Techniical Report

Team name: Post-Moore AI Chip Team Cheng Zhang Zhenhao Wang Yingjie Zhao
Superviosr Names: Wei Mao and Bo Li

I. Introduction

In recent years, Large Language Models (LLMs) based on pre-training and Transformer architectures have achieved remarkable performance in various downstream natural language processing tasks, such as text understanding, text generation, sentiment analysis, machine translation, and interactive question answering. However, deploying these models efficiently on edge devices faces significant challenges due to data privacy concerns and computational efficiency requirements. The vast number of parameters in LLMs, ranging from billions to trillions, makes it difficult to deploy them efficiently on edge devices. Moreover, the growth rate of model parameters far exceeds the improvement rate of hardware performance. To address these challenges, both academia and industry are exploring methods such as model compression, data flow optimization, and operator invocation to deploy and run large models under limited hardware conditions. Designing and implementing specialized accelerators for efficient LLM inference is a viable technical approach. To promote research in this area and cultivate interdisciplinary talents in computer architecture, circuits and systems, artificial intelligence, and high-performance computing, the AICAS competition this year will focus on optimizing the performance of large models on FPGA platforms. This competition aims to explore innovative solutions to improve the efficiency of LLM inference on edge devices while addressing the challenges of deploying large models with limited hardware resources.

II. Background

A. Inference stages

The inference process of some LLMs, such as Qwen, can be divided into two distinct stages: the prefill and the decode stage, as illustrated in Fig. 1.

During the prefill stage, the LLMs process long sequences of tokens with the aim of providing the necessary context information to understand the input data and to prepare for the generation task. The primary operation in this stage is general matrix multiplication (GEMM). During the decode stage, the LLMs only process one token at a time during each generation step. This stage is tasked with converting the model-generated token sequence into readable text, essentially performing the decoding of the output. The core operation here is general matrix vector multiplication (GEMV).

Consequently, improving the performance of LLMs can be achieved by accelerating both GEMM and GEMV operations.

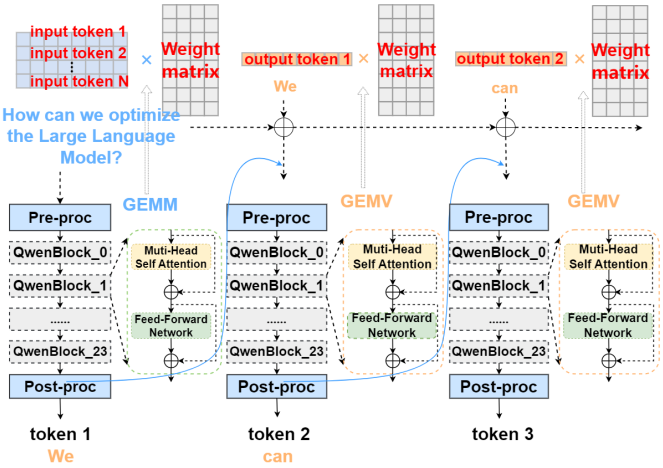

Fig. 1. An example for prefill and decode stages.

B. PYNQ

PYNQ is an open-source project from AMD. It provides a Jupyter-based framework with Python APIs for using AMD Xilinx Adaptive Computing platforms.Simplifying the development and interaction of Zynq SoC (FPGA + ARM processor) with Python. It integrates programmable logic (FPGA) with a software ecosystem, enabling the rapid construction of high-performance embedded systems using Python.

PYNQ framework provides software developers with Python interfaces to access FPGA resources. When developing in Python, these implementation details can be ignored, allowing for easy access to the FPGA and dynamic loading of various pre-compiled FPGA applications. You can call various FPGA-accelerated applications or access peripherals connected to the FPGA as if they were functions. Building programs on PYNQ allows you to easily enjoy the benefits of FPGA parallel computing and flexible configuration. As for various FPGA applications, they still need to be developed on vivado.

III. Method

In this section, we provide a detailed and systematic explanation of the methods we designed and implemented to address the challenge of deploying large language models (LLMs) on edge devices, specifically focusing on accelerating matrix multiplication using FPGA. Our approach involves several key steps, including model quantization, inference adaptation, and hardware acceleration through the development of a general

matrix multiplication accelerator. We also discuss the challenges encountered during the process and the solutions we implemented.

A. Model Quantization and Inference Adaptation

To enable efficient deployment of LLMs on edge devices, we first quantized the model using the export_qwen2_bin tool, targeting an 8-bit quantization scheme (Q8_0). This process significantly reduces the model size and computational requirements while maintaining a reasonable level of accuracy. We modified the run.c file to adapt the inference process to the quantized model, ensuring compatibility and efficient execution.

During this adaptation, we identified a critical issue in the original inference code: the lack of proper handling for bias terms. This omission could lead to inaccuracies in the model's output. To address this, we modified the export_qwen2_bin tool to include bias handling, thereby improving the overall accuracy and reliability of the quantized model.

B. PYNQ platform deployment

PYNQ supports Zynq based boards (Zynq, Zynq Ultrascale+, Zynq RFSoC), Kria SOMs, Xilinx Alveo accelerator boards and AWS-F1 instances.For the Kria KV260 Vision AI Starter Kit, you need to first download the Ubuntu 22.04 LTS SD Card image.The Ubuntu 22.04 LTS image is specifically designed for Kria SOM accelerated applications and hardware overlays for the K26 SOM and KV260 Starter Kit. It provides access to a rich set of third-party software libraries in the Ubuntu community and allows running the latest accelerated applications from the App Store to evaluate the KV260 running Ubuntu.

After burning the image onto the SD card and inserting it into the KV260 to start the system, you need to clone the Xilinx/Kria-PYNQ repository from GitHub. This repository includes the installation and download procedures required to install PYNQ suitable for the KV260 on a Linux system. After cloning the repository on the KV260, navigate into the Kria-PYNQ folder and run 'sudo bash install.sh -b KV260'. Then wait for the installation of PYNQ on the KV260 to complete.

Then we can access Jupyter Notebook through the provided URL and password in any browser, and then we can proceed with Python development on the KV260.

C. Hardware Acceleration: FPGA-Based Matrix Multiplication

To further enhance the efficiency of large language model inference on edge devices, we focused on accelerating the matrix multiplication operation, which is a key computational bottleneck in large language models. We designed and implemented a custom general matrix-vector multiplication kernel (GEMV Kernel) using Verilog HDL, targeting the migration of a large number of matrix calculations and dequantization processes in the inference framework to the FPGA. The kernel leverages the parallel processing capabilities of FPGAs to achieve significant speedup compared to traditional CPU-based implementations.

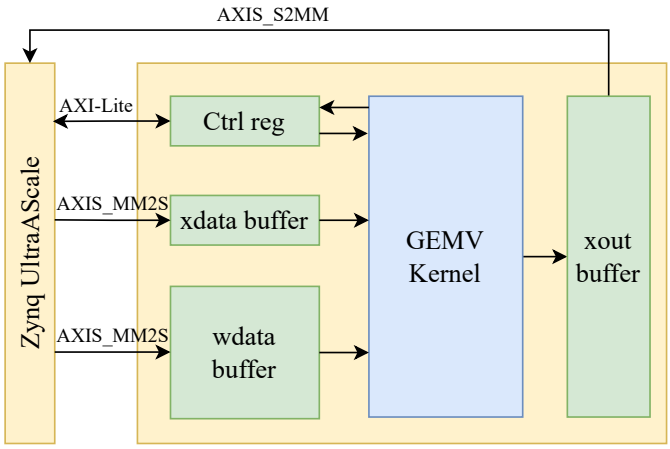

Fig. 2. Overall structure of the block design.

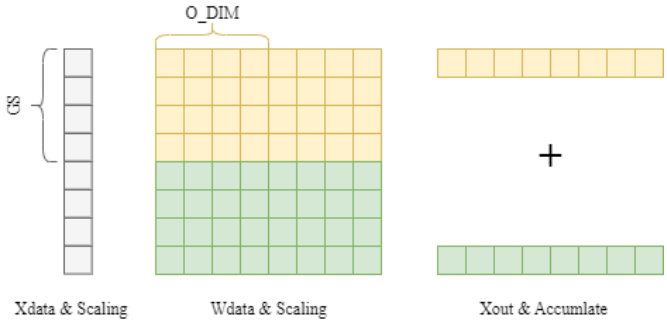

Fig. 3. GEMV Kernel design.

1) Overall structure: The overall structure of the block design is shown in the Fig. 2. The inference framework is deployed on the SoC of KV260, while the FPGA side is responsible for matrix multiplication and dequantization. There are three on-chip buffers on the FPGA side: the xdata buffer is used to cache the input word vectors or the split prompt vectors; the wdata buffer is used to cache the partial weight matrix required for the current GEMV; and the xout buffer is used to cache the result vector after the GEMV calculation. The SoC side communicates with the FPGA side through the AXI bus. The SoC sends control signals to the Ctrl reg and the control registers of the two AXI DMAs via AXI-Lite to enable the GEMV kernel to perform calculations and DMA data transfer. After the calculation is completed, the GEMV kernel returns a calculation completion signal to the SoC via AXI-Lite, and the next round of calculation can begin.

2) GEMV kernel: As shown in Fig. 3, the GEMV kernel employs batched computation and cached accumulation. The size of each input vector is I_DIM, which is the total dimension of the vectors to be computed. The input matrix consists of O_DIM vectors of I_DIM dimensions. The size of O_DIM is determined by the KV260 computing resources. Inside the GEMV kernel, computations and dequantization are performed in batches of size GS each time, yielding O_DIM intermediate

TABLE I
RESOURCE UTILIZATION OF OUR DESIGN.

Resource	Utilization	Available	Utilization%
LUT	47927	117120	40.92
FF	25537	234240	10.90
BRAM	28	144	19.44
DSP	1136	1248	91.03
BUFG	1	352	0.28

results that are cached. The final results are obtained only after all vectors have been traversed. With this approach, the number of memory accesses for a single computation vector is reduced from O_DIM times to just 1 time.

In summary, our FPGA-based matrix multiplication accelerator demonstrates the feasibility of accelerating LLM inference using hardware acceleration, with future work focusing on further optimization to improve performance and resource utilization.

D. Verification and Challenges

After our theoretical calculations, the final O_DIM is determined to be 16. To validate the functionality of GEMV Kernel, we developed a testbench to simulate the module's behavior under various input conditions. The testbench provides stimulus to the module, monitors the output, and verifies the correctness of the results. The post synethesis simulation result is in line with our expectations. Besides, the utilization of resources in our design is shown in Table I. We have achieved a higher utilization rate of DSP resources, but the utilization rate of other resources still needs to be improved.

Our work still has some imperfections. For example, the utilization of resources is not efficient enough, resulting in a relatively small batch size for batched computation. Due to the time constraint, we did not systematically design the GEMM kernel and instead used multiple GEMV computations as a substitute, which led to low efficiency. We will make targeted improvements in the future.

IV. RESULTS

Accuracy: The precision of our quantized model achieved an accuracy of 0.5079, indicating a satisfactory trade-off between computational efficiency and output quality. This result confirms that our quantization and hardware acceleration strategies effectively preserve the model's performance.

Compression Ratio: We achieved a compression ratio of 0.4686, significantly reducing the model size and computational requirements. This compression enables more efficient deployment on edge devices with limited resources, while maintaining the essential functionality of the model.

Throughput: The throughput performance of our accelerator is evaluated in two phases: prefill and decode. During the prefill stage, our accelerator achieved a throughput of 7.4967, demonstrating efficient handling of initial computations. In the decode stage, the throughput was 4.17, showcasing the accelerator's ability to sustain high performance throughout the inference process.