Pushing up to the Limit of Memory Bandwidth and Capacity Utilization for Efficient LLM Decoding on Embedded FPGA

Jindong Li[1,2,4] Tenglong Li[1,2,4] Guobin Shen[1,2,3] Dongcheng Zhao[1,2] Qian Zhang[1,2,4] Yi Zeng[1,2,3,4,5]
[1]Brain-inspired Cognitive Intelligence Lab, Institute of Automation, Chinese Academy of Sciences
[2] Center for Long-term Artificial Intelligence
[3] School of Future Technology, [4] School of Artificial Intelligence, University of Chinese Academy of Sciences
[5] Key Laboratory of Brain Cognition and Brain-inspired Intelligence Technology, Chinese Academy of Sciences
{lijindong2022, litenglong2023, shenguobin2021,zhaodongcheng2016, q.zhang, yi.zeng}@ia.ac.cn

***Abstract*—The extremely high computational and storage demands of large language models have excluded most edge devices, which were widely used for efficient machine learning, from being viable options. A typical edge device usually only has 4GB of memory capacity and a bandwidth of less than 20GB/s, while a large language model quantized to 4-bit precision with 7B parameters already requires 3.5GB of capacity, and its decoding process is purely bandwidth-bound. In this paper, we aim to explore these limits by proposing a hardware accelerator for large language model (LLM) inference on the Zynq-based KV260 platform, equipped with 4GB of 64-bit 2400Mbps DDR4 memory. We successfully deploy a LLaMA2-7B model, achieving a decoding speed of around 5 token/s, utilizing 93.3% of the memory capacity and reaching 85% decoding speed of the theoretical memory bandwidth limit. To fully reserve the memory capacity for model weights and key-value cache, we develop the system in a bare-metal environment without an operating system. To fully reserve the bandwidth for model weight transfers, we implement a customized dataflow with an operator fusion pipeline and propose a data arrangement format that can maximize the data transaction efficiency. This research marks the first attempt to deploy a 7B level LLM on a standalone embedded field programmable gate array (FPGA) device. It provides key insights into efficient LLM inference on embedded FPGA devices and provides guidelines for future architecture design.**

***Index Terms*—FPGA, Accelerator, Large language model**

I. Introduction

Large Language Models (LLMs) have revolutionized natural language processing (NLP) by enabling applications ranging from chatbots to complex analytical tools. These models can generate human-like text, provide insightful analysis and assist in complex decision-making processes, transforming industries and enhancing user experiences. However, their utility comes at a great cost: LLMs are compute and memory-intensive, requiring substantial resources for training and inference. Yet, cloud-based solution introduces dependencies on internet connectivity and potential concerns about privacy and security.

In the past decade, there has been extensive research on Field Programmable Gate Array(FPGA)-based accelerators for efficient deep learning models like convolutional neural networks [1] [2], spiking neural networks [3] [4] and vision transformers [5] [6]. However, when faced with the significantly larger scale of LLM workloads, FPGAs have yet to fully realize their potential. Recent work, DFX [7] and FlightLLM [8], has demonstrated the feasibility of deploying LLMs on cloud-based Alveo U280 FPGAs with high-bandwidth memory (HBM). However, these approaches are not applicable to embedded FPGAs. **The biggest challenge in deploying LLMs onto embedded FPGAs lies in the limited memory resources.** A typical embedded Xilinx FPGA board, such as the Ultra96v2, KV260 or ZCU104, has only 2-4GB DDR4 memory with a speed grade ranging from 1600-2400Mbps, resulting in a total bandwidth of less than 19.2GB/s. LLaMA2-7B [9], an entry-level LLM, requires around 3.5GB of memory just for the model weights after 4-bit quantization. This does not account for the key-value cache, which grows as the context increases, further escalating the challenge.

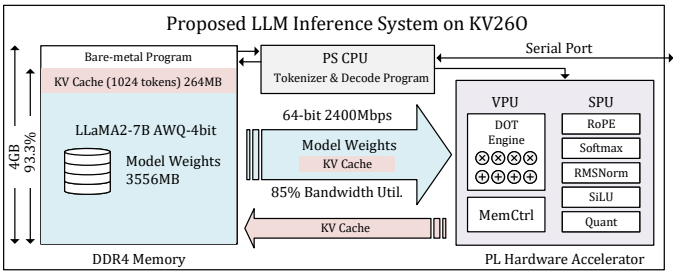

Fig. 1. LLaMA2-7B inference on embedded KV260 platform, with 93.3% of the memory capacity occupied and 85% of the memory bandwidth utilization.

To the best of our knowledge, no prior research has attempted to deploy LLMs with 7B parameters on embedded FPGA devices. Therefore, in this paper, we take on this challenge and aim to push the bandwidth and capacity limits of embedded FPGAs for LLM inference. The key contributions of this paper are listed below.

1) For the first time, we successfully deployed a 7B parameter LLM on an embedded FPGA, the KV260. With only 4GB of available memory, we developed the system in a bare-metal

Corresponding author: q.zhang@ia.ac.cn and yi.zeng@ia.ac.cn.

978-3-9826741-0-0/DATE25/© 2025 EDAA

environment, reserving 93.3% of the space for model weights and the key-value cache, supporting a context length of up to 1024 tokens, as shown in Fig.1.

2) To improve bandwidth utilization for the bandwidth-bound LLM decoding process, we implemented a customized dataflow with an operator fusion pipeline and proposed a custom data arrangement format to maximize data transaction efficiency. Despite the limited 19.2GB/s bandwidth of the KV260 platform, we achieve a decoding speed of around 5 token/s, 85% of the theoretical decoding limit.

3) We implemented a bandwidth-area optimized hardware architecture that covers all the necessary operations for LLM inference, including LayerNorm [10], Softmax [11], RoPE [12], SiLU [13], online quantization, and dequantization. We successfully fit the architecture within the KV260's limited hardware resource budget and achieved operation at 300 MHz, even with up to 70% system LUT utilization.

II. Related Work

LLMs are based on the Transformer architecture [14]. Hardware architecture for Transformer have been extensively studied in the pre-LLM era, with numerous hardware accelerators optimized for models such as ViT [15] and BERT [16]. HeatVit [6] proposed a hardware-efficient image-adaptive token pruning framework for efficient yet accurate ViT acceleration on embedded FPGAs. Liu et al. [17] proposed to fully quantize the BERT and propose an accelerator on Xilinx ZCU102 and ZCU11 FPGA and achieve high power efficiency. While these methods have proven effective for ViT and BERT models, they cannot be directly applied to LLM workloads.

In the era of LLMs, research on hardware accelerators tailored for LLMs have gradually advanced. As LLMs are memory-intensive, recent work has focused on accelerating LLMs using HBM-equipped cloud FPGAs, such as the Xilinx Alveo U280 and VCU128, achieving decoding speed improvements over GPUs in single-batch LLM inference setups. DFX [7] represents one of the first studies on FPGA accelerators for decoder-only transformer, utilizing four banks of Alveo U280 to accelerate the GPT-2 1.5B model [18]. FlightLLM [8] has deployed the LLaMA2-7B model [9] on both the Alveo U280 and the Versal VHK158, demonstrating advantages over NVIDIA GPUs in single-batch inference tasks. Chen et al. [19] provide a comprehensive analytical framework, offering the first in-depth examination of both the benefits and limitations of FPGA-based LLM spatial acceleration using the Alveo U280. EdgeLLM [20] adopts the VCU128, which is also a very large FPGA equipped with HBM, to deploy LLaMA2-7B, achieving a decoding speed higher than FlightLLM.

While these studies represent the initial steps in FPGA-based LLM acceleration for single-batch decoding, they are not applicable to real-world scenarios. Expensive server-class FPGAs are expected to handle multiple concurrent user inquiries, whereas single-batch decoding is more commonly seen in edge environments. Accelerating single-batch decoding for LLMs with embedded FPGAs remains a challenging yet unexplored area, which will be the focus of this paper.

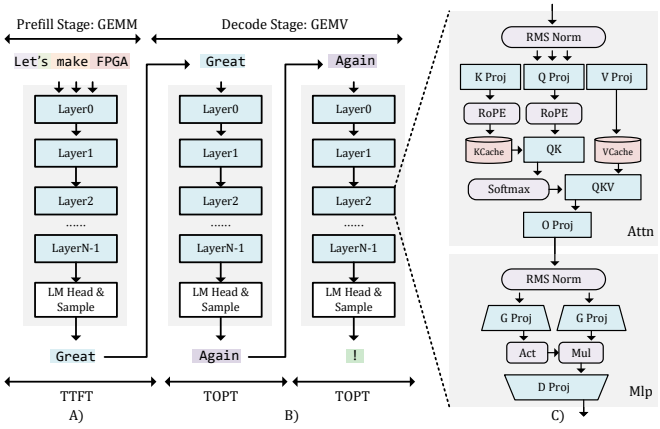

Fig. 2. LLM Inference Process of a LLaMA-like model. A) The prefill phase. B) The decode phase. C) Inference process breakdown of a single layer.

III. LLM Basics

A typical LLM model, such as LLaMA, consists of dozens of identical cascading building blocks, with each block comprising an attention layer and an MLP layer, as shown in Fig.2C. The attention layer consists of three projection layers to generate the query, key, and value, and the multi-head attention mechanism is applied to the current QKV and the history KV cache. The MLP layer includes up and down projection layers, with an additional gate projection layer applied to the up projection output.

The generative inference process of LLMs consists of two phases: the prefill phase and the decode phase, as shown in Fig.2. During the prefill phase, the model processes user prompts through the entire model architecture to produce the initial token, as shown in Fig.2A. This stage involves processing multiple tokens simultaneously, necessitating compute-bound matrix-matrix multiplication in the linear projection layer. In the decode phase, the model uses the previously generated token to sequentially generate new tokens in an auto-regressive manner, as shown in Fig.2B.

In this work, we follow the insights from Chen's analysis [19]: existing FPGAs are less efficient than GPUs during the compute-intensive prefill stage but can outperform GPUs in the memory-intensive decode stage. We focus on optimizing the decode stage of LLM inference.

IV. Algorithmic Optimization

A. Model Weight Quantization: W4A16

Post-training quantization (PTQ) has become a common practice to lowers the computational and memory demands of LLMs, as quantization aware training (QAT) of LLM becomes computationally impratical. FlightLLM employed SmoothQuant [21] to quantize both weights and activations to 8-bit. Later research, AWQ [22], indicated that quantizing weights to a lower bit-width can yield greater speed improvements than quantizing activations in LLM inference. AWQ quantized weights to 4-bit while keeping activations as 16-bit floating points, achieving less performance loss

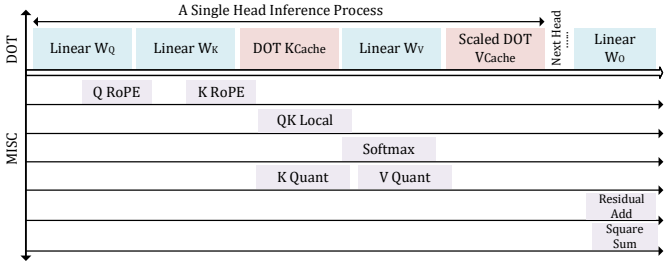

Fig. 3. The pipelining dataflow in the attention layer, with all the miscellaneous process hidden in the dense computation to avoid cycle penalties.

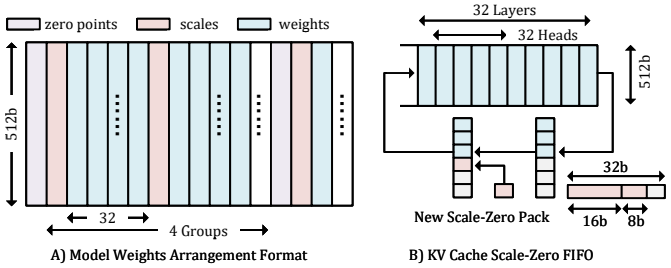

Fig. 4. Bus-width Aligned Data Arrangement Format. A) Compact model weight arrangement format interleaving zero points, scales, and weights. B) KV cache scale-zero packing process to minimize scalar data transfers.

than SmoothQuant while significantly enhancing decoding performance and reducing memory requirements. We adopt the W4A16 quantization strategy by AWQ in this work.

B. Key-Value Cache Quantization: KV8

Quantizing the KV cache can also reduce memory requirements as the context size increases. While it is possible to aggressively quantize the KV cache to 4-bit, it is recommended that for smaller models ($\leq$13B), KV8 quantization is more suitable for preserving capabilities such as multi-step reasoning and self-calibration [23]. In this work, we adopt the KV8 linear quantization strategy, where the quantization process is performed on-chip as the key and value are generated and then sent back to external memory. The dequantization process occurs when fetching the key and value from memory.

V. Hardware Acceleration Strategy

A. Dataflow in the Attention Layer

To achieve high bandwidth utilization, it is crucial to conceal miscellaneous processes during the dense computation to ensure no cycle penalties. The attention layer in the transformer inference process involves the majority of miscellaneous operations, making it the focus of our research.

Fig.3 shows the fine-grain pipelining dataflow in the attention layer during the inference process. Unlike DFX's [7] coarse-grained pipeline where the query, key, and value projections occur before the multi-head attention, we adopt a fine-grained, head-wise pipeline that fuses the projection and attention computation processes. In the pipeline of each single head, the query projection occurs first, followed by the key projection. During the DOT operation, the query and key are generated element by element. RoPE is applied to the query and key on-the-fly. The product of the current query and the current key is computed after the RoPE. Next, the DOT operation between the rotated query and the historical key cache takes place, followed by the value projection. This sequencing allows sufficient time for the softmax operation, ensuring the softmax output is prepared before the final stage—summing the values weighted by the softmax probabilities. Finally, the scaled-dot product between the attention score and the values of each historical token is computed. It is also important to note that the quantization process occurs concurrently as the current key and value are generated. After all heads complete their computations, the outputs are concatenated and used as input for the output projection operation. As the output projection results are generated, the residual connection is added, and the square sum required for post-attention layer normalization is computed simultaneously. All the miscellaneous operations are hidden in the dense computation of the attention layer, ensuring no cycle penalties.

B. Data Arrangement Format

One of the most effective strategies for optimizing bandwidth utilization is implementing sequential burst transfers. Large consecutive burst transfers can achieve significantly higher bandwidth efficiency compared to short bursts with discontinuous addresses. We propose a customized data arrangement format for model weights and the KV cache to ensure that all transactions occur as large consecutive bursts.

1) Model Weight Data Arrangement: Model quantization generates scales and zero points, which are used to recover the original floating-point values. While fetching the scales and zero points group by group during the quantization process is straightforward, fetching small amounts of data from DDR memory introduces high latency and reduces bandwidth utilization. Alternatively, loading all the scales and zero points at once and storing them in on-chip memory is feasible, but the size of the scales and zero points for a single linear layer is still too large for the local BRAM or URAM.

To address this, we propose a specialized data arrangement format for the 4-bit quantized model weights, where the zero points, scales, and weights are interleaved, as shown in Fig.4A. Assuming a 512-bit data bus, each transaction can carry 64 4-bit weights, 64 4-bit zero points, or 16 16-bit scales. Assuming a quantization group size of 128, each transaction for scales includes all the scales for 2,048 quantized weights, which corresponds to 32 transactions. Each transaction for zero points contains 4 groups of scales and weights, as illustrated in Fig.4A. By packing zero points, scales, and weights together, we ensure consecutive burst transfers, minimizing the on-chip buffer needed to store zero points and scales. Zero points, scales, and weights are separated for subsequent computation as they flow through the PL logic.

2) Key-Value Scale-Zero Packing: Unlike the scales and zero points for model weights, which can be pre-arranged in a specific layout, the scales and zero points for the KV cache

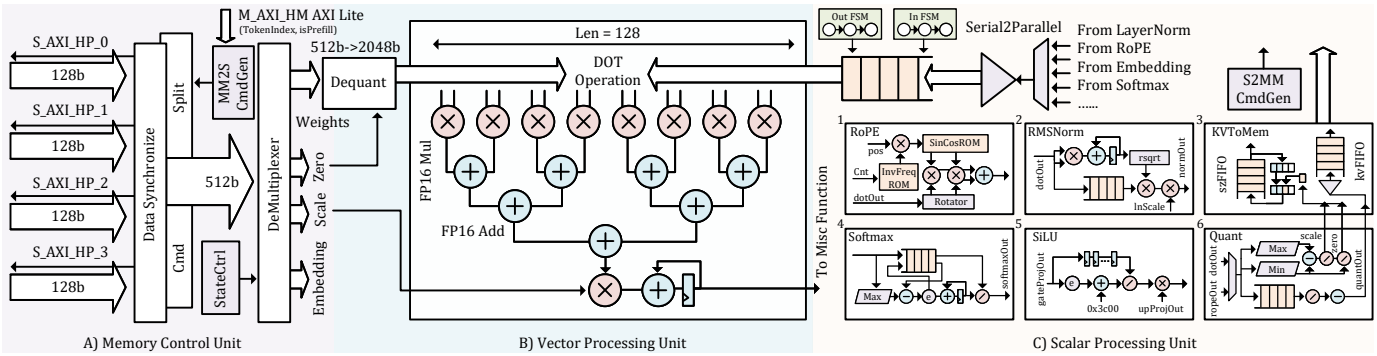

Fig. 5. Hardware Architecture of the Accelerator. A) Memory Control Unit (shaded in light purple) ensures full access to DDR bandwidth. B) Vector Processing Unit (shaded in light blue) performs dense computations. C) Scalar Processing Unit (shaded in light orange) handles miscellaneous processes.

are generated on-the-fly during inference. To avoid transferring small data volumes for scales and zero points, we maintain a FIFO buffer that stores the packed scales and zero points of the KV cache until element inside the FIFO contains all vaild packs, as shown in Fig.4B. As the inference proceeds head-wise and layer-wise for each token, every time a scale-zero pack is generated, we pop one element from the FIFO, update it by appending the new valid scale-zero pack, and then push the element back into the FIFO. Each scale-zero pack is 32-bit, consisting of 16-bit for the scale, 8-bit for the zero point, and a dummy 8-bit space for data alignment. Assuming a 512-bit data bus, such a FIFO can maintain scale-zero packs for 16 tokens. The scale-zero packs are transferred back to DDR once we begin the inference process for the 16th token. In this way, we ensure bus-aligned transfer of the KV cache scales and zero points, improving the bandwidth utilization.

VI. Hardware Architecture

The hardware architecture of the LLM accelerator we proposed consists of three main components: the Memory Control Unit (MCU), the Vector Processing Unit (VPU), and the Scalar Processing Unit (SPU), as shown in Fig.5. We will introduce the design of each component in the following subsections.

A. Memory Control Unit

The MCU ensures that the on-chip bandwidth resources are fully accessible to the PL logic. The KV260 is equipped with 4GB of 2400 Mbps 64-bit DDR4 on the processing system (PS) side, delivering a bandwidth of 19.2 GB/s. The Zynq UltraScale+ MPSoC provides several AXI ports with a maximum 128-bit, allowing the programmable logic (PL) to access the DDR4. To align with the available bandwidth of KV260, we utilize four AXI ports, with the system frequency set to 300 MHz, as shown in Fig.5A. Four 128-bit streams from the four AXI ports are synchronized, concatenated into a 512-bit stream, and sent to a demultiplexer that separates the scales, zero points, quantized weights, and token embeddings. The processing system (PS) is responsible for tokenizing the input context and sending the token index to the memory command generator via the AXI-Lite bus. The commands are then split into four, one for each AXI port.

B. Vector Processing Unit

The 512-bit weights are dequantizd to 128 sets of 16-bit floating points data and sent to the VPU for subsequent computation, as shown in Fig.5B. End-to-end LLM inference involves both the prefill and decode phases, which have significantly different computing requirements. In GPU devices, where computing resources are far more abundant than memory resources, the prefill stage involves matrix multiplication, processing multiple tokens at the same time, with most tensor cores operational and on-chip model weights being reused. In contrast, during the decode stage, most tensor cores are idle due to the bandwidth-bound nature of matrix-vector computations in the ”one token at a time” autoregressive decoding process. However, in domain-specific architecture design where power, performance, and area (PPA) are critical considerations, it is inefficient to have most computing resources idle during the decode stage. Therefore, we sacrifice some performance in the prefill stage and implement a bandwidth-area balanced DOT computing engine that can fully utilize the bandwidth **without wasting additional compute resources, reaching the optimized PPA requirements in the decode stage.** The simple vector dot engine (rather than a matrix engine [24] [25]) has 128 multipliers, matching the size of the quantized weights set after the dequantization, along with an adder tree for summing the products, a scaling multiplier, and an accumulator to perform the dot operations, shown in Fig.5B.

It is worth noting that we adopt FP16 computation on FPGA. Although floating point computation is inherently expensive on FPGAs, the decoding performance bottleneck in LLM inference is the bandwidth, not the logic resources. While recent studies have shown that more aggressive quantization methods can be applied to activations for a fully integer-only inference [26], we aim to maximize the use of existing computing resources by building a floating point compute engine to maintain LLM's accuracy as much as possible.

C. Scalar Processing Unit

The SPU performs miscellaneous functions concurrently with the VPU to ensure no cycle penalties, as demonstrated in section VA. In this subsection, we briefly introduce the design of the miscellaneous submodules in the SPU as follows.

RoPE. The RoPE submodule comprises three primary components: the rotator, the sin/cos generator, and the address generator, as shown in Fig.5C1. The rotator caches half of the query or key and generates the rotation pair. The sin/cos generator includes a look-up table (LUT) that stores 4096 points of one-quarter cycle of sine wave values in read-only memory (ROM). The address generator consists of another LUT that stores the inverted frequency values: $10000.0^{-\frac{i}{4096}}, i = 0, 2, 4...4094$, to generate the addresses for reading sine and cosine values based on the current token number. The rotation pair is then multiplied by the sine and cosine values to obtain the rotated query or key.

RMSNorm. The RMSNorm process involves two sequential passes over the input vector, as shown in Fig.5C2. The first pass computes the root mean square (RMS) of the input vector: $\mathbf{RMS}(\mathbf{a}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} a_i^2}$; however, this step can be bypassed if the mean square value can be computed by the DOT engine. The second pass performs normalization based on the RMS values obtained: $\overline{a_i} = \frac{a_i}{\mathbf{RMS}(\mathbf{a})}$.

Softmax. The softmax submodule implements a numerically stable variant [11] of the softmax function to mitigate potential numerical issues. This approach involves three sequential passes over the input vector, as shown in Fig.5C4. The first pass identifies the maximum value m of the input vector. The second pass computes the normalization term $d = \sum_{n}^{i=1} e^{x_i - m}$. Finally, the third pass calculates the resulting softmax values: $s_i = \frac{e^{x_i - m}}{d}$.

SiLU. The SiLU submodules contains logic pipeline for computing the $\frac{x}{1+e^{-x}}$ function, as shown in Fig.5C5, where x is the gate projection output in the MLP layer. The SiLU output is then multiplied by the output of the up-projection layer to produce the input for the down-projection layer.

Quantization. The quantization submodule operates in two passes over the input vector, as shown in Fig.5C6. The first pass determines the scaling factor $s = \frac{x_{max} - x_{min}}{255}$ and zero point $z = \lceil \frac{x_{min}}{s} \rceil$, while the second pass performs the quantization of the input: $(x - z) \cdot s$. The generated scale and zero point are temporarily stored in the FIFO mentioned in Section VB, and the quantized output is sent to a serial-to-parallel unit to match the AXI bus width before being transferred back to DDR memory, as shown in Fig.5C3.

D. On-chip Dataflow of Hidden States

To fully utilize the bandwidth and capacity for model weight transfers, hidden states generated during the inference process are always kept on-chip and not offloaded to DDR. The serial outputs from the RMSNorm, RoPE, softmax, and MLP gated outputs are multiplexed and sent to a serial-to-parallel adapter. These hidden states are then pushed to a DOT operand FIFO, where two in-out finite state machines ensures the correct data is sent to the VPU for DOT computation.

VII. Experiments

A. Basic Setup

The hardware accelerator is designed using SpinalHDL [27], with behavioral simulations conducted via cocotb [28].

TABLE I
Resources Consumption Breakdown of the Accelerator.

	Total	MemCtrl	VPU	SPU
	Used/Util.	Used/Util.	Used/Util.	Used/Util.
LUTs	78K / 67%	14K / 12%	34K / 30%	29K / 25%
FFs	105K / 45%	21K / 8%	44K / 19%	40K / 16%
CARRY	3.8K / 26%	0.6K / 4%	2.1K / 14%	1K / 6%
DSP	291 / 24%	1 / 0	266 / 21%	24 / 2%
URAM	10 / 16%	7 / 11%	0 / 0	3 / 4%
BRAM	36.5 / 25%	30 / 20%	0 / 0	6.5 / 5%

The Verilog codes generated by the SpinalHDL compiler are synthesized and implemented using Xilinx Vivado 2022.2. Power consumption estimates are also obtained from the reports generated by the Vivado Design Suite.

The original LLaMA2-7B model is quantized using the AutoAWQ library, converted to our proposed format, loaded onto an SD card, and then transferred to the DDR memory of the KV260 platform in the C bare-metal program. The 4GB address space in the KV260 platform is divided into two parts: the lower 2GB spans from 0x00000000-0x7FF00000, with 1MB reserved by the compiler, and the higher 2GB spans from 0x80000000-0xFFFFFFFF. We place the embedding table, model weights, and reserved space for KV cache upto 1024 tokens for the first 16 layers to the higher address space, while the remaining data is placed in the lower address space. **It is worth noting that the model weights of the LLaMA2-7B model have reached the limit of the DDR4 memory capacity, making it impossible to load a Linux operating system with so little memory remaining.**

B. Resources Consumption Breakdown

The resource consumption of the proposed accelerator is shown in Table I. The accelerator utilizes 67% of the LUTs, 45% of the FFs, 26% of the CARRYs, 24% of the DSPs, 16% of the URAMs, and 25% of the BRAMs. The VPU consumes the most LUTs and DSPs, as it serves as the core of the accelerator, performing the dense FP16 computations. The MCU consumes the most of the BRAMs and URAMs, as it contains the AXI Datamover IP and buffers the data from the AXI interface. The power consumption reported by the Vivado is **6.57W**, with the accelerator operating at **300MHz**.

C. Comparison with Existing FPGA Research

Performance comparison between our proposed accelerator and existing research is shown in Table.II. Bandwidth utilization is a reliable metric for assessing the efficiency of different platforms, as it indicates how closely frameworks can approach the theoretical decoding speed of the hardware backend. Given that decoding speed is entirely bandwidth-bound, utilization is calculated as the ratio of the measured token/s to the number of model weight transfers possible within one second.

Comparisons with research utilizing large cloud FPGAs with HBM [8] [7] [20] [19] can be hard due to the significant bandwidth disparity (460GB/s for Alveo U280 vs. 19.2GB/s for KV260).Nevertheless, for reference, we present

TABLE II
PERFORMANCE COMPARISON WITH EXISTING FPGA RESEARCH.

		Device	LUT	FF	BRAM	DSP	MHz	W	GB/s	Tasks	Opt.	token/s[1]	token/s[2]	Util. %
Cloud HBM	DFX[3]	U280	520K	1107K	1192	3533	200	45	460	GPT2-1.5B	W16	∼153	∼21[4]	13.7%
	FlightLLM	U280	574K	943K	1252	6345	225	45	460	LLaMA2-7B	W4[5]	∼131	∼55	42/65.9%[6]
	EdgeLLM	U280	967K	607K	1734	5587	250	50.7	460	ChatGLM-6B	W4	∼153	∼75	49/73.8%[7]
Edge DDR	SECDA [29]	PYNQ	/	/	/	/	/	/	2.1	TinyLLaMA[7]	W4	3.8	0.58	15.2%
	LlamaF [30]	ZCU102	164K	171K	223	528	205	5.08	21.3	TinyLLaMA	W8	19.3	1.5	7.7%
	Ours	KV260	78K	105K	36.5	291	300	6.57	19.2	LLaMA2-7B	W4	5.8	4.9	84.5%

[1] The theoretical peak decoding speed of the given bandwidth. [2] The actual reported decoding performance data. [3] The table shows the resource utilization of a single FPGA in DFX's designs. [4] DFX only reports the single FPGA decoding performance for a 345M model. The single FPGA decoding performance for the 1.5B model is extrapolated by linear scaling. [5] Although FlightLLM uses 8-bit quantization, it employs the SparseGPT method, achieving an effective average bit-width of 3.5 bits, equivalent to 4-bit quantization in terms of capacity and bandwidth. [6,7] Both FlightLLM and EdgeLLM report bandwidth utilization metrics higher than theoretical calculations, we list both of the results for reference. [8] TinyLLaMA model is 1.1B.

the performance of these works in Table.II. To the best of our knowledge, little research has been conducted on deploying LLMs on embedded FPGAs. SECDA-LLM [29], evaluates the TinyLLama-1.1B on the Pynq-Z2, achieving a performance of 0.58 token/s. Similarly, LlamaF [30] deploys the TinyLLama-1.1B on the ZCU102, achieving 1.5 token/s. While the decoding speed and bandwidth utilization of both are not yet optimal, these works represent some of the earliest efforts to deploy LLMs on edge FPGAs.

D. Comparison with Embedded CPU and GPUs

Table.III presents a comparison of decoding performance and bandwidth utilization efficiency for the 4-bit quantized LLaMA2-7B inference task between our proposed accelerator on the KV260, the embedded CPU Raspberry Pi, and the Jetson series embedded GPUs, using various inference frameworks. It is important to note that few frameworks are available for implementing LLMs on edge devices, and these frameworks vary significantly in performance. The reported decoding performance data is sourced from existing research or official websites of the respective inference frameworks. The llama.cpp [31] represents an inference framework that requires minimal setup and delivers good performance across a wide variety of hardware. However, it results in low token/s and bandwidth utilization on Raspberry Pi or Jetson AGX Orin [32]. TinyChatEngine [33] is an on-device LLM inference library enabled by LLM model compression and achieves good decoding performance on the Jetson AGX Orin [34]. However, the bandwidth utilization remains relatively low. NanoLLM [35] is a high-performance library with official support for Jetson devices. It achieves excellent decoding performance on Jetson devices, reaching around 80% bandwidth utilization. Although the KV260 has significantly less bandwidth compared to the Jetson GPUs, our proposed hardware accelerator achieves superior resource utilization with nearly 85% bandwidth efficiency, with 6% higher utilization than the Jetson Orin Nano using the NanoLLM, as shown in Table.III.

TABLE III
COMPARISON WITH EMBEDED GPUS IN 4-BIT LLAMA2-7B INFERENCE

Device	GB/s	FrameWork	token/s[1]	token/s[2]	Util.
Pi-4B 8GB	12.8	llama.cpp	3.9	0.11 [32]	2.8%
JetsonAGXOrin	204.8	llama.cpp	62.5	4.49 [32]	7.2%
		TinyChat	62.5	33 [34]	52.8%
		NanoLLM	62.5	47.1 [36]	75.4%
JetsonOrinNano	68	NanoLLM	20.7	16.4 [36]	79.2%
KV260	19.2	Ours	5.8	4.9	84.5%

[1,2] The theoretical peak decoding speed and the actual reported speed.

VIII. DISCUSSION AND CONCLUSION

We present several insights for designing architectures for LLM inference with embedded FPGAs in this section.

Memory Resources is Essential. The memory capacity of an device determines the feasibility of LLM deployment, and the performance of LLM decoding is directly tied to the available bandwidth. We have pushed up to the limit by deploying a 7B model on a 4GB device in this work. **Nevertheless, further improving LLM decoding speed and supporting larger LLM size remains challenging without sufficient bandwidth and capacity.** With DDR5 and unified memory—both of which offer increased bandwidth—becoming more common in laptops and smartphones, it is timely for FPGA vendors to integrate advanced memory support into embedded devices.

Bandwidth Utilization is Critical. When bandwidth is limited, it is essential to maximize the utilization. For CPU or GPU hardware, precisely controlling on-chip cache behavior to reserve full bandwidth for non-temporal weight transfers can be hard. This is where specialized architectures with customized cache and dataflow offer a significant advantage.

In this work, we introduce an LLM accelerator based on embedded FPGAs, capable of supporting models with up to 7 billion parameters by maximizing bandwidth and capacity utilization. As the first research to deploy a 7B LLM on an embedded FPGA, this study offers insights for the domain-specific architecture community in LLM inference applications and underscores the potential of FPGA-based LLM solutions.

IX. ACKNOWLEDGEMENT

This work, as part of the software-hardware codesigns research of the BrainCog Engine [37] [38], is supported by the Chinese Academy of Sciences Foundation Frontier Scientific Research Program (ZDBS-LY- JSC013) and a funding from Institute of Automation, Chinese Academy of Sciences (Grant No. E411230101). We would also like to thank Hongzheng Zhang and Niansong Zhang from Cornell University for providing statistics of the paper [19].

REFERENCES

[1] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang, "Angel-eye: A complete design flow for mapping cnn onto embedded fpga," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 37, no. 1, pp. 35–47, 2017.

[2] Xiang Chen, Jindong Li, and Yong Zhao, "Hardware resource and computational density efficient cnn accelerator design based on fpga," in *2021 IEEE International Conference on Integrated Circuits, Technologies and Applications (ICTA)*. IEEE, 2021, pp. 204–205.

[3] Jindong Li, Guobin Shen, Dongcheng Zhao, Qian Zhang, and Yi Zeng, "Firefly v2: Advancing hardware support for high-performance spiking neural network with a spatiotemporal fpga accelerator," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[4] Tenglong Li, Jindong Li, Guobin Shen, Dongcheng Zhao, Qian Zhang, and Yi Zeng, "Firefly-s: Exploiting dual-side sparsity for spiking neural networks acceleration with reconfigurable spatial architecture," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2024.

[5] Zhengang Li, Mengshu Sun, Alec Lu, Haoyu Ma, Geng Yuan, Yanyue Xie, Hao Tang, Yanyu Li, Miriam Leeser, Zhangyang Wang *et al.*, "Auto-vit-acc: An fpga-aware automatic acceleration framework for vision transformer with mixed-scheme quantization," in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2022, pp. 109–116.

[6] Peiyan Dong, Mengshu Sun, Alec Lu, Yanyue Xie, Kenneth Liu, Zhenglun Kong, Xin Meng, Zhengang Li, Xue Lin, Zhenman Fang *et al.*, "Heatvit: Hardware-efficient adaptive token pruning for vision transformers," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 442–455.

[7] Seongmin Hong, Seungjae Moon, Junsoo Kim, Sungjae Lee, Minsub Kim, Dongsoo Lee, and Joo-Young Kim, "Dfx: A low-latency multi-fpga appliance for accelerating transformer-based text generation," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 616–630.

[8] Shulin Zeng, Jun Liu, Guohao Dai, Xinhao Yang, Tianyu Fu, Hongyi Wang, Wenheng Ma, Hanbo Sun, Shiyao Li, Zixiao Huang *et al.*, "Flightllm: Efficient large language model inference with a complete mapping flow on fpga," *arXiv preprint arXiv:2401.03868*, 2024.

[9] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[10] Biao Zhang and Rico Sennrich, "Root mean square layer normalization," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[11] Maxim Milakov and Natalia Gimelshein, "Online normalizer calculation for softmax," *arXiv preprint arXiv:1805.02867*, 2018.

[12] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu, "Roformer: Enhanced transformer with rotary position embedding," *Neurocomputing*, vol. 568, p. 127063, 2024.

[13] Stefan Elfwing, Eiji Uchibe, and Kenji Doya, "Sigmoid-weighted linear units for neural network function approximation in reinforcement learning," *Neural networks*, vol. 107, pp. 3–11, 2018.

[14] A Vaswani, "Attention is all you need," *Advances in Neural Information Processing Systems*, 2017.

[15] Alexey Dosovitskiy, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[17] Zejian Liu, Gang Li, and Jian Cheng, "Hardware acceleration of fully quantized bert for efficient natural language processing," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 513–516.

[18] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[19] Hongzheng Chen, Jiahao Zhang, Yixiao Du, Shaojie Xiang, Zichao Yue, Niansong Zhang, Yaohui Cai, and Zhiru Zhang, "Understanding the potential of fpga-based spatial acceleration for large language model inference," *arXiv preprint arXiv:2312.15159*, 2023.

[20] Mingqiang Huang, Ao Shen, Kai Li, Haoxiang Peng, Boyu Li, and Hao Yu, "Edgellm: A highly efficient cpu-fpga heterogeneous edge accelerator for large language models," *arXiv preprint arXiv:2407.21325*, 2024.

[21] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han, "Smoothquant: Accurate and efficient post-training quantization for large language models," in *International Conference on Machine Learning*. PMLR, 2023, pp. 38 087–38 099.

[22] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han, "Awq: Activation-aware weight quantization for on-device llm compression and acceleration," *Proceedings of Machine Learning and Systems*, vol. 6, pp. 87–100, 2024.

[23] Shiyao Li, Xuefei Ning, Luning Wang, Tengxuan Liu, Xiangsheng Shi, Shengen Yan, Guohao Dai, Huazhong Yang, and Yu Wang, "Evaluating quantized large language models," *arXiv preprint arXiv:2402.18158*, 2024.

[24] Jindong Li, Tenglong Li, Guobin Shen, Dongcheng Zhao, Qian Zhang, and Yi Zeng, "Revealing untapped dsp optimization potentials for fpga-based systolic matrix engines," in *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2024, pp. 197–203.

[25] Jindong Li, Guobin Shen, Dongcheng Zhao, Qian Zhang, and Yi Zeng, "Firefly: A high-throughput hardware accelerator for spiking neural networks with efficient dsp and memory optimization," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 31, no. 8, pp. 1178–1191, 2023.

[26] Xing Hu, Yuan Chen, Dawei Yang, Sifan Zhou, Zhihang Yuan, Jiangyong Yu, and Chen Xu, "I-llm: Efficient integer-only inference for fully-quantized low-bit large language models," *arXiv preprint arXiv:2405.17849*, 2024.

[27] "Spinalhdl: Scala based hdl." [Online]. Available: https://github.com/SpinalHDL/SpinalHDL

[28] Benjamin John Rosser, "Cocotb: a python-based digital logic verification framework," in *Micro-electronics Section seminar. CERN, Geneva, Switzerland*, 2018.

[29] Jude Haris, Rappy Saha, Wenhao Hu, and José Cano, "Designing efficient llm accelerators for edge devices," *arXiv preprint arXiv:2408.00462*, 2024.

[30] Han Xu, Yutong Li, and Shihao Ji, "Llamaf: An efficient llama2 architecture accelerator on embedded fpgas," 2024. [Online]. Available: https://arxiv.org/abs/2409.11424

[31] "Georgi gerganov. ggerganov/llama.cpp: Port of facebook's llama model in c/c++." [Online]. Available: https://github.com/ggerganov/

[32] Nobel Dhar, Bobin Deng, Dan Lo, Xiaofeng Wu, Liang Zhao, and Kun Suo, "An empirical analysis and resource footprint study of deploying large language models on edge devices," in *Proceedings of the 2024 ACM Southeast Conference*, 2024, pp. 69–76.

[33] "Tinychat: Large language model on the edge." [Online]. Available: https://hanlab.mit.edu/blog/tinychat

[34] "Tinychat: Efficient and lightweight chatbot with awq." [Online]. Available: https://github.com/mit-han-lab/llm-awq/blob/main/tinychat/README.md

[35] "Optimized local inference for llms with huggingface-like apis for quantization, vision/language models, multimodal agents, speech, vector db, and rag." [Online]. Available: https://dusty-nv.github.io/NanoLLM/

[36] "Jetson ai lab benchmark." [Online]. Available: https://www.jetson-ai-lab.com/benchmarks.html

[37] "Braincog: Brain-inspired cognitive intelligence engine." [Online]. Available: http://www.brain-cog.network

[38] Yi Zeng, Dongcheng Zhao, Feifei Zhao, Guobin Shen, Yiting Dong, Enmeng Lu, Qian Zhang, Yinqian Sun, Qian Liang, Yuxuan Zhao, Zhuoya Zhao, Hongjian Fang, Yuwei Wang, Yang Li, Xin Liu, Chengcheng Du, Qingqun Kong, Zizhe Ruan, and Weida Bi, "BrainCog: A spiking neural network based, brain-inspired cognitive intelligence engine for brain-inspired AI and brain simulation," *Patterns*, p. 100789, Jul. 2023. [Online]. Available: https://doi.org/10.1016/j.patter.2023.100789