LORA: A Latency-Oriented Recurrent Architecture for Large Language Model on Multi-FPGA Platform with Communication Optimization

ZhenDong Zheng, Qianyu Cheng, Teng Wang, Wenqi Lou *Member, IEEE,*
Lei Gong *Member, IEEE,* Xianglan Chen, Chao Wang *Senior Member, IEEE,* Xuehai Zhou

***Abstract*—The remarkable performance of Large Language Models (LLMs) has driven their widespread deployment in data centers to support diverse user-facing applications. However, the rapidly growing computational and storage demands of these models have made single-device deployment increasingly impractical. Prior research on LLM inference has primarily addressed this challenge through algorithmic optimizations such as quantization or by integrating customized hardware acceleration frameworks. As model parameters continue to scale, multi-device deployment has become a necessary approach for enabling efficient LLM inference. Nevertheless, constructing low-latency multi-device platforms for LLMs inference using available FPGA or GPU accelerators remains constrained by inefficient synchronization schemes or limited compute intensity in current architectures. Furthermore, existing solutions often lack co-optimized designs that effectively integrate communication with computation.**

To address these limitations, this paper proposes LORA, a low-latency end-to-end LLMs acceleration platform utilizing multiple FPGAs. Firstly, we optimize the synchronization timing within the LLMs to minimize storage, computation, and BRAM overhead. Secondly, we tightly couple communication and computation through techniques such as pipeline overlapping and input data packing. Next, we deploy homogeneous accelerators on each FPGA device, leveraging a recurrent architecture to further reduce inference latency. Finally, we apply FPGA-specific optimizations and conduct performance modeling and analysis of the acceleration framework to select optimal deployment parameters for various computational tasks. Implemented on Xilinx Alveo U280 FPGAs, LORA-F and LORA-Q achieve average speedups of $14.4\times$ and $32.6\times$, respectively, compared to NVIDIA V100 GPUs when running modern LLMs. Compared with existing multi-FPGA accelerator platforms, LORA-F and LORA-Q demonstrate average performance improvements of up to $2.6\times$ and $4.3\times$, respectively.

***Index Terms*—Data Center, LLM, Multi-FPGA Acceleration**

I. INTRODUCTION

IN recent years, Transformer-based Large Language Models (LLMs) [1]–[4] have achieved a significant influence on both academic research and daily life [5]. Among them, the design paradigm of decoder-only Transformer models has been adopted by the majority of today's models. LLMs can achieve excellent results in complex natural language processing (NLP) applications such as scientific writing [6], machine translation [7], and text classification [8]. Part of these benefits come from the architectural innovation brought by self-attention, and on the other hand, come from a large number of parameters and training data, which also bring massive computing and memory overhead to the platform.

TABLE I: The model size with different precision

MODEL (GB)	Mistral -7B [16]	Llama2 -13B [17]	Falcon -40B [18]	Llama3 -70B [1]	OPT -175B [19]
BF16	17	31	96	170	420
W8A8	8.4	15.6	48	84	210
W4A16	6.3	12	36	63	158
W4A4	4.2	7.8	24	42	105

Researchers have extensively utilized optimization techniques such as quantization [9]–[11], pruning [12], [13], and low-rank approximation [14] to reduce the deployment costs of LLMs. Among these, quantization has prevailed over others due to its effectiveness in both performance boost and memory reduction [15]. Notably, as demonstrated in Table I, even when compressing models with state-of-the-art quantization algorithms, the storage requirements persist at 5-420 GB. Furthermore, the memory overhead of KV Cache grows substantially as input and output sequence lengths increase [20]. To reduce the huge storage burden brought by LLMs, a multi-device accelerated platform remains an essential optimization strategy as model parameters and sequence length continue to scale.

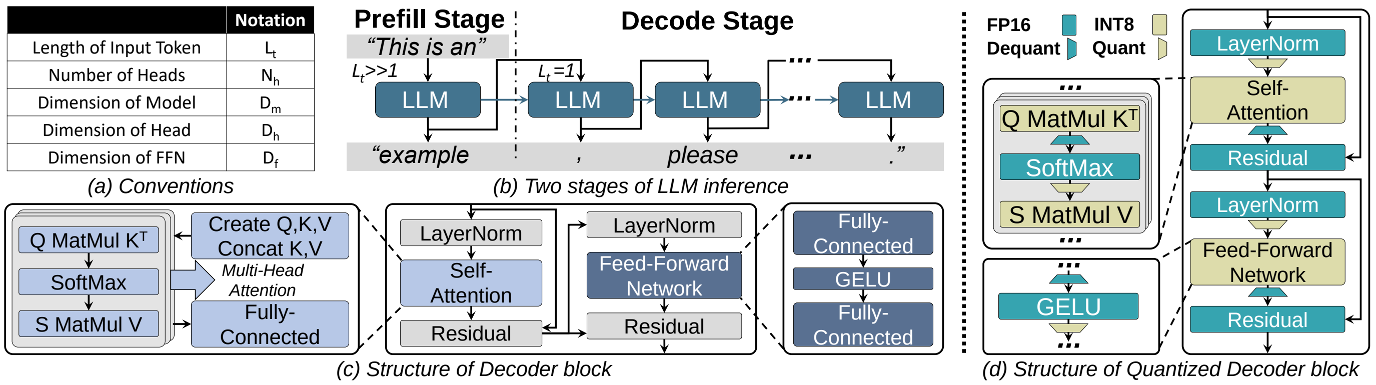
Data centers mainly rely on GPUs [21] to construct multi-device acceleration platforms. However, several factors may degrade GPU efficiency. First, the performance of the LLMs' Decode stage is severely limited by memory bandwidth, and the utilization of the Tensor Core is even less than 50% [22]. Second, to ensure high Quality of Service (QoS) and meet the real-time requirements, acceleration clusters will not process small batches of user input by waiting for more data, so the batch size is usually set to one [23]. Since GPU architectures are optimized for data-parallel workloads with large batch sizes, this setup leads to insufficient computational intensity and underutilization of resources.

Previous works have leveraged FPGAs to alleviate the aforementioned issues, taking advantage of their short development cycles, deep customizability, and low-latency characteristics.

Manuscript received xx xx, 2023; revised xx xx, 2023; accepted xx xx, 2024.

Z. Zheng, Q. Cheng, T. Wang, W. Lou, L. Gong, X. Chen, C. Wang, and X. Zhou are with the University of Science and Technology of China, Hefei 230026, China, and also with Suzhou Institute for Advanced Research, University of Science and Technology of China, Suzhou 215123, China. L. Gong is also with the Anhui Provincial Key Laboratory of High Performance Computing, Heifei, Anhui 230027, China. E-mail: {zzd1411, qycheng}@mail.ustc.edu.cn, {wangt635, louwenqi, leigong0203, xlanchen, cswang, xhzhou}@ustc.edu.cn.

Corresponding author: Teng Wang, Chao Wang



	Notation
Length of Input Token	L_t
Number of Heads	N_h
Dimension of Model	D_m
Dimension of Head	D_h
Dimension of FFN	D_f

Fig. 1: The execution process and basic framework of LLMs.

FPGAs have been widely adopted to deploy various neural network accelerators [25]–[29]. Most existing FPGA-based LLM accelerators are co-designed alongside algorithmic optimizations; however, they are primarily deployed on single devices [31]–[34]. Simply replicating such designs for large-scale deployment in data centers introduces several challenges. First, complex scheduling software is required to implement any form of parallelism. Second, due to the tightly coupled hardware-software co-design, the internal computing processes of these accelerators are typically difficult to modify, making it challenging to overlap computation with communication to reduce multi-device synchronization latency.

Recent efforts, such as [48], [49], have introduced multi-FPGA acceleration frameworks specifically tailored for LLM inference, incorporating co-design of synchronization and computation to achieve low-latency performance. However, these solutions still lack deep coupling between synchronization and computation stages, resulting in synchronization latency remaining a significant contributor to overall inference time. In addition, their tiling schemes and compute intensity in architecture design remain sub-optimal.

To address these limitations, this paper proposes LORA, a low-latency recurrent architecture based on multi-FPGA to accelerate LLMs. In response to the storage limitation issue faced by single-device deployment, we use the model parallel algorithms to split the LLMs and deploy them to multiple FPGA devices. Secondly, we set more reasonable synchronization timings and deeply couple the communication and computing processes to reduce inference latency. Then, to address the low utilization problem faced by GPUs and existing FPGA accelerators, we design an accelerator with an input batch size of one by using recurrent structures and deploy it on each device of the platform. Finally, we apply customized optimization techniques to enhance the deployment platform and perform modeling to analyze and determine the optimal architectural parameters for different computational tasks. Our contributions are summarized as follows:

- We optimize the synchronization timing required for running the LLMs on multi-FPGA platforms to reduce the storage, computation, and BRAM consumption. (Section III)
- We use customized optimization methods such as overlapping pipeline, packaging input data, and early communication with reordered computation to reduce synchronization latency. We further accelerate the synchronization steps at the link layer through the Router module. (Section III&IV)
- We use customized tiling schemes and recurrent structures to improve the data reuse rate and increase the computational intensity to accelerate inter-matrix and vector-matrix multiplications in a low-latency manner. (Section IV)
- We employ techniques such as DSP packing to optimize the deployment platform. Then, we model the resource consumption and the inference latency of our platform. Finally, we leverage a genetic algorithm to explore the design space for optimal architectural configurations under diverse workloads. (Section V)
- We propose two LORA prototypes to accelerate the modern LLMs on a multi-FPGA platform. Evaluation results demonstrate that LORA achieves a lower latency acceleration effect than the GPU and the previous multi-FPGA appliances. (Section VI)

II. Background And Motivation

This section presents the background of LLM architectures, state-of-the-art (SOTA) multi-FPGA LLM acceleration frameworks, and the motivations of our work.

A. Large Language Models

LLMs first perform a series of preprocessing steps on input data, such as token embedding and positional encoding. However, these steps consume minimal storage and computational resources and are often omitted from discussions. Modern LLM architectures are mainly composed of Decoder-only structures, such as GPTs [36], [37], LLaMA [1], PaLM [3], and Bloom [4], etc. In addition to the foundational Multi-Head Attention (MHA) Transformers, architectural extensions including Mixture-of-Experts [38], Multi-Query Attention [39], and Grouped-Query Attention [40] have been introduced recently. However, the above architectures generally consist of similar operators and are easy to expand. Therefore, without loss of generality, this paper focuses on optimization techniques for Decoder blocks within GPT-style Transformers.

As illustrated in Fig. 1b, the inference process of most LLMs can be divided into two stages: Prefill and Decode. In the Prefill stage, all input tokens are processed in a single pass to produce the first output token. The Decode stage then executes the model iteratively, where each iteration takes the previously generated token as input and outputs the next token. This process continues until a termination token is produced or the upper limit is reached. Therefore, inter-matrix multiplications account for most of the workload in the Prefill stage, whereas vector-matrix multiplications dominate the Decode stage.

As shown in Fig. 1c, a Decoder Block primarily consists of four modules: LayerNorm, Self-Attention, Residual, and Feed-Forward Network (FFN). The Self-Attention module is further composed of the QKV Generation, MHA, and Fully-Connected (FC) layers. The QKV Generation layer is responsible for producing the Query (Q), Key (K), and Value (V) matrices. For each head, the computation performed by the MHA layer is defined by the formula:

$$MHA(Q,K,V) = SoftMax(QK^T/\sqrt{D_m})V \tag{1}$$

Following the computation order defined in Eq. 1, we denote the two matrix multiplications as QmulKT and SmulV, respectively, where the matrix S represents the output of SoftMax. Additionally, we refer to the FC within the Self-Attention module as AttentionFC. The FFN module comprises two FC layers, FC0 and FC1, with a GELU layer applied between them.

While the explosive growth in model parameters has delivered unprecedented performance gains, it also imposes substantial computational and storage burdens on deployment platforms. To reduce the deployment costs of LLMs, quantization is widely adopted due to its effectiveness in enhancing computational performance while reducing memory requirements. As shown in Fig. 1d, a typical quantization scheme [41] processes different modules within a Decoder block with two precision formats. Most computations of Self-Attention and FFN modules are performed using integer precision, while operations such as SoftMax and LayerNorm are executed in half-precision floating-point format. In addition, several Quant and Dequant units are introduced to facilitate precision conversion.

B. Multi-FPGA LLM Acceleration Framework

Although optimization algorithms can significantly alleviate the pressure of single-device deployment, the continued growth in model parameters and the substantial storage requirements of the KV Cache make multi-device deployment an unavoidable solution for serving LLMs. For low-latency LLM inference acceleration tasks, SOTA multi-FPGA synchronization frameworks [48] primarily focus on two components: a model partitioning scheme and a synchronization strategy. The model partitioning scheme determines how the parameters of each module are distributed across multiple devices. The synchronization strategy specifies when and how communication is performed, which we refer to as synchronization timing and synchronization steps, respectively.

As shown in Fig. 2, the DFX framework adopts a tensor-parallel partitioning scheme, in which a single operator is decomposed into smaller components and distributed across multiple devices. Specifically, it partitions all FC and QKV Generation weights along the column and head dimensions,

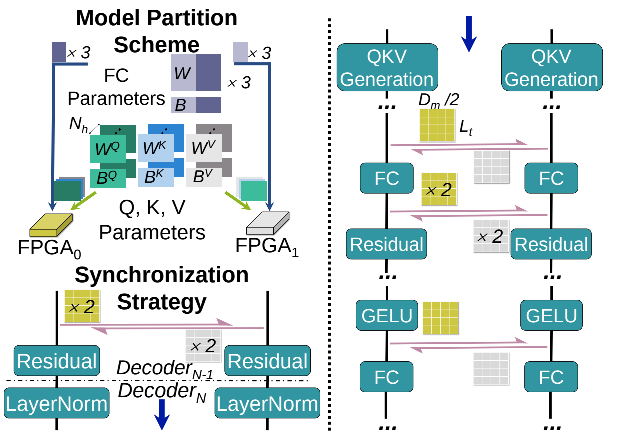
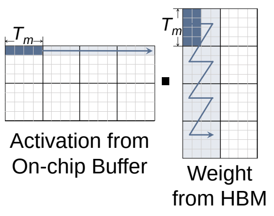
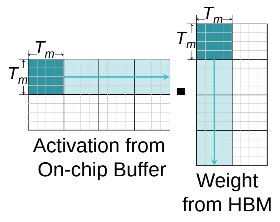


Fig. 2: DFX's model partition and synchronization schemes on two FPGAs.



(a) $CTC_{Weight} = 1$.



(b) $CTC_{Weight} = T_m$.

Fig. 3: Different tiling schemes for matrix multiplication.

respectively, while LayerNorm parameters are fully replicated across devices. Based on this partitioning, DFX performs activation matrix synchronization before the Residual, AttentionFC, and FC1 layers, totaling four all-gather synchronizations. As for the operation steps, computation is completely stalled during synchronization.

C. Motivation

SOTA frameworks for Multi-FPGA accelerated LLMs exhibit bottlenecks in inter-FPGA synchronization schemes and computational performance.

a) Synchronization Schemes: The first issue with the synchronization timing choice in DFX is the unnecessary waste of storage, computation, and BRAM resources. Take the first synchronization timing in Fig. 2 as an example—it is placed before the Residual operation, whereas the layer that actually requires the globally synchronized data is the subsequent QKV Generation. As a result, both the intermediate Residual and LayerNorm operations are forced to run on globally synchronized data. In a four-device cluster, an ideal synchronization scheme would reduce the storage requirement for LayerNorm parameters by 4× and accelerate the execution of Residual and LayerNorm by 4× compared to single-device deployment. However, in the four-device DFX configuration, the storage requirement is 4× higher, and neither Residual nor LayerNorm computation benefits from acceleration.

Although the storage and computational costs of Residual and LayerNorm are relatively minor in the entire LLM, a more significant issue lies in BRAM consumption. In a recurrent architecture, performing Residual and LayerNorm on globally synchronized data requires allocating 4× BRAM for intermediate buffering compared to an ideal design. Since this buffer

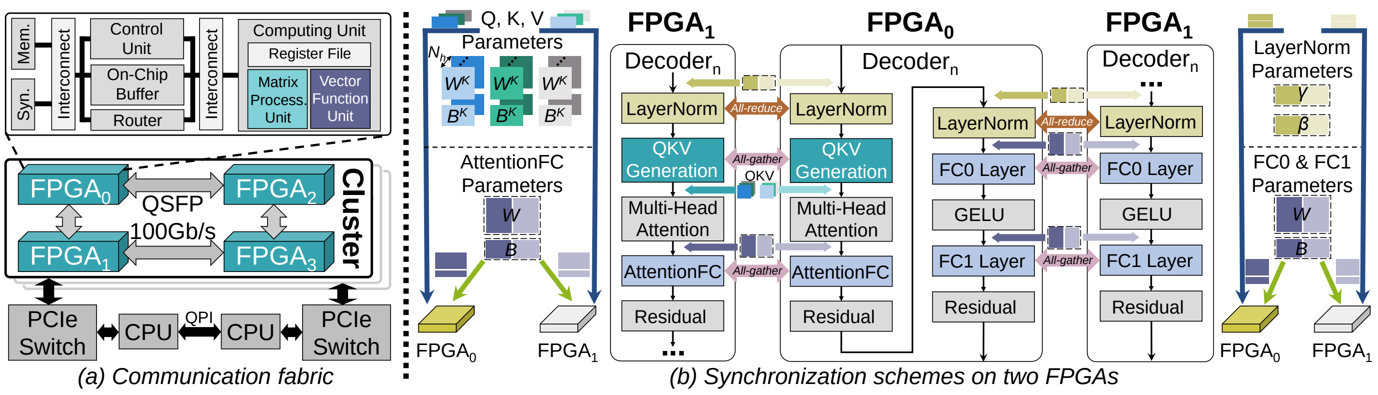


Fig. 4: The communication fabric and synchronization scheme designs of LORA.

is tightly coupled with the module-level parallelism, the excessive BRAM usage—three times more than necessary—can reduce the achievable parallelism to as low as 25% of the ideal in the worst case. Our final experimental results show that the buffer used for caching intermediate variables consumes more than 60% of the total BRAM resources consumption in LORA on average. This highlights a fundamental limitation of DFX-style synchronization timing.

DFX adopts a synchronization step that stalls computation entirely. Based on one of the experimental cases provided, synchronization accounts for approximately 17% of the total latency. Leveraging the highly customizable nature of FPGAs, we argue that such latency can be effectively masked through fine-grained pipelining.

b) Computational Performance: The accelerating architecture of DFX is mainly designed for the Decode stage, and its tiling scheme is shown in Fig. 3a, which only optimizes vector-matrix multiplication. To use this scheme to carry out inter-matrix multiplication (Prefill stage), we need to split the activation matrix into vectors by rows and then multiply them with the weight in turn. This will lead to a low computation-to-communication (CTC) ratio. When using the tiling scheme in Fig. 3b to perform the Prefill stage, the CTC will increase by $T_m\times$, which means the inference latency can theoretically be reduced up to $T_m\times$.

What's more, DFX uses a multiply-add tree in the half-precision floating point with a size of 64×16. The DSP strategy used requires the multiplier to complete the calculation in six cycles and the adder in eleven cycles. By optimizing multipliers and adders and using a systolic array, we can significantly reduce the latency of computing matrix multiplication.

III. Synchronization Schemes Design

A multi-FPGA LLM acceleration platform must first establish its hardware-level communication architecture, including interconnect topology and communication protocols. In this work, we adopt the communication architecture proposed in DFX [48], as illustrated in Fig.4a. We focus on a deployment platform consisting of four homogeneous FPGA boards within a single cluster. Once the hardware interconnect is defined, the next step is to design the cluster-level synchronization strategy, including model parameter partitioning, synchronization timing, and synchronization steps. This section presents a detailed discussion of these three components.

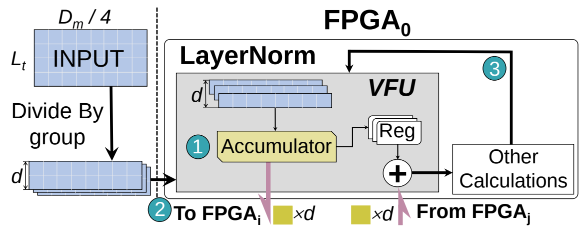


Fig. 5: Synchronization steps for the LayerNorm layer.

A. Model Partition and Synchronization Timing

Since our acceleration platform targets low-latency inference with a small number of input tokens, we adopt tensor parallelism for model partitioning. Ideally, this approach reduces the per-device storage by a factor of N in an N-card cluster compared to a single-device deployment. For simplicity, Fig. 4 illustrates the synchronization scheme between only two FPGA boards. Under this scheme, except for the GELU layer parameters, which remain fully replicated, the weights and biases of all other modules experience an $N\times$ reduction in per-device storage. Specifically, the parameters of the QKV Generation layer are partitioned across devices along the head dimension, while those of the LayerNorm and FC layers are partitioned along the column dimension.

After reshaping the outputs of the Self-Attention and FC layers, each device holds only a subset of the data corresponding to specific columns of the model. Consequently, subsequent LayerNorm, QKV Generation, and FC layers cannot compute their full outputs from local data alone. To initiate synchronization only when global data is required, communication is triggered only when the computation reaches the LayerNorm, QKV Generation, or FC layers.

Our design requires a total of four all-gather and two all-reduce operations per decoder block. To ensure that these additional operations do not increase the overall latency, we subsequently introduce optimization methods to hide communication latency behind computation.

B. Synchronization Steps

There are two types of operations that require synchronization in the Decoder block during the Prefill stage: the summation of row vectors and the matrix multiplication. We introduce four customized optimization methods for these two operations to reduce the synchronization latency.

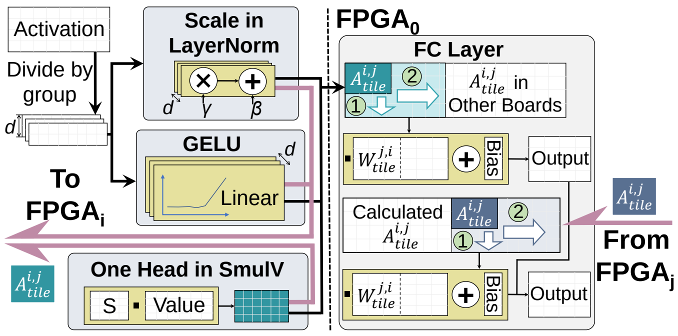


Fig. 6: Synchronization steps for FC layers.

a) Summation: When the input matrix of LayerNorm is partitioned across devices along the column dimension, it is necessary to initiate all-reduce communication for the summation on the row vector. Our **first optimization method** is to overlap the computation and communication as shown on the right side of Fig. 5. The entire pipeline of LayerNorm can be roughly divided into three stages: ① Each device computes a partial sum using its local data. ② These partial sums are immediately transferred across devices using all-reduce communication. ③ Once all devices have received the aggregated global sum, each completes the remaining computations.

However, this distributed computing requires frequent small-scale communications, which suffer from substantial latency overhead. As demonstrated by our experiment in Section V, the Aurora IP core [51] introduces a link latency of approximately 300ns, which is prohibitively high for transmitting small-scale data. To address this issue, our **second optimization method** is to package the input row vectors, as illustrated on the left side of Fig. 5. Specifically, during processing, we package every d consecutive input rows into a group. The rows in the same group are processed in parallel, and the pipelined algorithm is applied across groups. Moreover, since the bit width of the streaming port in the communication IP core is typically much larger than the data width used in LLM computations, this grouping strategy also improves link bandwidth utilization by aligning the data size with the communication interface.

We assume that the bit width of the stream port in the communication IP kernel is $d\times$ greater than the model's bit width, and the number of rows in a group is also d. Let N_{acc} denote the number of accumulators (using Multiply-Accumulate trees to calculate sums). Given an input matrix of size $L_t \times D_m$ and a transmission duration of K cycles per transaction. Taking the mean calculation in two FPGAs as an example, the total latency without grouping is:

$$Max(\frac{D_m}{N_{acc}} + log_2 N_{acc}, K) \times L_t + K \quad (2)$$

When d rows of input data are grouped and processed together, the total latency becomes:

$$Max(\frac{d \times D_m}{N_{acc}} + log_2 N_{acc}, K) \times \frac{L_t}{d} + K \quad (3)$$

The grouping scheme increases the computational load per transmission, thus making it more likely for the computation latency to overlap and hide the communication latency, as reflected in the Max term in Eq.3.

b) Matrix Multiplication: We take the processing of the FC layers as an example. Under the aforementioned model partitioning and synchronization timing scheme, both the Activation (A) and Weight (W) matrices are partitioned across devices along the column dimension. To reduce the storage burden of weights within the cluster, we perform an all-gather synchronization on the A matrix to complete the required computation. The total latency for processing the FC layers can be expressed by the following simplified formula:

$$\sum_i \sum_j Max(Cal(A_{tile}^{i,j}, W_{tile}^{j,i}), Load(A_{tile}^{i-1,j})) \quad (4)$$

Here, each summation term (i,j) accounts for the time to process tile (i,j) of matrix A together with tile (j,i) of matrix W. $Cal(\cdot)$ denotes the compute latency of the tile-wise matrix multiplication, and $Load(\cdot)$ denotes the latency to fetch a tile of A. When the tile is local, $Load(\cdot)$ is the local memory-access time. When the tile is remote, $Load(\cdot)$ is the synchronization latency to transfer that tile. Our optimization goal is to restructure the pipeline to hide synchronization latency, reducing the impact of the $Load$ term within the Max function for each tile-wise computation.

In conventional implementations, the pipeline initiates both $Cal_{i,j}$ and $Load_{i-1,j}$ concurrently at the start of computing tile (i,j). However, since the A matrix is generated prior to FC execution, the $Load_{i-1,j}$ operation can be triggered earlier. As shown on the left side of Fig. 6, our **third optimization method** is to initiate data transfer as early as possible. Once the preceding layer completes computations on the A matrix, the corresponding all-gather step is immediately launched. For instance, in the case of the AttentionFC layer, its preceding layer is SmulV. Upon finishing the computation for a single attention head in SmulV, each FPGA board can mark, package, and broadcast its results to the other boards.

Our **fourth optimization method**, illustrated on the right side of Fig. 6, involves reordering the loop indices in Eq. 4. While the conventional order $\sum_i \sum_j$ implies row-major traversal of A, our column-wise partitioning motivates a switch to $\sum_j \sum_i$, or column-major traversal. This modification allows each board to first complete all computations involving locally available A tiles before depending on remote data from peer boards. As a result, the latency contribution of the $Load$ operation is further suppressed within the Max expression, yielding a more efficient pipeline.

IV. Architecture Design of LORA

The overall architecture of LORA is shown in Fig. 7, comprising five key components: Peripherals, Control Unit, On-chip Buffer, Router, and Computing Unit. Peripherals are primarily responsible for storing model parameters and providing interfaces for inter-device synchronization. The On-chip Buffer bridges the gap between computing throughput and memory bandwidth, which also supports the partial implementation of KV Cache. The Control Unit manages data flow within the device, handling both inter-device synchronization and local computations. The Router performs link-layer optimizations for communication tasks dispatched by the Control Unit. The Computing Unit executes matrix and vector

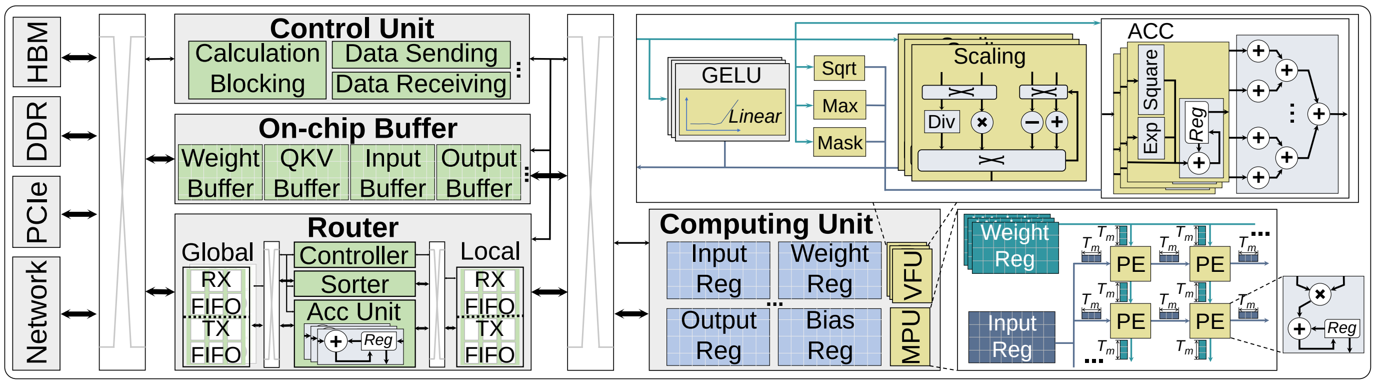


Fig. 7: The overall architecture of LORA.

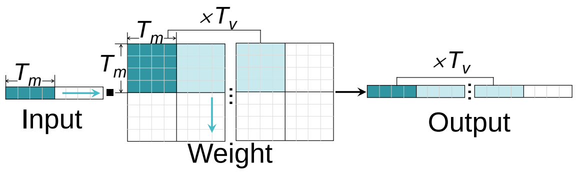


Fig. 8: The tiling scheme of vector-matrix multiplication.

operations as assigned by the Control Unit. This section details the architectural design of the Control Unit, Computing Unit, and Router.

A. Control Unit

The Control Unit is composed of two functional modules: inter-device synchronization and local computation management. The inter-device synchronization module is responsible for initiating communication events across FPGA devices. It also enforces computation stalls when necessary to ensure data consistency during synchronization. The local computation module will design appropriate tiling schemes for different computing loads, that is, the order in which data is taken out from the peripherals and filled into the Computing Unit, and how to organize the computation results and write them back.

a) inter-device synchronization: This module implements the corresponding hardware logic for the aforementioned synchronization schemes. It formulates the timing, payload, and destination of each device's send/receive operations, and delegates the actual data transmission to the Router module for link-layer execution. Given that the data flow of a specific LLM is fixed at compile time, the control logic remains relatively straightforward and introduces minimal overhead. We design three key sub-modules: Data Sending, Data Receiving, and Computation Blocking.

The Data Sending module monitors the progress of the Computing Unit. When computation reaches a designated stage, the corresponding output data is packaged, tagged, fetched from the on-chip buffer, and pushed to the stream write interface. The Data Receiving module is activated when the computing process requires external input. It continuously monitors the stream read interface; once valid data arrives, it is unpacked, decoded, and stored into the appropriate on-chip buffer. The Computation Blocking module supervises the state of the Computing Unit and stalls execution when required input data is not yet available from the Data Receiving module.

b) Local computing control: The functions of this module primarily encompass two aspects. First, it implements in hardware the modification to the computation order proposed in the fourth optimization strategy of Section III. Second, it handles tiling, a widely adopted technique in FPGA-based accelerators. We design distinct tiling schemes tailored to different computational loads in order to maximize data reuse and minimize overall processing latency.

For inter-matrix multiplication, we employ the tiling scheme illustrated in Fig.3b, where large matrices are partitioned into smaller tiles of size $T_m \times T_m$. These tiles serve as the basic computation unit, and the activation and weight matrices are traversed in column-major and row-major order, respectively. The input to the Computing Unit consists of two $T_m \times T_m$ matrices per processing cycle.

For vector-matrix multiplication workloads, we adopt the scheme shown in Fig.8. During the Decode stage, the maximum CTC ratio is 1, requiring large amounts of weight data to be loaded in each processing cycle to maintain computational intensity, which imposes substantial memory bandwidth demands. To address this, most LLM weight parameters are stored in High Bandwidth Memory (HBM) [53]. In each processing cycle, the system loads a vector of dimension T_m along with T_v adjacent tiles of size $T_m \times T_m$ into the Computing Unit.

B. Computing Unit

The Computing Unit consists of the Matrix Processing Unit (MPU) and the Vector Function Unit (VFU), which are responsible for processing the data loaded by the Control Unit. Given that the design objective is to achieve low-latency processing of LLMs with a batch size of one, we adopt a recurrent architecture to optimize for this use case. The primary workloads in LLM inference include inter-matrix multiplications during the Prefill stage and vector-matrix multiplications during the Decode stage. Both types of computations are handled by reusing the MPU. Meanwhile, operations such as vector scaling, exponential computations, and other element-wise vector operations required by layers like LayerNorm and SoftMax are delegated to the VFU.

a) Matrix Processing Unit: The MPU is primarily composed of an $D_{sa} \times D_{sa}$ array of Processing Elements (PEs) organized into a two-dimensional systolic array (SA). In this architecture, activation data and weights propagate horizontally and vertically, respectively, with the output accumulated inside each PE. Each PE consists of a multiplier, an adder, and an accumulation register. For inter-matrix multiplication, the MPU receives two $T_m \times T_m$ matrices as input, which are processed using the classic systolic array computation pattern, as shown in Fig.3b. For vector-matrix multiplication, the input comprises a single T_m-dimensional vector and T_v distinct $T_m \times T_m$ matrices, as illustrated in Fig.8. In order to reuse the MPU, we reinterpret the 2D SA as multiple one-dimensional SAs of length T_m, where each 1D SA receives the same horizontal vector input, and each vertical input corresponds to one of the T_v different matrices. This reuse is achieved by augmenting the original 2D SA with additional data paths.

Compared to broadcast arrays and multiply-add trees, SAs offer more regular structures, better scalability, and superior timing characteristics, making them well-suited for FPGA deployment. Nevertheless, the inherent fill and drain latency of SAs can introduce performance bottlenecks, which we mitigate through the use of streaming interfaces, double buffering, and instruction-level overlapping techniques.

b) Vector Function Unit: If the input is a matrix, we first split it into several vectors by rows and process them independently. The core component of this unit is the Accumulation module. Each accumulator integrates an exponentiation unit, a square root unit, a multiplier, and an adder. The partial results from all accumulators are then aggregated using a cumulative binary tree structure. For element-wise vector operations, the Scaling module in the VFU supports basic arithmetic operations, including multiplication, division, addition, and subtraction between vectors and scalar variables. Given the independence of data elements in these operations, we apply loop unrolling to further enhance parallelism and throughput. Other specialized functions, such as Masking, GELU, and square root operations, are implemented using established methods: GELU employs piece-wise linear approximation as described in [54], while the square root function leverages Xilinx's IP core libraries [55].

c) Supports for Quantization: To support SmoothQuant-O3 [41], which adopts per-tensor static quantization under the W8A8 format, we introduce two major enhancements to the Computing Unit. First, we add support for Quant, Dequant, and the so-called "quantization difficulty migration" operations. These operations are essentially vector-wise scaling and can be efficiently handled by the existing Scaling module in the VFU. These steps can be fused with adjacent operations; for example, quantization following LayerNorm can be merged into the final scaling stage by adjusting the scale factor, thereby incurring negligible additional cost.

Second, we extend the MPU to support INT8 precision multiplication. This enhancement is straightforward using standard HLS tools [63], [64] and enables resource-efficient design since INT8 multipliers require far fewer DSP slices than FP16. Consequently, we can enlarge the systolic array size. However, the expansion is bounded by the peak HBM bandwidth and the width of the HBM interface, which impose different array size limits in the Decode and Prefill stages, respectively.

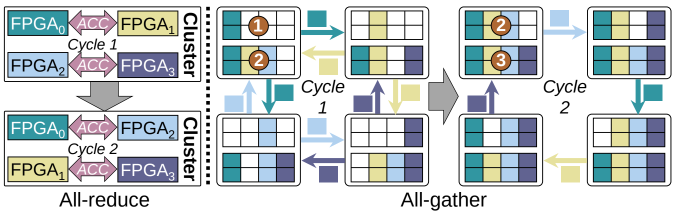


Fig. 9: Communication steps of all-reduce and all-gather.

C. Router

Fig. 7 presents the overall architecture of the Router. The Router receives synchronization tasks and associated data from the Control Unit, and manages the Aurora 64b/66b IP kernel [51] to facilitate inter-device communication by reading from and writing to stream ports. The configuration details of the Aurora IP core are deferred to Section V.

For all-reduce communication, each device first computes a partial sum from local data and then participates in a cluster-wide synchronization step. This process is illustrated on the left side of Fig. 9, where the Router contains an ACC Unit composed of an adder pair and accumulator registers to support this operation.

For all-gather communication, it takes one cycle in full-duplex and one more cycle in one-way transmission to complete the synchronization of a single data block, as shown on the right side of Fig. 9. Given that the Aurora 64b/66b IP can work in full-duplex mode and the order of one-way transmission and full-duplex cycles can be interchanged, we optimize bandwidth utilization by merging one-way transmission cycles from two consecutive blocks—achieving two-block synchronization in just three cycles. Additionally, a Sorter Unit is embedded in the Router to reorder the received data into the correct positions in the local cache. All control flows within the Router are orchestrated by the Control Unit through a fixed state machine design.

V. Implementation of LORA

Due to their high degree of customization, low latency, and rapid development cycle, we select FPGAs to implement LORA. This section first introduces three optimization techniques tailored to the FPGA platform. Furthermore, to fully utilize on-chip resources such as DSPs and BRAMs for different task characteristics and achieve optimal low-latency prototypes, we conduct performance modeling and analysis of our acceleration framework.

A. DSP Packing for Quantized Multiplications

Supporting the W8A8 quantization algorithm requires handling the multiplication of 8-bit input activations and weights. However, the DSP blocks in FPGAs have fixed input bit-widths that are significantly wider than 8 bits. For instance, the DSP48E2 block supports the computation of $P = (A + B) \times C + D$, where both A and B are 27-bit operands, C is an 18-bit operand, and the product P is 45 bits wide.

To improve DSP utilization, we employ an efficient packing and unpacking strategy for activations, weights, and products. Following the approach presented in [56], [57], two 8-bit activations are packed into the high 8 bits of operand B and

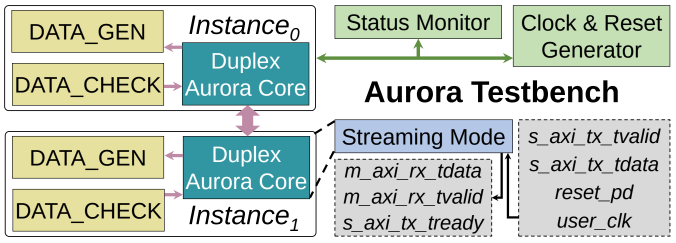


Fig. 10: The testbench and configuration mode of Aurora.

the low 8 bits of operand C, respectively. The remaining bits of B and C are filled with zeros and the sign bit of the corresponding activation. Each 8-bit weight is stored in the low 8 bits of operand A, with upper bits padded with zeros. This configuration allows two independent multiplication results to be extracted from the high and low 16 bits of the product P.

B. Communication IP Kernel Configuration

We select Xilinx's Aurora 64B/66B communication IP [51] to manage all transmission operations between FPGAs. First, we configure the IP core to operate in stream and full-duplex mode, as illustrated in Fig. 10. This configuration allows us to ignore the initialization phases of the IP core when evaluating link latency.

Secondly, according to Xilinx's official recommendations [51], reducing the line rate by approximately 50% from the maximum supported by the transceiver and IP core can improve timing performance and eliminate the need for additional synthesis optimizations. Accordingly, in our final deployment, we configure each Aurora IP core with four lanes and set the line rate to 12.8 Gbps.

Finally, since the schemes proposed in Section III frequently trigger small-scale data communication, it is necessary to verify that the Aurora 64B/66B IP core introduces minimal link latency. As shown in Fig. 10, we instantiated two Aurora 64B/66B IPs on the U280 device. Vivado simulation results show that the link latency of the Aurora core is approximately 300 ns, which is negligible compared to the computation latency.

C. Hybrid Memory System Utilization

Modern FPGA devices typically feature a hybrid memory architecture combining HBM and DDR, where HBM provides significantly higher bandwidth but limited capacity compared to DDR. To leverage this architecture, we optimize the placement strategy for various model parameters across the hybrid memory system. Specifically, we store all parameters required for the Prefill stage and small single-access data structures (e.g., GeLU lookup tables and SoftMax parameters) used in the Decode stage in DDR, while allocating the weights for the Decode stage to HBM.

As discussed previously, the Prefill stage of LLM inference primarily involves inter-matrix multiplications, which exhibit relatively low memory bandwidth demands. In contrast, the Decode stage is dominated by vector-matrix multiplications, which are highly bandwidth-sensitive. Applying the tiling schemes introduced in Section IV and configuring the systolic array to a size of 32×32, the theoretical peak bandwidth requirement for the Prefill stage at 200 MHz is approximately

TABLE II: Symbol Definitions.

Notation	Definition
$\mathcal{R}_*$	The BRAM consumption of the buffer $*$
T_m	The tiling size of the matrix, as shown in Fig. 3b
D_{sa}	The dimension of the systolic array
P_w	The partition factor of the weight parameters
N_{data}	The number of data transmitted per cycle
N_{tran}	The number of transmission cycles that can be buffered
N_{acc}	The number of ACC units in the VFU
$\mathcal{D}_*$	The DSP cost of the unit $*$
Ψ_*	The DSP cost of the operation $*$
$\mathcal{L}_{buf}$	The on-chip buffer access latency of each iteration
$\mathcal{L}_{cal}$	The computation latency of each iteration
CNT	The loop count outside each iteration
$\mathcal{L}_{axi}$	The DRAM or HBM access latency of each iteration
$\mathcal{L}_{syn}^i$	The synchronization latency of the i^{th} tile in stage II
$\mathcal{L}_{link}$	The link latency
$Rate$	The transfer rate of the communication IP

11 GB/s-well within the capabilities of DDR memory. However, the Decode stage demands around 190 GB/s, which necessitates the use of HBM.

Our hybrid memory management strategy maximizes HBM availability for storing model weights, thus enabling the deployment of larger LLMs. Additionally, for models with smaller parameter footprints, this strategy supports longer output sequences. Since the storage requirement for KV Cache grows rapidly with output length, and HBM offers sufficient bandwidth for KV access, a substantial portion of the KV Cache can be offloaded from on-chip URAM to HBM.

D. Design Space Exploration

To accommodate deployment tasks with varying computational characteristics, the allocation strategy for on-chip FPGA resources must be carefully adapted. For instance, INT8 multiplications consume significantly fewer DSP resources compared to FP16 operations. When handling INT8-precision inference tasks, LORA's MPU can save a substantial number of DSP blocks. However, due to the limitations of HBM bandwidth, we cannot increase MPU parallelism to improve DSP utilization. In this case, the remaining DSPs can be allocated to the VPU to enhance its parallelism. Nevertheless, increasing VPU parallelism rapidly exhausts on-chip buffer resources, potentially reducing the maximum supported token length for KV Cache.

To obtain the optimal deployment scheme under varying input token lengths and model parameters while reducing manual efforts, we implement a performance model.

a) Resource Consumption: For commonly used resources in FPGAs, our modeling focuses on the consumption of BRAM and DSP. The BRAM consumption mainly arises from three components: on-chip buffers, buffers for AXI interfaces, and buffers used by the Router. For on-chip buffers, we take the Weights Buffer in the FFN computation as an example, assuming the data type to be FP16. The BRAM consumption is:

$$\mathcal{R}_{Weight} = \lceil \frac{16b}{18b} \rceil \times \lceil \frac{T_m^2}{1024 \times P_w} \rceil \times P_w \times 2 \quad (5)$$

where the multiplication by 2 accounts for the use of double buffering. We use the reshape mode of BRAM to support the buffers for the Router and the AXI interfaces. Below is the BRAM consumption:

$$\mathcal{R}_{Router/AXI} = \lceil \frac{N_{data} \times 16b}{36b} \rceil \times \lceil \frac{N_{tran}}{512} \rceil \tag{6}$$

The DSP utilization primarily originates from the MPU and the VFU. We take the ACC unit in VFU as an example. It consists of N_{acc} sets of Exp, Square, and Add operators, as well as an accumulation binary tree with an input size of N_{acc}. Therefore, the DSP consumption of the ACC is given by:

$$\mathcal{D}_{ACC} = (\Psi_{exp} + \Psi_{square}) \times N_{acc} + \Psi_{add} \times (3 \times N_{acc} - 1) \tag{7}$$

The exact values of Ψ_* depend on the implementation configuration.

b) Inference Latency: The inference latency analysis of the Decoder block can be categorized into two cases based on whether synchronization is required. For layers that do not involve synchronization, the inference latency primarily consists of the access latency of on-chip buffers and the computation latency of the computing unit. Given the adoption of double buffering, the inference latency of these layers is expressed as:

$$\mathcal{L} = \mathcal{L}_{buf} + Max(\mathcal{L}_{buf}, \mathcal{L}_{cal}) \times CNT + \mathcal{L}_{cal} \tag{8}$$

For layers that require synchronization, $\mathcal{L}$ is further increased by the access latency of off-chip data reads or writes and the communication latency needed for synchronization. We take the processing of the FC0 layer in the Prefill stage as an example. The MPU in each device is required to perform a matrix multiplication between matrices of size $L_t \times D_m$ and $D_m \times D_f/4$. According to the customized strategies outlined in Section III, the processing can be divided into four stages based on the source of activations. The first $L_t \times D_m/4$ activation block is sourced from the on-chip buffer, while the remaining activations are obtained via synchronization from the other three devices. The corresponding weights are entirely loaded from the local HBM.

In each sub-iteration of the four stages, the MPU performs a matrix multiplication between two $T_m \times T_m$ matrices with a parallelism of $D_{sa} \times D_{sa}$. Meanwhile, each device loads the next $T_m \times T_m$ tiles of activations and weights into the MPU. Below, we only present the inference latency of FC0-related processing in stage II:

$$\mathcal{L}^{II} \approx Max(\mathcal{L}_{axi}, \mathcal{L}^{i}_{syn}, \mathcal{L}_{cal}) \times CNT \tag{9}$$

$$\mathcal{L}^{i}_{syn} = \mathcal{L}^{i}_{syn_{comm}} - \mathcal{L}^{i}_{syn_{cal}} \tag{10}$$

$$\mathcal{L}^{i}_{syn_{comm}} \approx \mathcal{L}_{link}[i == 0]^1 + \frac{T_m^2}{Rate} \tag{11}$$

$$\mathcal{L}^{i}_{syn_{cal}} = \mathcal{L}_{LN}(\frac{L_t \times D_m/4}{T_m^2} - i) + \mathcal{L}^{I} + \mathcal{L}_{cal} \times (i-1) \tag{12}$$

for simplicity, some steps are omitted, such as the preprocessing and postprocessing steps in Eq.9. Eq.10 represents the communication latency that cannot be hidden by computation. Eq.11 adopts the alpha-beta communication model [42] to approximate the communication latency.

The transmission of FC0-related activations is initiated immediately after the variance computation in the LayerNorm. Therefore, Eq.12 decomposes the overlapping compute latency during the synchronization of the i^{th} tile data into three parts: the remaining computation latency of the LayerNorm, the computation latency of stage I, and the stage II computation that may already have been completed.

¹The Iverson bracket [58]

TABLE III: Model configuration

LLM	Embedding Dimension	Head Dimension	FFN Dimension	Number of Layers
GPT-2 345M	1024	64	4096	24
GPT-2 774M	1280	64	5120	36
GPT-2 1.5B	1536	64	6144	48
OPT-6.7B	4096	128	16384	32
LLaMA2-7B	4096	128	11008	32

c) Exploration Algorithm: The search variables are the hardware configuration parameters, including the on-chip buffer size, parallelism degrees, systolic array size, the off-chip bandwidth of each interface, and other related parameters. In order to quickly obtain an optimal configuration, we first define a set of candidate values for each hardware parameter to prune the search space, and then we encode all the variables and use an evolutionary algorithm [54] to search with Eq.13 as the objective function.

$$\begin{gathered} Minimize \quad \mathcal{L}_{Decoder}(\mathcal{P}_{Decoder}, Net) \\ s.t. \mathcal{R}_*, \mathcal{D}_*, \mathcal{B}and\mathcal{W}idth \leq Limit \end{gathered} \tag{13}$$

Here $\mathcal{P}_{Decoder}$, Net denote the parameters of the Decoder block and workload, respectively. The search takes only a couple of seconds on the CPU.

VI. Evaluation

A. Evaluation Setup

a) Models and Datasets: We evaluate the effectiveness of LORA using five LLM models: GPT-2 345M, GPT-2 774M, GPT-2 1.5B, OPT-6.7B, and LLaMA2-7B [17], [19], [36]. GPT-2 models are evaluated on the LAMBADA dataset [59], while the OPT and LLaMA2 models are tested on the WikiText dataset [60]. To ensure comparability with DFX [48], we follow the same modification strategy as DFX by adjusting the number of attention heads in GPT-2 1.5B from 25 to 24. Table III summarizes the detailed configurations of the evaluated models.

b) Metrics: We perform a comprehensive latency-based evaluation of LORA and baseline accelerators. Latency is defined as the end-to-end time for completing a single inference pass and serves as the primary performance metric and optimization target in this work.

c) Platform: We use the AUS ESC8000A-E11 rack server [61], AMD EPYC 7543 CPU [62], and Alveo U280 data center accelerator devices [52] for evaluation. Each U280 board features 9,024 DSP slices, 2,016 BRAMs, and 960 URAMs. The memory hierarchy includes 8 GB of HBM2 with a peak bandwidth of 460 GB/s and 32 GB of DDR with a bandwidth of 38 GB/s. Every U280 FPGA is also equipped with two QSFP28 ports to enable high-speed interconnect.

d) FPGA Prototypes: We implement two LORA prototypes using Vitis 2022.2 [63] and Vivado 2022.2 [64]: LORA-F (FP16 precision [65]) and LORA-Q (W8A8 quantization [41]). Table IV summarizes the hardware parameters, performance, and resource utilization of several multi-FPGA

TABLE IV: Hardware parameters, performance, and utilization of multi-FPGA acceleration schemes.

Name	Freq (MHz)	Quantization Scheme	Latency (ms)	Speedup	Throughput (tokens/s)	LUT(K)	FF(K)	URAM	BRAM	DSP
LORA-F	200	FP16	678.4	**×2.28**	377.4	965(74%)	1170(45%)	864(90%)	1367(68%)	6792(75%)
- MPU	-	-	-	-	-	321(25%)	421 (16%)	0 (0%)	256 (13%)	6144(68%)
- VFU	-	-	-	-	-	69 (5%)	98 (4%)	0 (0%)	0 (0%)	648 (7%)
LORA-Q	250	W8A8	307.3	**×5.03**	833.1	683(52%)	964 (37%)	864(90%)	1265(63%)	4744(53%)
- MPU	-	-	-	-	-	255(20%)	318 (12%)	0 (0%)	0 (0%)	4096(45%)
[49]	245	W4A8	1329.1	×1.16	192.6	569(44%)	653 (25%)	111(12%)	389 (19%)	1780(20%)
DFX	200	FP16	1546.8	×1	165.5	520(40%)	1107(42%)	104(11%)	1192(59%)	3533(39%)

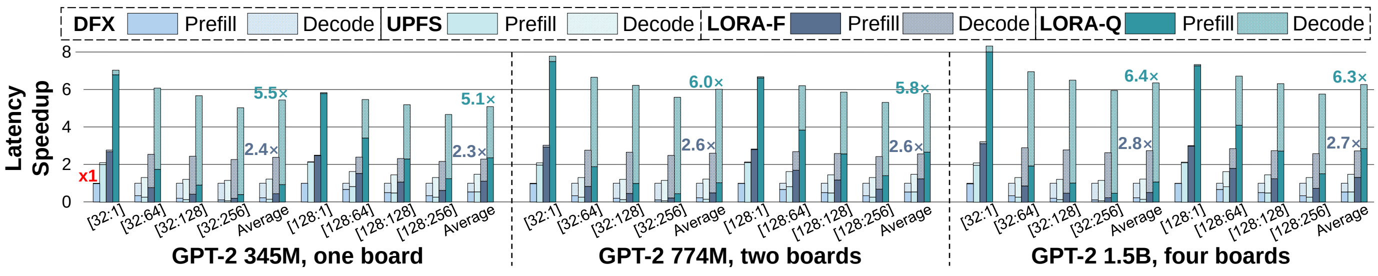


Fig. 11: Inference latency for SOTA, LORA-F, and LORA-Q when deploying GPT-2 345M, GPT-2 774M, and GPT-2 1.5B on U280s, with input and output token lengths denoted as [Input Size: Output Size].

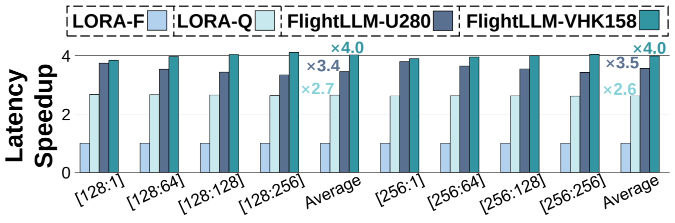


Fig. 12: Inference latency of FlightLLM and LORA when deploying LLaMA2-7B on a single board.

schemes for GPT-2 345M, and the input/output token lengths are fixed at 32/256. In LORA-F, the degree of parallelism is constrained by DSP availability, while in LORA-Q, it is bounded by the HBM bandwidth. All 864 URAMs are dedicated to KV Cache buffering, enabling support for up to 256 output tokens. For sequences exceeding this threshold, excess KV data is offloaded to HBM.

e) SOTA Accelerator Baselines: SOTA frameworks can be categorized into two groups: those supporting only single-device acceleration and those enabling multi-device acceleration. To evaluate LORA's effectiveness in single-device scenarios, we use FlightLLM [23], DFX, and [49] as the baseline; for multi-device scenarios, we only compare against DFX and [49]. Due to limitations in available data and open-source implementations of SOTA frameworks, we compare LORA with FlightLLM on the LLaMA2-7B model, and conduct performance evaluations of LORA, DFX, and [49] on the GPT-2 model.

f) GPU Baselines: We use the NVIDIA V100 GPU [66] as the baseline for our GPU-based evaluation, running GPT-2 models with the GPU-optimized Megatron-LM framework [67] and CUDA Toolkit 11.1, which supports both multi-GPU training and inference. Specifically, we evaluate the 345M model from NVIDIA Megatron-LM [67] and the 774M and 1.5B models from OpenAI [36].

B. Inference Accuracy

Compared to the NVIDIA V100 GPU implementation, the accuracy loss in our FPGA prototypes may result from two factors: precision deviation introduced by the Xilinx toolchain and the W8A8 quantization algorithm. To assess the toolchain's impact, we compare LORA-F with the PyTorch baseline on the GPT-2 345M model using the LAMBADA and CBT-CN datasets [68] under FP16 precision. Results show that LORA-F incurs only a 0.2% drop on LAMBADA and achieves a 0.11% gain on CBT-CN, suggesting the toolchain has a negligible impact on accuracy. The W8A8 quantization, adopted from SmoothQuant-O3 [41], preserves inference accuracy comparable to FP16 for the models evaluated.

a) Comparison with SOTA Accelerators: As shown in Fig. 11, on U280 running GPT-2-345M, LORA-F and LORA-Q achieve 2.40× and 5.45× average speedups over DFX [48]. At the stage level (Prefill/Decode), the speedups are 2.81×/2.30× for LORA-F and 7.12×/5.11× for LORA-Q. Compared with [49], the average speedups are 1.94× for LORA-F and 4.41× for LORA-Q, with stage-level gains of 1.30×/2.09× and 3.30×/4.64×, respectively. Fig. 12 further compares LORA with FlightLLM [23] on LLaMA2-7B: when deployed on U280, FlightLLM yields 3.4×/1.3× average speedups over LORA-F/LORA-Q; on the VHK158 [69] platform, the speedups are about 4.0×/1.5×.

Relative to SOTA multi-FPGA baselines, LORA's single-board architecture achieves substantial gains owing to our efficient compute core and flexible tiling schemes. On a single device, FlightLLM achieves lower latency; however, its design lacks multi-board scaling, limiting extension to multi-FPGA acceleration. Independent of platform effects, FlightLLM adopts more aggressive quantization and sparsity—approximately 3.5-bit weights and 8-bit activations.

LORA's speedup declines with increasing output length due to the growing KV Cache and the rising compute and

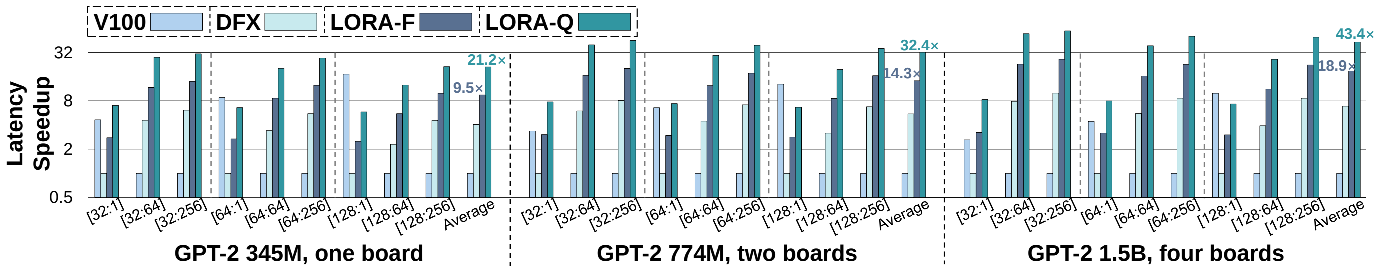


Fig. 13: Inference latency of V100 GPU, DFX, and LORA when deploying GPT-2 345M, GPT-2 774M, and GPT-2 1.5B on U280s.

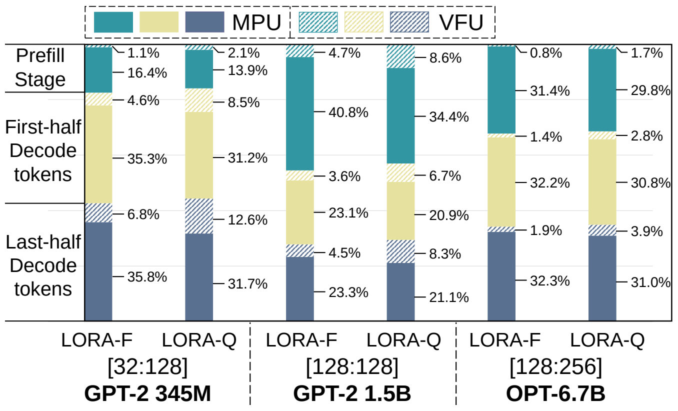


Fig. 14: Latency breakdown for different deployment schemes.

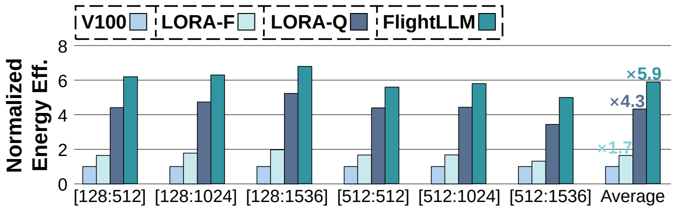


Fig. 15: Energy efficiency of V100, FlighLLM and LORA.

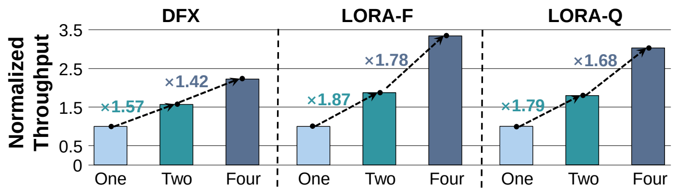


Fig. 16: Scalability of DFX and LORA on the 345M model.

bandwidth demands of MHA in Decode, which increase per-token latency. In contrast, previous SOTA designs maintain relatively stable per-token latency. This limitation will be addressed in future work. The decline is more pronounced for LORA-Q, as SoftMax remains in FP16, and its growing share of total latency reduces the benefit of W8A8 quantization at longer outputs.

Fig. 11 also compares the latency of multi-FPGA accelerators for GPT-2 774M and 1.5B models deployed across two and four U280 devices. In these scenarios, LORA achieves greater speedups over prior SOTA designs, mainly due to its more efficient inter-device synchronization, which will be detailed later.

b) Comparison with GPU: Fig. 13 compares the latency of various deployment schemes against the V100 GPU baseline, which only supports FP16 due to the lack of native INT8 $\times$ INT8 acceleration. Several observations emerge. First, for small output token lengths, the V100 shows significantly lower latency. As input length increases, the performance gap between GPU and FPGA widens. In the Prefill stage—dominated by matrix multiplications—the GPU's abundant parallel units enable faster processing. With longer inputs, its compute intensity grows, further extending its advantage, making FPGA latency higher in Prefill. Second, with fixed input length, increasing output tokens improves FPGA speedup, as the Decode stage dominates and the GPU faces bandwidth and compute limitations. Finally, as model and hardware scales grow, LORA achieves greater speedups by leveraging the larger share of Decode computation and its efficient inter-device synchronization, reducing overall latency.

c) Latency Breakdown: Fig.14 shows the latency breakdown across deployment schemes, revealing four key observations. First, inference latency is largely dominated by the MPU, while the VFU remains mostly idle. Second, as output token count increases, average per-token latency grows, and the VFU accounts for a larger share of total latency. This is due to the KV Cache mechanism and architectural design, where SoftMax grows faster than other MHA components—aligning with the speedup degradation in Fig.11. Third, the VFU latency ratio during the Prefill stage is lower for GPT-2 1.5B than for 345M, confirming that the higher speedup in the four-device deployment stems from LORA's efficient inter-device synchronization, rather than a larger MPU latency share. Finally, LORA-Q outperforms LORA-F when VFU overhead is smaller, as it specifically accelerates the MPU. This implies that, relative to GPT-2, LORA-Q delivers greater speedup on OPT or LLaMA2 models.

d) Energy Results: Fig. 15 compares the energy efficiency of deploying LLaMA2-7B on V100 GPU, FlightLLM (U280 version), and LORA. Relative to the V100, LORA-F and LORA-Q improve energy efficiency by about 1.7$\times$ and 4.3$\times$, respectively. Relative to LORA-F and LORA-Q, FlightLLM achieves 3.5$\times$ and 1.4$\times$ higher energy efficiency, respectively. With both systems on U280, FlightLLM and LORA exhibit similar power consumption, and the energy-efficiency gap mainly arises from compute-architecture performance.

e) Scalability: Fig. 16 shows the scalability of the U280 FPGA in DFX and LORA for the 345M model with 64:64

tokens, and each work reports normalized results for deployments on one, two, and four FPGAs. The performance of LORA-F and LORA-Q increases linearly with the number of FPGAs at the rates of 1.8 and 1.7, respectively. Compared with DFX, LORA exhibits better scalability. As the number of boards increases, LORA shows a tendency for scalability to deteriorate. This occurs because the per-board computational load decreases, and the share of hardware overhead, such as pipeline drain and startup latency, grows. For a small model like GPT-2 345M, the impact of such overhead on scalability is more pronounced.

C. Synchronization Optimization Analysis

a) Summation Optimization: In the four-device deployments of GPT-2 1.5B and OPT-6.7B, the former is theoretically more sensitive to synchronization due to its smaller D_m after partitioning, which lowers per-row computation latency relative to communication latency. Therefore, the following analysis focuses on GPT-2 1.5B.

In the Prefill stage, without input packing and pipeline optimization, computing the mean and variance for a single row takes about 121 cycles, while communication latency reaches approximately 220 cycles. Including the remaining LayerNorm operations, total latency increases by roughly 107% compared to our optimized design. To mitigate this, we apply pipeline optimization and balance stage delays by processing input rows in groups. Given the 256-bit data width of the communication IP, up to 16 input rows can be packed per transfer, effectively overlapping computation and communication.

SOTA designs [48], [49] typically replicate the full input across all devices to perform redundant LayerNorm computation. In contrast, our distributed scheme achieves a $3.9\times$ speedup in total LayerNorm latency while reducing buffer and storage usage by approximately 75%. In terms of KV Cache, the saved space is sufficient to store six additional output tokens.

In the Decode stage, communication latency cannot be hidden through pipelining. Nevertheless, for the LayerNorm operation alone, LORA achieves a theoretical $1.3\times$ reduction in total latency compared to SOTA designs.

b) Matrix Multiplication Optimization: For matrix multiplication, we apply two key optimizations. We modify the traversal order to prioritize the completion of all computations associated with the local activation matrix, followed by an earlier broadcast of the local activation data. For example, consider running the Prefill stage of OPT-6.7B on four U280 devices using LORA-Q with an input token length of 32. Under the conventional traversal pattern, each device computes the initial $32 \times 1024 \times 32$ MACs before waiting for activation data from others. To simplify control, we assume no global activation-related computation begins until all 32×1024 elements are synchronized. While computing these MACs takes roughly 400 cycles, receiving $32 \times 1024 \times 8$ bits from neighbors takes roughly 1090 cycles, leaving the MPU idle for roughly 690 cycles.

With the first optimization, each device performs $32 \times 1024 \times 1024$ MACs before requiring global activations, extending the computation window that overlaps with communication to about 9208 cycles. In practice, limited BRAM capacity prevents full buffering of all global activations. Therefore, our second optimization—early activation broadcasting—further expands the overlap window. Experimental results show that without these synchronization optimizations, LORA-Q's total Prefill latency for OPT-6.7B with 32 input tokens would increase by approximately 45%.

VII. Conclusion

This work presents LORA, a low-latency LLM accelerator for data centers built on a multi-FPGA platform. We propose fine-grained synchronization schemes to reduce communication latency and adopt a systolic array to construct a recurrent, high-intensity compute structure that accelerates both inter-matrix and vector-matrix multiplication by reusing the MPU. Implemented on Xilinx U280, our FPGA prototypes—LORA-F and LORA-Q—achieve average speedups of $14.4\times$ and $32.6\times$ over the NVIDIA V100 GPU, respectively. Compared to existing SOTA designs, LORA-F outperforms the best FP16-based accelerator by $2.6\times$, while LORA-Q surpasses the best W8A8-based design by $4.3\times$.

Acknowledgment

This work was supported in part by the National Key R&D Program of China under Grants 2022YFB4501600 and 2022YFB4501603, in part by the National Natural Science Foundation of China under Grants 62172380, in part by Jiangsu Provincial Natural Science Foundation under Grant BK20241818, in part by Youth Innovation Promotion Association CAS under Grant Y2021121, in part by USTC Research Funds of the Double First-Class Initiative under Grant YD2150002011.

References

[1] Grattafiori A, Dubey A, Jauhri A, et al. The llama 3 herd of models[J]. arXiv preprint arXiv:2407.21783, 2024.
[2] Achiam J, Adler S, Agarwal S, et al. Gpt-4 technical report[J]. arXiv preprint arXiv:2303.08774, 2023.
[3] Anil R, Dai A M, Firat O, et al. Palm 2 technical report[J]. arXiv preprint arXiv:2305.10403, 2023.
[4] Le Scao T, Fan A, Akiki C, et al. Bloom: A 176b-parameter open-access multilingual language model[J]. 2023.
[5] Wei J, Tay Y, Bommasani R, et al. Emergent abilities of large language models[J]. arXiv preprint arXiv:2206.07682, 2022.
[6] Zhang C, Zhang C, Li C, et al. One small step for generative ai, one giant leap for agi: A complete survey on chatgpt in aigc era[J]. arXiv preprint arXiv:2304.06488, 2023.
[7] Wang Q, Li B, Xiao T, et al. Learning deep transformer models for machine translation[J]. arXiv preprint arXiv:1906.01787, 2019.
[8] Pascual D, Egressy B, Meister C, et al. A plug-and-play method for controlled text generation[J]. arXiv preprint arXiv:2109.09707, 2021.
[9] Koo J, Park D, Jung S, et al. OPAL: Outlier-Preserved Microscaling Quantization Accelerator for Generative Large Language Models[C]//Proceedings of the 61st ACM/IEEE Design Automation Conference. 2024: 1-6.
[10] Lee C, Jin J, Kim T, et al. Owq: Outlier-aware weight quantization for efficient fine-tuning and inference of large language models[C]//Proceedings of the AAAI Conference on Artificial Intelligence. 2024, 38(12): 13355-13364.
[11] Frantar E, Ashkboos S, Hoefler T, et al. Gptq: Accurate post-training quantization for generative pre-trained transformers[J]. arXiv preprint arXiv:2210.17323, 2022.

[12] Zhao J, Zeng P, Shen G, et al. Hardware-software co-design enabling static and dynamic sparse attention mechanisms[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2024.
[13] Lu L, Jin Y, Bi H, et al. Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture[C]//MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture. 2021: 977-991.
[14] Dass J, Wu S, Shi H, et al. Vitality: Unifying low-rank and sparse approximation for vision transformer acceleration with a linear taylor attention[C]//2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA). IEEE, 2023: 415-428.
[15] Park G, Park B, Kim M, et al. Lut-gemm: Quantized matrix multiplication based on luts for efficient inference in large-scale generative language models[J]. arXiv preprint arXiv:2206.09557, 2022.
[16] Q.Jiang A, Sablayrolles A, Mensch A, et al. Mistral 7B[J]. arXiv preprint arXiv:2310.06825, 2023.
[17] Touvron H, Martin L, Stone K, et al. Llama 2: Open foundation and fine-tuned chat models[J]. arXiv preprint arXiv:2307.09288, 2023.
[18] Almazrouei E, Alobeidli H, Alshamsi A, et al. The falcon series of open language models[J]. arXiv preprint arXiv:2311.16867, 2023.
[19] Zhang S, Roller S, Goyal N, et al. Opt: Open pre-trained transformer language models[J]. arXiv preprint arXiv:2205.01068, 2022.
[20] Lee W, Lee J, Seo J, et al. InfiniGen: Efficient generative inference of large language models with dynamic KV cache management[C]//18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24). 2024: 155-172.
[21] “NVIDIA DGX Platform.” [Online]. Available: https://www.nvidia.com/en-us/data-center/dgx-platform/
[22] Xia, Haojun, et al. ”Flash-LLM: Enabling Cost-Effective and Highly-Efficient Large Generative Model Inference with Unstructured Sparsity.” Proceedings of the VLDB Endowment 17.2 (2023): 211-224.
[23] Zeng S, Liu J, Dai G, et al. FlightLLM: Efficient Large Language Model Inference with a Complete Mapping Flow on FPGA[J]. arXiv preprint arXiv:2401.03868, 2024.
[24] Zhang H, Ning A, Prabhakar R B, et al. Llmcompass: Enabling efficient hardware design for large language model inference[C]//2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA). IEEE, 2024: 1080-1096.
[25] Wang C, Gong L, Yu Q, et al. DLAU: A scalable deep learning accelerator unit on FPGA[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2016, 36(3): 513-517.
[26] Wang T, Gong L, Wang C, et al. Via: A novel vision-transformer accelerator based on fpga[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2022, 41(11): 4088-4099.
[27] Gao Y, Gong L, Wang C, et al. Algorithm/hardware co-optimization for sparsity-aware SpMM acceleration of GNNs[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2023.
[28] Lou W, Gong L, Wang C, et al. Unleashing Network/Accelerator Co-Exploration Potential on FPGAs: A Deeper Joint Search[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2024.
[29] Wang C, Gong L, Ma X, et al. WooKong: A ubiquitous accelerator for recommendation algorithms with custom instruction sets on FPGA[J]. IEEE Transactions on Computers, 2020, 69(7): 1071-1082.
[30] Zhang X, Ye H, Wang J, et al. DNNExplorer: a framework for modeling and exploring a novel paradigm of FPGA-based DNN accelerator[C]//Proceedings of the 39th International Conference on Computer-Aided Design. 2020: 1-9.
[31] Khan H, Khan A, Khan Z, et al. NPE: An FPGA-based Overlay Processor for Natural Language Processing[C]//The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. 2021: 227-227.
[32] Qu Z, Liu L, Tu F, et al. Dota: detect and omit weak attentions for scalable transformer acceleration[C]//Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2022: 14-26.
[33] Qin Y, Wang Y, Deng D, et al. FACT: FFN-Attention Co-optimized Transformer Architecture with EagerCorrelation Prediction[C]//Proceedings of the 50th Annual International Symposium on Computer Architecture.2023: 1-14.
[34] Liu, Zejian, Gang Li, and Jian Cheng. ”Hardware acceleration of fully quantized bert for efficient natural language processing.” 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2021.
[35] Bambhaniya A, Raj R, Jeong G, et al. Demystifying platform requirements for diverse llm inference use cases[J]. arXiv preprint arXiv:2406.01698, 2024.
[36] Radford A, Wu J, Child R, et al. Language models are unsupervised multitask learners[J]. OpenAI blog, 2019, 1(8): 9.
[37] Brown T, Mann B, Ryder N, et al. Language models are few-shot learners[J]. Advances in neural information processing systems, 2020, 33: 1877-1901.
[38] Fedus W, Zoph B, Shazeer N. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity[J]. Journal of Machine Learning Research, 2022, 23(120): 1-39.
[39] Chowdhery A, Narang S, Devlin J, et al. Palm: Scaling language modeling with pathways[J]. Journal of Machine Learning Research, 2023, 24(240): 1-113.
[40] Ainslie J, Lee-Thorp J, De Jong M, et al. Gqa: Training generalized multi-query transformer models from multi-head checkpoints[J]. arXiv preprint arXiv:2305.13245, 2023.
[41] Xiao G, Lin J, Seznec M, et al. Smoothquant: Accurate and efficient post-training quantization for large language models[C]//International Conference on Machine Learning. PMLR, 2023: 38087-38099.
[42] Chen J, Li S, Guo R, et al. Autoddl: Automatic distributed deep learning with near-optimal bandwidth cost[J]. IEEE Transactions on Parallel and Distributed Systems, 2024.
[43] Jia, Zhihao, Matei Zaharia, and Alex Aiken. ”Beyond data and model parallelism for deep neural networks.” Proceedings of Machine Learning and Systems 1 (2019): 1-13.
[44] Yuan J, Li X, Cheng C, et al. Oneflow: Redesign the distributed deep learning framework from scratch[J]. arXiv preprint arXiv:2110.15032, 2021.
[45] Narayanan D, Shoeybi M, Casper J, et al. Efficient large-scale language model training on gpu clusters using megatron-lm[C]//Proceedings of the international conference for high performance computing, networking, storage and analysis. 2021: 1-15.
[46] ”AMD Alveo V80 Compute Accelerator” [Online]. Available: https://www.amd.com/en/products/accelerators/alveo/v80.html
[47] Prakriya N, Chi Y, Basalama S, et al. TAPA-CS: Enabling Scalable Accelerator Design on Distributed HBM-FPGAs[C]//Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3. 2024: 966-980.
[48] Hong S, Moon S, Kim J, et al. DFX: A low-latency multi-FPGA appliance for accelerating transformer-based text generation[C]//2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2022: 616-630.
[49] Chen H, Zhang J, Du Y, et al. Understanding the potential of fpga-based spatial acceleration for large language model inference[J]. ACM Transactions on Reconfigurable Technology and Systems, 2024, 18(1): 1-29.
[50] ”Intel QuickPath Interconnect” [Online]. Available: https://www.intel.com/content/www/us/en/io/quickpath-technology/quickpath-technology-general.html
[51] ”Aurora 64B/66B LogiCORE IP Product Guide” [Online]. Available: https://docs.amd.com/r/en-US/pg074-aurora-64b66b
[52] ”Alveo U280 Data Center Accelerator Card” [Online]. Available: https://www.xilinx.com/products/boards-and-kits/alveo/u280.html
[53] ”HBM IP” [Online]. Available: https://www.amd.com/en/products/adaptive-socs-and-fpgas/intellectual-property/hbm.html
[54] Gao Y, Wang T, Gong L, et al. Hardware Accelerated Vision Transformer via Heterogeneous Architecture Design and Adaptive Dataflow Mapping[J]. IEEE Transactions on Computers, 2024.
[55] ”HLS Math Library” [Online]. Available: https://docs.amd.com/r/en-US/ug1399-vitis-hls/HLS-Math-Library
[56] Xilinx. 2017. Deep Learning with INT8 Optimization on Xilinx Devices. https://www.xilinx.com/support/documentation/white_papers/wp486deep-learning-int8.pdf
[57] Xilinx. 2020. Convolutional Neural Network with INT4 Optimization on Xilinx Devices. https://www.xilinx.com/support/documentation/white_papers/wp5214bit-optimization.pdf
[58] GRAHAM R L, KNUTH D E, PATASHNIK O. Concrete Mathematics: A Foundation for Computer Science [M]. 2nd ed. Reading, MA: Addison-Wesley, 1994.
[59] Paperno D, Kruszewski G, Lazaridou A, et al. The LAMBADA dataset: Word prediction requiring a broad discourse context[J]. arXiv preprint arXiv:1606.06031, 2016.
[60] Merity S, Xiong C, Bradbury J, et al. Pointer sentinel mixture models[J]. arXiv preprint arXiv:1609.07843, 2016.
[61] ”ESC8000A-E11” [Online]. Available: https://servers.asus.com/products/Servers/GPU-Servers/ESC8000A-E11

[62] ”AMD EPYC 7543” [Online]. Available: https://www.amd.com/en/products/cpu/amd-epyc-7543
[63] ”Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393)” [Online]. Available: https://docs.amd.com/r/2022.2-English/ug1393-vitis-application-acceleration/Getting-Started-with-Vitis
[64] ”Vivado Design Suite User Guide: Release Notes, Installation, and Licensing (UG973)” [Online]. Available: https://docs.amd.com/r/2022.2-English/ug973-vivado-release-notes-install-license/Release-Notes
[65] ”IEEE Standard for Floating-Point Arithmetic” [Online]. Available: https://standards.ieee.org/ieee/754/6210/
[66] ”NVIDIA V100 GPU” [Online]. Available: https://www.nvidia. com/en-us/data-center/v100/
[67] Shoeybi M, Patwary M, Puri R, et al. Megatron-lm: Training multi-billion parameter language models using model parallelism[J]. arXiv preprint arXiv:1909.08053, 2019.
[68] Bajgar O, Kadlec R, Kleindienst J. Embracing data abundance: Booktest dataset for reading comprehension[J]. arXiv preprint arXiv:1610.00956, 2016.
[69] ”AMD Versal^{TM} HBM Series VHK158 Evaluation Kit” [Online]. Available: https://www.amd.com/en/products/adaptive-socs-and-fpgas/evaluation-boards/vhk158.html/

Zhendong Zheng received the B.S. degree from the Computer Science and Technology, University of Science and Technology of China, Hefei, China, in 2022, where he is currently pursuing the Ph.D. degree with the University of Science and Technology of China. His current research focuses on LLM accelerators and automation programming framework design.

Qianyu Cheng is currently a Ph.D. candidate in School of Computer Science and Technology, University of Science and Technology of China. Before that, Cheng received his bachelor’s degree from the Yingcai Honors College, University of Electronic Science and Technology of China in 2022. His research interests include CPU-FPGA heterogeneous systems, distributed accelerator systems, and their application on large-scale analytical processing and multi-tenant cloud service.

Teng Wang is a research scientist with Suzhou Institute for Advanced Research, University of Science and Technology of China. His research interests focus on algorithm-level and architecture-level optimizations of FPGA for deep learning applications.

Chao Wang is currently a professor at the University of Science and Technology of China, Hefei, China. His research interests focus on multicore and reconfigurable computing. He serves as the associate editor of the ACM Transactions on Design Automations for Electronics Systems (ACM TODAES), IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB), and Microprocessors & Microsystems. He is a senior member of the ACM, IEEE and CCF.

Xuehai Zhou received the B.S., M.S., and Ph.D. degrees from the University of Science and Technology of China, Hefei, China, in 1987, 1990, and 1997, respectively. He is a Professor with the School of Computer Science, University of Science and Technology of China. His current research interest includes various aspects of multicore and distributing systems.