizations, and the final performance level we achieved..

A. Machine Performance Analysis

We utilized STREAM to analyze the machine's memory bandwidth and MegPeak to assess the machine's theoretical computing power. The results obtained upon submission are as follows:

1) *Memory Bandwidth:* 99.274 Gbps
2) *Theoretical Computing Power*
   a) *smmla:* 2784 GFlops.
   b) *sdot:* 2784 GFlops.
   c) *fmla:* 352 GFlops.

B. Model Computation and Memory Access Analysis

We analyze the computation and memory access of the model in three scenarios:

1) Linear Operators: Primarily utilizing smmla for computations.

2) Attention Operators: Primarily using fmla for computations.

3) Other operators: Mainly focusing on analyzing memory access.

We use the following abbreviations: l: sequence length; h: hidden size; i: intermediate size; v: vocabulary size. When using W4A8 computation and fp16 inputs, the computational and memory access requirements for Linear are as follows:

OP	Linear Computation and Memory		
	FLOP	***IO***	***weight***
attn/c_attn	2 * lh3h	2 * (lh + 3lh)	0.5 * h3h
attn/c_proj	2 * lhh	2 * (lh + lh)	0.5 * hh
mlp/w1	2 * lhi	2 * (lh + li)	0.5 * hi
mlp/w2	2 * lhi	2 * (lh + li)	0.5 * hi
mlp/c_proj	2 * lih	2 * (li + lh)	0.5 * ih
block total	6lhi + 8lhh	18lh + 6li + 2hh + 1.5hi + 10h + 4i	

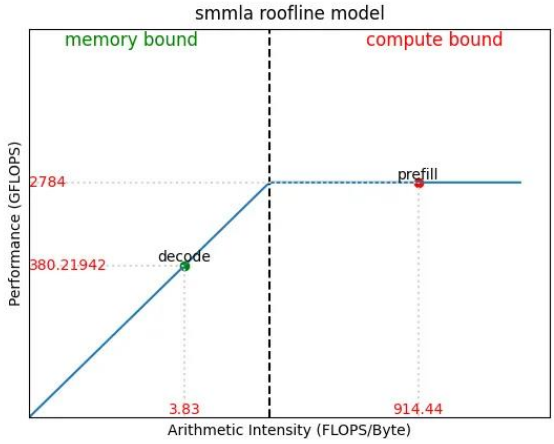
OP	Linear Computation and Memory		
	FLOP	***IO***	***weight***
Lm	2 * lhv	lh + lv + hv + 2v	

Considering the given parameters for the Qwen1.8B model, we can calculate the aggregate computational and memory access demands for the Linear layers in it. With (l=560), (h = 2048), (i = 5504) , and (v = 151936), alongside 24 blocks and one Lm layer, we can derive the following value:

stage	***FLOP***	***Memory***
prefill	1591.13 G	1.74 G
decode	2.84 G	0.71 G

Roofline models are a powerful visual method for understanding the performance characteristics of compute devices and applications. They graphically represent the maximum performance (in terms of computational throughput) as limited by either the machine's peak computational capacity or memory bandwidth. The computational intensity (I) of a device or application is typically measured as the number of flops executed per byte of memory accessed. Computational intensity helps to determine whether an application is compute-bound (limited by the machine's flops rate) or memory-bound (limited by the machine's memory bandwidth). The computational intensities are as follows:

1) *Device's Computational Intensity*
   a) I_smmla = 2784 / 99.274 = 28.04
2) *Model's Computational Intensity*
   a) I_prefill = 1591.13 / 1.74 = 914.44
   b) I_decode = 2.84 / 0.74 = 3.83



During the decode process, the sequence length is represented by "L". The computational and memory access requirements for Linear are as follows:

stage	***FLOP***	***Memory***
prefill	4llh + 22ll	16lh + 24ll
decode	4Lh + 22L	8Lh + 24L + 8h

Considering the given parameters for the Qwen1.8B model, we can calculate the aggregate computational and memory access demands for the Attention layers in it. With (l=560), (h =

2048), and(L=585) , alongside 24 blocks, we can derive the following value:

stage	***FLOP***	***Memory***
prefill	57.57 G	0.58 G
decode	0.11 G	0.21 G

The computational intensities are as follows:

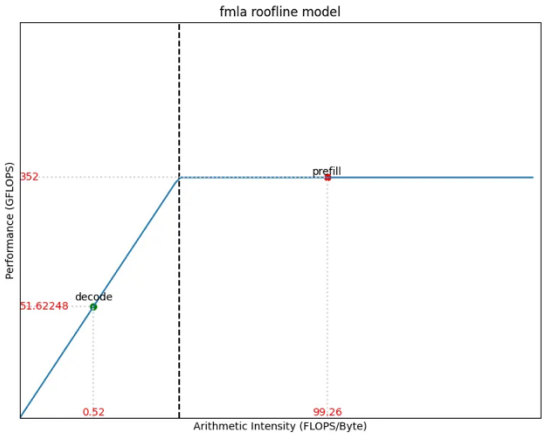
1) Device's Computational Intensity

a) I_fmla = 352 / 99.274 = 3.55

2) Model's Computational Intensity

a) I_prefill = 57.57 / 0.58 = 99.26

b) I_decode = 0.11 / 0.21 = 0.52



For other operators, the analysis of their computational intensities mostly points to them being memory-bound. It is estimated that each block's memory access is approximately (200lh), enabling us to calculate the memory access for both the prefill and decode phases as follows:

1) Prefill: 10.2539 G

2) Decode: 0.01831 G

C. Improvements from Performance Optimizatio

Based on the existing optimizations in MNN, we have made the following additional improvements in this practice:

1) Attention Fuse: Fusing the operations has significantly reduced the memory access overhead due to kv cache.

2) Pre-converting int4 values to uint4: This optimization eliminates subtraction operations within loops..

3) Arranging int4 weights in a 2x16 transpose layout: This change optimizes away zip instructions within loops.

The performance enhancements resulting from these optimizations are shown below:

Optimization	***prefill***	***decode***
Attention Fuse	34.62 %	300.54 %
Int4 to Uint4	10.91 %	113.66 %
Pre-transpose	12.02 %	7.5 %

D. Comparison Between Optimized and Peak Throughput

Based on the content of Section B of this chapter, we can calculate the theoretical peak throughput and compare it with the throughput we actually tested, as follows:

stage	***Peak***	***Our Optimized***
prefill	668.31 tok/s	638.03 tok/s
decode	105.26 tok/s	99.31 tok/s

After our optimizations, the prefill reached 95% of the theoretical peak value, and the decode throughput achieved 94% of the theoretical peak; this proves that our optimizations can effectively utilize the device's performance.