Efficient Edge Deployment of LLMs via Grouped Quantization and CPU-FPGA Co-Design

HKUSTGZ-MICS  
Junwei Cui, Yifan Yang, Zikun Wei, Liangcheng Zhao, Xunqin Lai  
Shanshi Huang, Jiayi Huang  
The Hong Kong University of Science and Technology (GuangZhou)

# Introduction

Large Language Models (LLMs) have become increasingly influential and widely applied in various domains, ranging from natural language processing to complex decision-making systems. As their applications continue to expand, the need for efficient and accessible deployment on edge devices has emerged as a critical challenge. Deploying LLMs on edge devices not only enhances their accessibility and responsiveness but also offers substantial benefits for everyday tasks, particularly in environments with limited computational resources.

However, LLM inference on edge devices faces significant challenges due to limited resources and insufficient computation rates. To solve this problem, several methods have been developed and proven effective, including quantization, pruning, and tensor compression.

Quantization is a technique used to reduce the size and computational requirements of LLMs by representing the model’s parameters with fewer bits. It is typically divided into two types: post-training quantization and quantization-aware training. Due to the substantial resources required for training, post training quantization is often preferred for large language models. Based on previous experience, INT8 quantization strikes the best balance between accuracy and model size, making it widely used for LLM compression. Pruning, on the other hand, involves cutting the unimportant weights to zero, significantly reducing operations and parameters. However, pruning usually requires fine-tuning to maintain accuracy. Another method, called tensor compression, replaces the original tensor with low-rank tensors. In this work, we focus on quantization as it better aligns with hardware optimization.

FPGAs, equipped with CPUs, provide an ideal solution for this challenge. Their flexibility, combined with relatively high inference performance, makes them particularly well-suited for efficient model deployment at the edge. The goal of this research is to develop an efficient solution for deploying LLMs on such edge devices, ensuring optimal performance while maintaining flexibility for various deployment scenarios.

In this work, we propose a novel approach that leverages the collaboration between the Processing System (PS) and Programmable Logic (PL) to efficiently deploy LLMs on FPGA-equipped edge devices. By employing grouped quantization for model compression, we reduce computational complexity without sacrificing model accuracy. Furthermore, we accelerate matrix operations by deploying appropriately sized multiply-accumulate operators in the PL. Through collaborative scheduling of computations between the PS and PL, we achieve significant improvements in inference efficiency.

This approach distinguishes itself from existing solutions by focusing on the synergistic cooperation between the PS and PL, thereby enhancing the overall performance of LLM inference at the edge. Our work addresses the challenges of resource-constrained environments while contributing to the growing field of efficient edge-side deployment of complex models.

We used INT8 symmetric per-group quantization to quantize most of the weights of Qwen2.5-0.5B, storing the rest in FP32. This reduces the model size from approximately 1.8GB to 480MB. We employ the PL on KV260 to accelerate matrix multiplication and the PS to handle remaining operations. This approach realizes the inference of the quantized model, achieving acc=0.63, prefill=5.72, and decode=4.16 and a compression rate of 0.484.

# Hardware Methods

In the hardware design of this system, the matrix multiplication operation is realized through a multiply-accumulate (MAC) unit implemented in the FPGA's Programmable Logic (PL). Data exchange and register read/write operations between the PL and Processing System (PS) are facilitated through the AXI interface. The system-on-chip (SoC) architecture of the hardware design is shown in Figure 1, where green arrows represent data paths and red arrows indicate control paths.

The PS interacts with the FPGA through three AXI4-Full master interfaces: The first AXI interface transfers quantization scale factors (Scale) and Xq data from DDR memory to on-chip buffers, while also performing register read/write operations on the FPGA. These registers are critical for configuring, activating, and monitoring on-chip acceleration units. The second AXI interface is responsible for transferring weight matrix Wq data from DDR memory to on-chip buffers. The third AXI interface reads the computed results of matrix multiplication from buffers and writes them back to DDR.

The on-chip acceleration unit, termed the Processing Elements Array (PEA), is configured and activated through registers. During operation, the PEA updates its status to registers, reads data from designated buffers for matrix computations, and writes intermediate results (e.g., partial sums) back to dedicated buffers for subsequent processing. This design ensures efficient matrix computation execution while maintaining seamless communication between the PS and PL.

By integrating PS-PL interfaces with the MAC architecture in the PL, the system achieves high-efficiency matrix operations, essential for accelerating machine learning workloads (e.g., LLM inference) on edge devices. The combination of dedicated computing hardware and flexible control/data paths provides a robust solution for real-time, resource-constrained applications.

手机屏幕截图

AI 生成的内容可能不正确。

**Figure 1. Accelerator System-on-Chip Architecture**

The hardware design comprises two primary components: the PEA module and the Prefill\_scale module.

The PEA module is designed for matrix multiplication between Xq (dimensions: 10×128) and Wq (dimensions: 128×896), both in INT8 format. It incorporates 32 parallel 32-bit multiply-accumulate trees and a matrix multiplication control state machine. Each MAC tree computes the product of 32 INT8 elements, sums the results with an INT32 constant, and outputs an INT32 result. The tree structure includes: Multiplication unit, Addition tree (pairwise summation over 5 pipeline stages), Constant accumulation unit. The fully pipelined design spans 7 stages: 1 for multiplication, 5 for addition, and 1 for constant accumulation, enabling high-speed matrix-matrix multiplication.

图示

AI 生成的内容可能不正确。

**Figure 2. PEA and Prefill\_scale’s Microarchitecture**

The state machine operates in four modes:

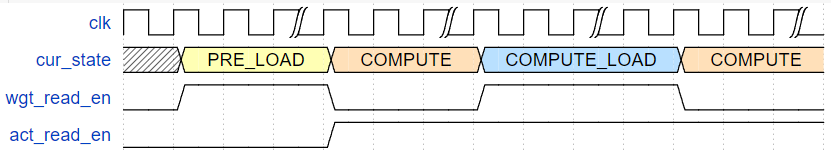
**IDLE:** Initialization state.

**PRE\_LOAD:** Pre-loads Wq weights into MAC trees.

**COMPUTE:** Executes MAC operations using Xq data.

**COMPUTE\_LOAD:** Concurrently computes current blocks while loading the next Wq block.

For each Xq and Wq block computation: Xq is column-partitioned with a stride of 32 columns, split into 4 blocks. Wq is first column-partitioned (32 columns) and then row-partitioned (32 rows), yielding 112 blocks in total.



**Figure 3. PEA Module FSM Timing Diagram**

The hardware design used the Weight-Stationary Mode in calculating GEMM. When loading Xq from buffer, Xq was broadcasted to 10×32 blocks row-major within blocks and column-major across blocks to all 32 MAC trees. When loading Wq from buffer, Wq was loading to 32×32 blocks column-major within blocks, iterating row-wise across the first 32 columns (4 blocks) and column-wise thereafter. Each Wq block loads over 8 clock cycles, with ping-pong registers enabling concurrent loading of the next block during COMPUTE\_LOAD. Partial sums are stored in registers for two cycles before being fed back as INT32 inputs to the MAC trees. Post-computation, results pass through 32 fully pipelined INT32-to-FP32 converters (4-cycle latency).

手机屏幕截图

AI 生成的内容可能不正确。

**Figure 4. PEA Matrix Partitioning Schematic**

The Prefill\_scale module manages the multiplication of quantization scaling factors to enable scale-aware matrix operations. Designed for the Xq matrix (dimensions: 10×128, INT8 format) and Wq matrix (dimensions: 128×896, INT8 format), it processes their respective FP32 quantization factors: Sx (10×1) for Xq and Sw (1×896) for Wq. The module incorporates 32 fully pipelined FP32 multipliers synchronized with the PEA module’s computational flow. Quantization factors are stored in a dedicated Scale\_Buffer, where Sx and Sw are multiplied to generate intermediate scaling factors (Sr).

These Sr values are then multiplied with the PEA module’s output (Rr)—converted from INT32 to FP32—using the same pipelined FP32 multipliers. The scaled results are buffered and subsequently retrieved by the Processing System (PS) via AXI interfaces. By tightly coupling scale-factor computation with matrix multiplication pipelines and leveraging fully optimized data alignment, this architecture achieves efficient, low-latency scaling operations—critical for deploying large language models (LLMs) on resource-constrained edge devices where real-time inference and minimal power consumption are paramount.

# Software Design Methods

In this section, we discuss our optimization method on software. Firstly, we utilize the INT8 quantization to speed up the inference and decrease the memory used as well as match up the efficient hardware design. Secondly, finetune technique improves the performance in the textual entailment tasks.

## Quantization

For the software design, group-wise quantization is applied. Every 128 values from each row of the weight matrix are treated as a group and quantized with a shared scale. This method stores integer weights Wq (INT8) along with a smaller group of floating-point scales Sw (FP32) in the binary file, instead of the original floating-point weights W (FP32). For matrix-vector multiplication, the same group-wise quantization is applied to the input, generating Xq (INT8) and Sx (FP32). As a result, INT8 multiplications are employed, and the products are summed into INT32 partial results. These INT-based matrix-vector multiplications are implemented on the PL side. For further processing, these partial results are scaled back to FP32 values and accumulated, which is handled on the PS side. Experiments with bias quantization show that, although the bias is very important, the accuracy remains sufficiently high after the bias quantization. Additionally, the token embedding table is quantized. In the meantime, other nonlinear operations, such as RMS normalization and position rotary embedding, are calculated on the PS side due to their need for higher precision.

During inference, tokens are processed sequentially at the prefill stage without calculating the logits, as prediction is not required at this stage. Thus, matrix-vector multiplication is the primary operation during the prefill stage. Similarly, in the decoding stage, one token is taken as an input in each iteration. The KV cache technique is employed to avoid redundant operations, further optimizing performance.

## Finetune

Considering the dataset change from small one to the larger one WNLI, it costs obvious accuracy loss which the baseline of the original Qwen2.5-0.5B-Instrcut model only has 0.51 in accuracy performance. In this work, in order to increase the behavior of the large language model, we integrate the finetune method in our model. The prototype overall architecture of finetune we used in this work is LlamaFactory which is an efficient and concise tool for large language model finetune.

The dataset for final evaluation is WNLI task in GLUE dataset which is a textual entailment task. To make it learn more knowledge of this specific task, we choose the RTE, CB and MNLI dataset which has 2490, 250, and 39w samples respectively and they are all for textual entailment task. However, due to the distinguished ability for large language models, overfitting is a considerable problem easily caused by using simple finetune data. In that case, following by the experience, another general task dataset SNLI is merged in original dataset with 1:1 proportion.

The finetune method LoRa reduces the parameters for finetuning and speeds up the whole process. The main idea is to use some of the sub matrix to represent the huge original matrix. Besides, Noisy Embedding Instruction Finetuning (NEFT) is introduced to weaken the overfitting problem by fusing the random noise in embedding system.

# Results

In this section, we present the final results obtained from our methods, providing a clear and concise summary of our findings.

We first developed a test code to validate the functionality of the a forementioned PEA module, specifically focusing on the multiplication of a one-dimensional vector with a matrix. Initially, the program loads the FPGA bitstream and establishes MMIO windows for interaction with the FPGA. An input matrix A of size 10x128, two scale vectors Sx and Sw of size 10x1, 1x896 and a weight matrix B of size 128x896 are then initialized, followed by the generation of a result matrix and the calculation of theoretical values. Data is written into the PEA module through the MMIO interface to trigger computation. Upon completion, the results are read and compared with the values calculated on the PS side to verify accuracy. Finally, all MMIO windows are closed.

This test code is employed to perform matrix multiplication on the PL side, with the PS side tested based on the Qwen large language model. The system clock frequency on the PL side is set to 115 MHz, which is the maximum clock frequency that does not result in setup time violations on the PL. The test results are as follows:

**Accuracy**: The accuracy of the system is approximately 63.38%. Further fine-tuning of the model on the PS side or adjustments to the hardware configuration on the PL side could contribute to enhanced performance.

**Compression Rate**: The model achieves a compression rate of approximately 0.484, with a model size of around 500MB. The model has been compressed by about 50% of its original size, and during the compression process, redundant parameters have been minimized to the greatest extent possible. This helps to make the model more efficient for deployment on devices with limited computational resources, such as FPGA-based systems.

**Throughput**: The system’s throughput is 5.72 tokens per second for pre-filling and 4.16 tokens per second for decoding. Increasing the frequency on the PL side can lead to a certain degree of throughput improvement.

However, for more significant enhancements on the PL side, although the theoretical maximum data width of each AXI interface can be configured to 128 bits, this design only supports up to 32 bits in practice. This significantly limits the data transfer bandwidth, which may be related to the use of C/C++ APIs for PS-PL communication in this implementation, rather than leveraging the official Pynq framework. Furthermore, the matrix multiplication processing speed far exceeds the data transfer rate between the PS and PL, constraining further performance improvements. More critically, the matrix multiplication unit on the PL side consumes a substantial portion of on-chip logic resources, leaving insufficient resources to deploy other required computational operators on the PL.