electronics

MDPI

Article

EFA-Trans: An Efficient and Flexible Acceleration Architecture for Transformers

Xin Yang and Tao Su *

School of Electronics and Information Technology, Sun Yat-sen University, Guangzhou 510006, China
* Correspondence: sutao@mail.sysu.edu.cn

Abstract: The topic of transformers is rapidly emerging as one of the most important key primitives in neural networks. Unfortunately, most hardware designs for transformers are deficient, either hardly considering the configurability of the design or failing to realize the complete inference process of transformers. Specifically, few studies have paid attention to the compatibility of different computing paradigms. Thus, this paper presents EFA-Trans, a highly efficient and flexible hardware accelerator architecture for transformers. To reach high performance, we propose a configurable matrix computing array and leverage on-chip memories optimizations. In addition, with the design of nonlinear modules and fine-grained scheduling, our architecture can perform complete transformer inference. EFA-Trans is also compatible with dense and sparse patterns, which further expands its application scenarios. Moreover, a performance analytic model is abstracted to guide the determination of architecture parameter sets. Finally, our designs are developed by RTL and evaluated on Xilinx ZCU102. Experimental results demonstrate that EFA-Trans provides 23.74× and 7.58× improvement in energy efficiency compared with CPU and GPU, respectively. It also shows DSP efficiency is between 3.59× and 21.07× higher than others, outperforming existing advanced works.

Keywords: transformers; neural network; bank-balanced pruning; configurable and efficient hardware accelerator; FPGA

Citation: Yang, X.; Su, T. EFA-Trans: An Efficient and Flexible Acceleration Architecture for Transformers. *Electronics* **2022**, *11*, 3550. https://doi.org/10.3390/electronics11213550

Academic Editor: Luis Gomes

Received: 9 October 2022
Accepted: 28 October 2022
Published: 31 October 2022

1. Introduction

In recent years, transformers [1] have gained much popularity and have dramatically improved the accuracy of important machine learning tasks. The pretrained models such as BERT [2] have become the preferred architectures in natural language processing (NLP), and many excellent visual transformer models [3,4] have also achieved state-of-the-art performances.

With the in-depth research of neural network models, it is necessary to focus on efficient hardware deployment and acceleration due to its long execution time. However, existing convolutional neural network–oriented accelerators are designed for compute-intensive convolution operations, making them defeated or suboptimal for processing the particular structures of transformers. To the best of our knowledge, there are relatively insufficient studies on transformer acceleration. The efficient acceleration framework Ftrans [5] can quickly deploy the transformer model on FPGA, but its performance is unsatisfactory and unable to meet the real-time requirements. Some existing transformer accelerators sequentially hardwareize the network structure, in which they have dedicated computing modules such as multiple unique PE banks to perform each multiply subroutine. This implementation lacks configurability and cannot be adapted to other variants of the workload. The A^3 [6] and ELSA [7] in the open literature are hardware architectures for accelerating the attention mechanism in various neural networks, which are not specifically designed for the transformer. The authors of [8,9] simply implement a part of workloads such as matrix multiplication or two building blocks in transformers while neglecting nonlinear layers optimizations [8] or scheduling [9], resulting in the inability to complete

the inference of the whole transformer layer. Furthermore, in order to deploy neural network models in resource-constrained devices, the weight-pruning technique as a model compression method has been commonly used in different DNN real-time applications [5,10–13]. Although all the weight matrices are pruned in transformers, there are still some workloads such as $Q \times K^T$ and $V \times S$ in the computation process that belong to dense matrix operations. However, research in this field so far almost simply considers a single dense/sparse computing pattern [5,12–14], resulting in limited applicable scenarios.

To address these issues, this paper presents a high-performance and flexible hardware architecture for transformers. The main contributions can be summarized as follows:

- We propose a high-efficiency architecture EFA-Trans for accelerating transformers that possesses excellent configurability and can achieve high utilization in matrix multiplication of different sizes.
- We implement and optimize two complicated nonlinear functions to obtain better performance. Furthermore, we proposed a customized on-chip memories design, which can realize the on-the-fly transformation of matrix transpose.
- EFA-Trans is compatible with dense and sparse computing paradigms, which can dynamically switch during runtime, significantly reducing storage requirements and overall latency. Experiments have demonstrated the effectiveness and computationally inexpensiveness of our novel scheme.
- We devise a performance analytic model to evaluate the dependencies between latency and sparsity ratio and architecture parameter sets. After a comprehensive analysis of the model, we reach a balance between acceleration performance and resource consumption of EFA-Trans.

The remainder of this paper is organized as follows. Section 2 presents the background of the transformer network architecture and model pruning. Section 3 details the acceleration architecture of EFA-Trans. Section 4 presents a performance analytic model to guide the determination of architecture parameters. Section 5 gives the characteristics, resources, and performance comparison results of our proposed architecture. Finally, Section 6 draws the conclusions of this paper.

2. Background and Motivation

2.1. Attention-Based Transformer Models

The recent attention-based transformer models [1–4] have achieved a significant improvement on a wide range of NLP and Computer Vision (CV) tasks. Although the connectivity among layers and the organization forms of these networks may be different, they are all stacked by basic blocks. It can be noted that most of the trainable parameters and calculations are in these stacks. In addition, in the architecture of transformers, the decoder is similar to the encoder, and encoder is applied more universally. Our research mainly focuses on the inference acceleration of the transformer model. Figure 1 shows the basic structure of encoder in transformer, which is mainly composed of multihead attention (MHA) layers and feed-forward network (FFN) layers. All these layers are followed by the residual connection and layer normalization (LN). For an input sequence with L tokens, *embeddings* are linearly transformed to three tensors: Q (queries), K (keys), and V (values) by multiplying the embedding matrix with different weight matrices ($L \times d_q$, d_k, d_v). After feeding the Q, K, V tensors into the MHA layer, the attention probabilities are produced by employing the softmax function on $Q \times K^T$, and the dependencies between input tokens can be obtained by multiplying attention probabilities with V. Followed by MHA, a residual layer adds the attention outputs with embeddings and executes layer normalization. After the operations of MHA are completed, the output of residual addition and LN layers is fed into the FFN layer to generate the FFN output. Finally, another residual operation is conducted and outputs *block_out*, which can be used as the input to the next block.

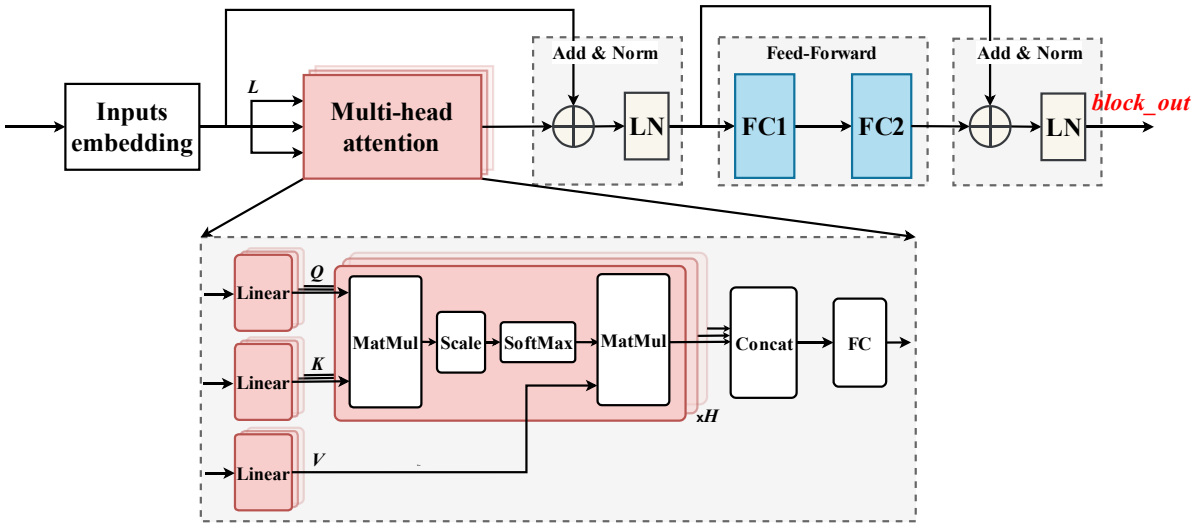

Figure 1. Model architecture of the transformer encoder.

The detailed key computations of transformer encoder model are illustrated in Table 1. X represents the input to an encoder, X_1–X_5 represent subsequent outputs after each operation. MHA is the structure to measure the relationship among tokens, which can be broken down into matrix multiplication, softmax function, and vector multiplied by a scalar (COM1~4). In the position-wise feed-forward network of each layer, two linear transformations are executed, which mainly contain vector matrix multiplication, and the first one involves Rectified Linear Unit (ReLU). Residual Addition and LN layers are inserted after each MHA and FFN. It mainly includes the addition of matrix elements and the nonlinear functions. Specifying these computing loads can help us to efficiently map the network building blocks to the hardware architecture.

Table 1. Computation for the transformer's encoder with H attention heads.

Embedding and Positional Encoding	
EM/PE	$X = Embedding(inputs) + PE(inputs)$
Multihead Self-Attention	
COM1	$[Q, K, V] = X\,[W_Q, W_K, W_V]$
COM2	$S = Softmax\left(QK^T/\sqrt{d_k}\right)$
COM3	$Z^{0-H} = V \cdot S$
COM4	$X_1 = Concat(Z^{0-H}) \cdot W_O$
Residual Addition and Layer Normalization A	
LN-A	$X_2 = LayerNorm(X + X_1)$
Position-Wise Feed-Forward	
FFN1	$X_3 = ReLU(X_2 W^{F1} + b^{F1})$
FFN2	$X_4 = X_3 W^{F2} + b^{F2}$
Residual Addition and Layer Normalization B	
LN-B	$X_5 = LayerNorm(X_2 + X_4)$

2.2. Model Pruning

Pruning technologies were introduced in previous neural network compression [5–9], consisting of irregular pruning and regular pruning. The irregular pruning method sets a threshold value for a specific pruning ratio and prunes the elements below the threshold value. Irregular pruning can achieve a high pruning rate without causing great performance degradation. However, the highly irregular weight matrices cannot effectively utilize

hardware resources, which is not suitable for hardware acceleration. In contrast, the layout of nonzero elements in the regular pruning scheme has a high regularity, which is hardware-friendly and easy to deploy on hardware. Common regular pruning schemes include bank-balanced pruning [5–7], block circular matrix pruning [8] and block pruning [9]. In this paper, we adopt the bank-balanced pruning scheme.

3. Accelerator Architecture

3.1. Architecture Overview

The top-level architecture of the EFA-Trans, a customized system for high-performance and flexible transformer inference, is illustrated in Figure 2. It can be divided into two parts: the software program on the host and the hardware accelerator. Section 2 has introduced the entire model layers. Different from the encoder layer, initial steps such as embedding layers usually require a large amount of memory bandwidth and negligible computation, which are conducted outside the accelerators, and the results are then sent to the accelerator. The input vectors and weights are stored in off-chip memory DRAM in advance and moved to the accelerator through the AXI4 interface.

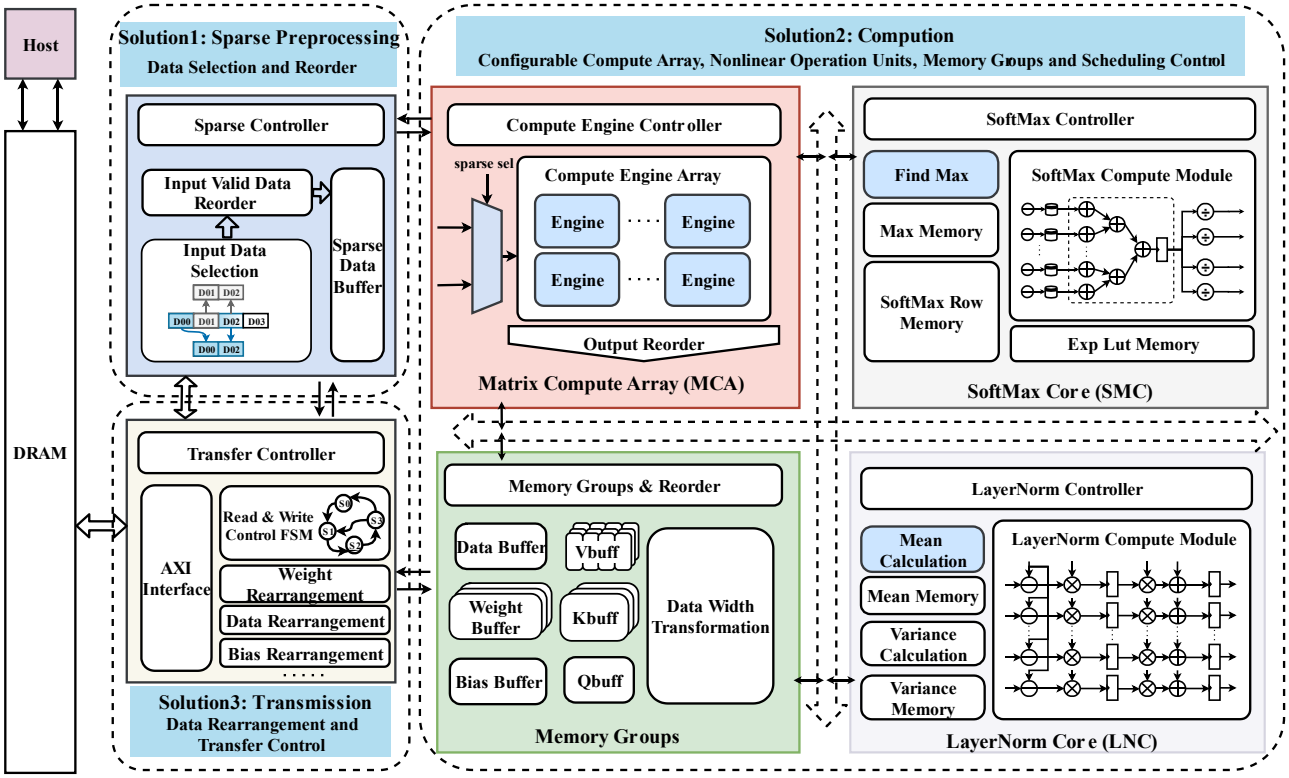

Figure 2. Top-level architecture diagram of the proposed hardware accelerator EFA-Trans.

On the acceleration architecture side, the accelerator consists of a data transmission solution block, an optional sparse preprocessing solution block, on-chip memories, and computation solution blocks. The transmission block can access external memory through the AXI4 bus, which is responsible for data movement. The optional sparse processing block can organize the inputs beforehand to facilitate the subsequent direct calculations. The on-chip intermediate buffers are used for storing all the intermediate data such as weight, matrices, parameters, and so on. Computation blocks are composed of the matrix computation array (MCA) and the optimal hardware implementation of nonlinear functions, where each engine in the MCA has two data paths to support matrix computation for both dense and sparse modes. Thus, all workloads of transformer layers can be realized through mapping on these configurable and flexible computation blocks in EFA-Trans.

3.2. Matrix Computation Array (MCA) and Workload Mapping

The workload of transformer contains matrix multiplication $(M \times M)$ and matrix vector multiplication $(M \times V)$. For the sake of simplicity and unity, we decompose matrix multiplication into matrix vector multiplication using inner product operation. The MCA, as the main arithmetic module in EFA-Trans, is designed to achieve high performance for all kinds of matrix multiplications and matrix vector multiplication, which could achieve near-peak utilization.

Figure 3a shows the circuits of the MCA, which contains n engines, each of which implements independents m multiplication lanes. Parallelism parameters m and n belong to architecture parameters and can be configured. Each engine is composed of m-multipliers, m-adder tree, and an accumulator with dynamically adjustable modes. After accumulation is complete, a postprocessing unit (PPU) optionally performs operations such as ReLU and scaling. Final results are stored in output memory. In addition, there are residual addition layers in the transformer network structure; in order to realize element-wise addition, the original data are prepared in the on-chip buffers in advance. By configuring the mode of the accumulator, we can directly perform the element-wise addition operation in the MCA, which not only reduces the additional hardware resources but also ensures the efficiency of the operations. Since the calculation process takes more than one cycle to complete, the circuits of the MCA are designed to be pipelined to improve the utilization of the MCA and throughput.

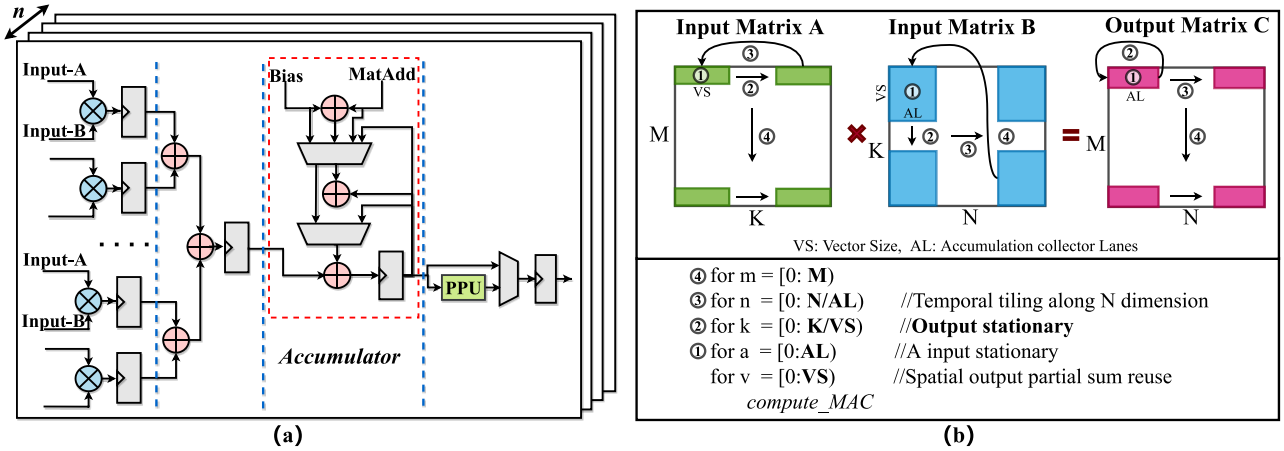

Figure 3. (**a**) The block diagram of an engine in the MCA. (**b**) Workload mapping and data reuse.

The accelerator minimizes on-chip buffer reads by employing an output-stationary dataflow. For example, the workload of $A \times B = C$ *(matrix)* can be split into subtasks through tiling and data reuse, as illustrated in Figure 3b. After meticulous scheduling and control, the entire dataflow can be fully mapped to the MCA for execution. A-matrix inputs are read out of the A-buffer and reused across the n engines, and B-matrix inputs are read out of multi-B-buffer once each cycle. Partial sums are temporally accumulated in a register before sending the completed sum to the PPU. By configuring the buffer address generators and repeating the process of calculating the output matrix, the MCA can compute the MHA layer, element-wise addition, and fully connected layers.

3.3. Nonlinear Modules Design and Optimization Strategies

3.3.1. Softmax Core (SMC) and Optimization

In the attention mechanism, relevances between pairs of inputs are measured by $Q \times K^T$ and then normalized by the softmax function. Normally, to perform an exponent computation, a complicated hardware exponent calculation unit should be used. However, exponential hardware implementation is quite expensive in terms of resource consumption, so we use lookup tables instead. Thus, one challenge with this approach arises: handling

data overflow. The outcome of an exponent function is sensitive to the increase in inputs, and thus it can easily cause an overflow in a fixed-point representation. Fortunately, softmax can keep the invariant to subtraction. Therefore, we subtract the maximum from all the elements [6], and then the exponential results will always be limited between 0 and 1, which can be rewritten as Formula (1).

$$softmax(x_i) = \frac{exp(x_i - x_{max})}{\sum_{j=1}^{d_k} exp(x_j - x_{max})} \quad (1)$$

We propose a hardware implementation SMC for the softmax function to support and accelerate corresponding operations, as is highlighted in Figure 4. The first stage of the pipeline computes the exponential outcome after subtracting the maximum value from the input value. This hardware consists of M subtractors and exponential function lookup tables *(EXP_LUT)* followed by an m-way adder tree for a sum reduction operation. The second stage is an accumulator, which can allow multiple cycles to complete the numerical addition of a single row through fine control. The results of this operation are stored in the corresponding register. At the same time, single-row data are delayed for multiple cycles and stored in the *ping-pong ROW BUFF*, as shown in the red dotted box. Then, once the sum of the exponential values of a single row is finished, these set of values stored in *ROW BUFF* are passed to the m-way divider arrays for the final results of the softmax function.

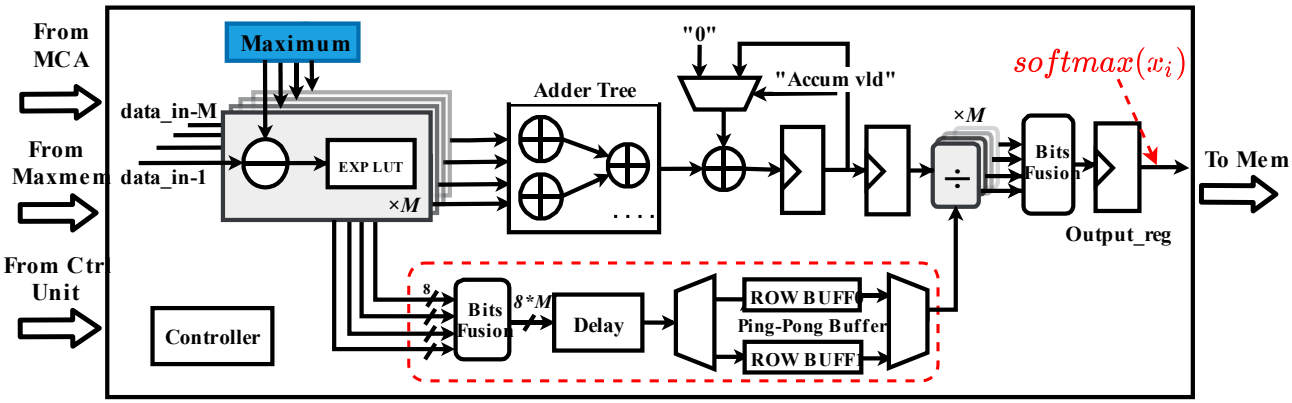

Figure 4. The detailed architecture of SMC.

When we carefully analyze Formula (1), we have to calculate the maximum value in each row before calculating the softmax outputs, where the maximum value acquisition needs to occupy partial latency. Thereby benefiting from the output stationary of the MCA, we directly couple the outputs of the MCA with the maximum calculation unit to find the maximum in advance so as to optimize the delay. Here, the maximum calculation unit is composed of the comparator tree, and the intermediate value is temporarily stored in the registers.

3.3.2. LayerNorm Core (LNC) and Optimization

For purpose of ensuring the stability of data feature distribution during model training, LN is added to the transformer model to accelerate the convergence rate of the model. Different from Batch Normalization (BN)'s idea of normalizing each batch of data, LN's main idea is to calculate the mean and variance on different channels of each sample and then normalize.

As discussed in Section 2, the LN function needs to be calculated before the FFN and next block starts. This means the LN function is always on the critical path of the system latency when we analyze the computation process. Thus, we designed an independent LNC

to address the computation inefficiency in LN due to the fact that the MCA is unsuitable for LN. The LN function used in transformer is:

$$Output(i,j) = \frac{G(i,j) - E(G,i)}{\sqrt{var(G,i) + \varepsilon}} \gamma_j + \beta_j \tag{2}$$

$$E(G,i) = \frac{1}{d_{model}} \sum_{k=1}^{d_{model}} G(i,k) \tag{3}$$

$$var(G,i) = E(G,i)^2 - \frac{1}{d_{model}} \sum_{k=1}^{d_{model}} G(i,k)^2 \tag{4}$$

where *E (G,i)* and *var (G,i)* are the mean value and variance of all the elements in the *i*-th row of matrix *G*. The constant ε is used to avoid the denominator from being zero. According to the intuitive method, we need to calculate the mean and variance successively and then use them to calculate the outputs of LN. If the matrix size is $M * N$ and the parallelism of the mean and variance operation module is n, we roughly estimate that at least $2 * M * (N/n)$ cycles are added to the whole system latency.

For the sake of minimizing the delay of this module, the key is to allow the LN module to start running ahead of schedule. We adopt two methods. First, we choose another way to calculate the variance [9] so that the mean and variance can be operated essentially simultaneously, as shown in Expression (4). Second, since the MCA uses an output-stationary dataflow for the outputs, we can keep the inputs of the mean and variance calculation module directly connected to the outputs of the MCA. At last, the matrix and variance can be acquired concurrently during the matrix operation, and very few cycles are required between finishing the calculation of all the elements of matrix *G* and starting LN computation, significantly reducing the on-chip storage access and runtime latency of the entire system. As is shown in Figure 5, In the optimized way, the mean and variance calculation time can be roughly hidden in the matrix calculation time. The delay time is further reduced compared to the straightforward way.

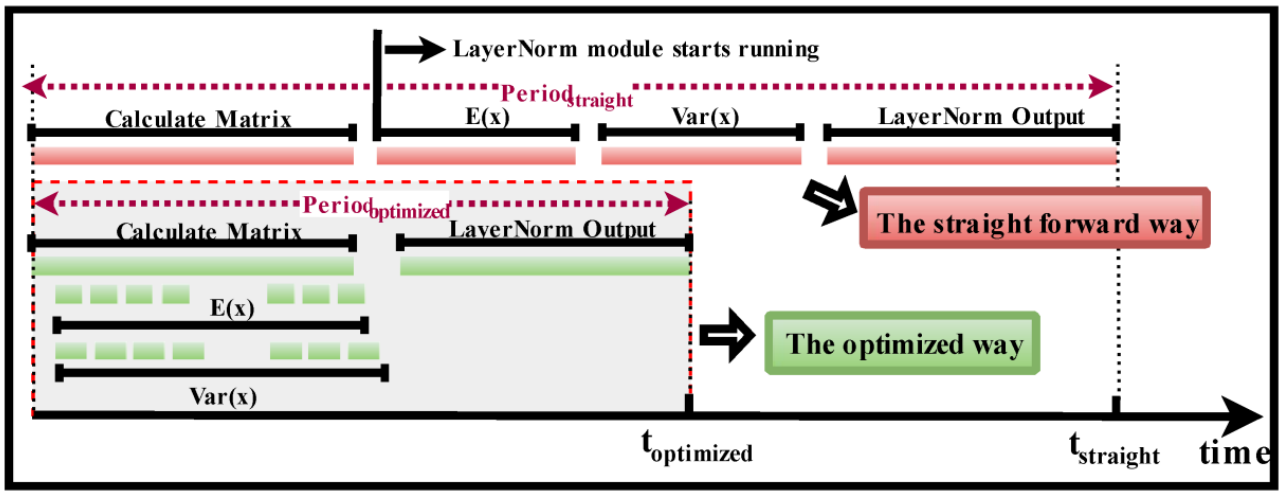

Figure 5. Timing diagram of LNC optimization.

The architecture of the LNC is described in Figure 6. Figure 6a details the circuit of mean and variance calculation. Among them, the red dotted box shows the mean value calculation circuit. Every valid cycle, this module takes *N* values and then sends them to the *n*-way adder tree for a parallel sum reduction. The accumulator follows and generates the valid signal identifier and the corresponding resultant data after the single-row data operation is completed. The lower part is the variance computing circuit, where each of its vector elements is self-multiplied to complete the $G(i,k)^2$ operation before the data are

fed into the adder tree. In addition, the mean value is bypassed to the variance computing module to realize the Formula (4).

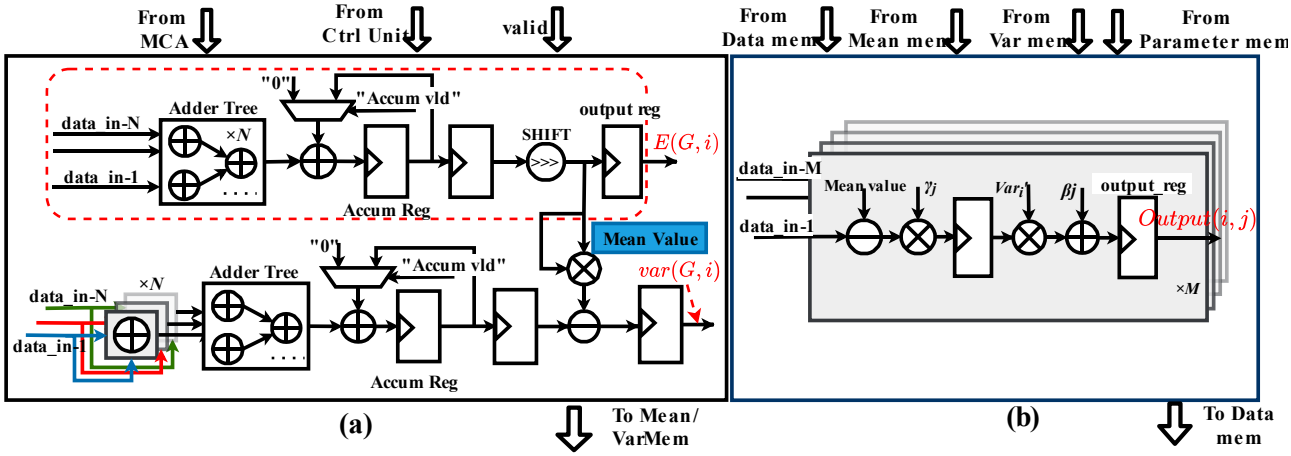

Figure 6. The detailed architecture of (**a**) mean value, variance computing modules, and (**b**) LNC.

Figure 6b makes a hardware implementation of the Expression (2) using a pipeline approach to generate the final outcome of LN. The parallelism is designed as m, which matches the overall hardware architecture parameters. Except matrices data, the mean and variance need to be obtained from *mem/var* memory. The parameters γ_j, β_j also need to be stored in a parameter buffer in advance, and the final LN outputs are stored in output data buffers.

3.4. Improvement for the on-Chip Memories

3.4.1. The Features of on-Chip Memories

In the hardware design, the design method of double buffering and collaborative optimization with scheduling could result in remarkable inference performance for the transformers. Double buffers are operated in a ping-pong manner to overlap data transfer with computation.

As is described in Section 2, the transformer model contains many operations between matrices. Thus, memory hierarchy is equipped with multiple on-chip buffers to satisfy varying workloads. In the workload of the MHA layer, the three matrices Q, K, V cannot be simply stored in data buffers because they involve mutual operations. Therefore, we separately set the *Q-buffer*, *K-buffer*, and *V-buffer* to store the corresponding matrices, which can reduce duplicate off-chip memory accesses. Furthermore, to meet the on-chip data transfer bandwidth requirements, we adopt the design method of multibank memory.

3.4.2. On-the-fly Transformation of Matrix Transpose

The workload of self-attention mechanism includes $Q \times K^T$. In the original data flow, we can obtain the K matrix after the MCA operation. Therefore, if we want to ensure that the control logic of the next matrix multiplication operation remains unchanged, we need to transpose the K matrix. However, a separate matrix transpose operation is not only time-consuming but also needs additional on-chip buffer to assist data rearrangement. In this design, we ingeniously designed the *K-buffer* and the corresponding write control circuit so that the on-the-fly transformation of matrix transpose can be achieved without taking up additional delay.

As shown in Figure 7, we use a more fine-grained way to split the *K-buffer*. Byte enable *byte_en* can be used to control mask writing of *K-buffer*. Take a 4×4 matrix transpose as an example: From *cycle-1* to *cycle-4*, the same addresses are generated, and the change patterns of *byte_en* are: *4'b0001*, *4'b0010*, *4'b0100*, and *4'b1000*, as described in Figure 7. After four cycles, the computing results of the MCA are completed, and the data inside the *K-buffer* are the transposed form of the result matrix.

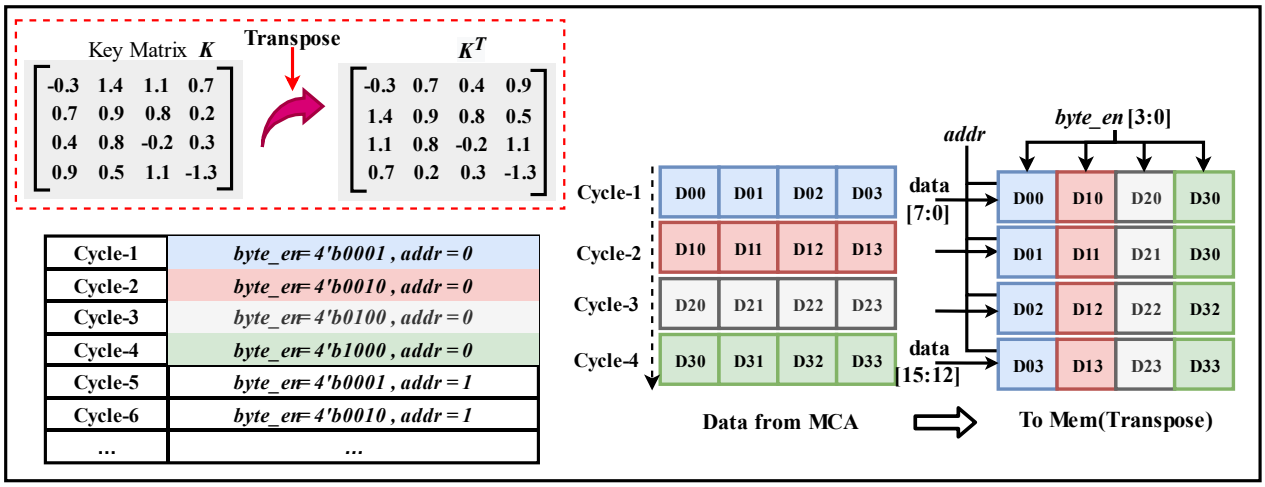

Figure 7. The schematic diagram of matrix transposition rearrangement.

3.5. Supporting Sparse Matrix Computation

For hardware-friendly weight pruning, bank-balanced pruning (BBP) [15] divides the weight matrix into subrows/columns and conducts Top-k pruning in each one. Depending on the bank partition and pruning rate, we can sort the data in each bank and find a reasonable pruning threshold, and then the data within a bank that are less than the threshold are selected and set to zero. Algorithm 1 describes this proposal in detail. Owing to fine-grained pruning, BBP can maintain more important information. Thus, compared to other hardware-friendly weight pruning techniques [13,16], BBP can achieve better accuracy under the same sparsity ratios.

Algorithm 1: Bank-balanced weight pruning algorithm

Input: weight matrix W, matrix width n, matrix height m, the number of each bank bn, the number of nonpruned per bank r
Output: pruned weight matrix W_p
Set $W_p = W$
Set column division $j = n/bn$
Divide W_p into m vectors: $W_1, W_2, \ldots, W_m$
foreach W_i in $W_1, W_2, \ldots, W_m$ do
    Divide the row R into j blocks: $R_1, R_2, \ldots, R_j$
    foreach R_j in $R_1, R_2, \ldots, R_j$ do
        sort the abs(***value***) in R_j
        Calculate the threshold T in each block
        foreach k in ***values*** do
            Setting the value as zero if (abs($value_k$) < T)
        end
    end
end
W_p = concatenate ($W_1, W_2, \ldots, W_m$)

We jointly optimized the BBP algorithm and the parameters of hardware architecture to cut down the logic complexity and ensure load balancing. Since in multiplication, the result of multiplying zero by any number is zero, we can skip the multiplication operations corresponding to zero to reduce the computational effort. Therefore, in order to ensure high utilization of the MCA, reduce pipeline stalls, and improve compatibility, we propose a sparse preprocessing unit to select and rearrange these data that are truly active for matrix

multiplication results. In this way, the accelerator can be compatible with both dense and sparse operations through different mode configurations of the MCA.

A coarse-grained, two-stage pipelined SIMD unit for sparse preprocessing was designed, as described in Figure 8. At the moment of *T1*, the inputs are *D00*, *D01*, *D02*, and *D03*, and the corresponding mask identification signals are *M01*, *M02*, *M00*, and *M02*. After the multiplexer selects, *D01*, *D02*, *D00*, and *D02* are identified as active signals, and then they are fused and sent to the on-chip buffer for storage. In general, the first stage selects the valid input data corresponding to the mask information temporarily. The second stage reorganizes and stores the data in the sparse on-chip buffers. Finally, the selected feature map and pruned weight are read from the buffers and fed to the MCA for subsequent operations.

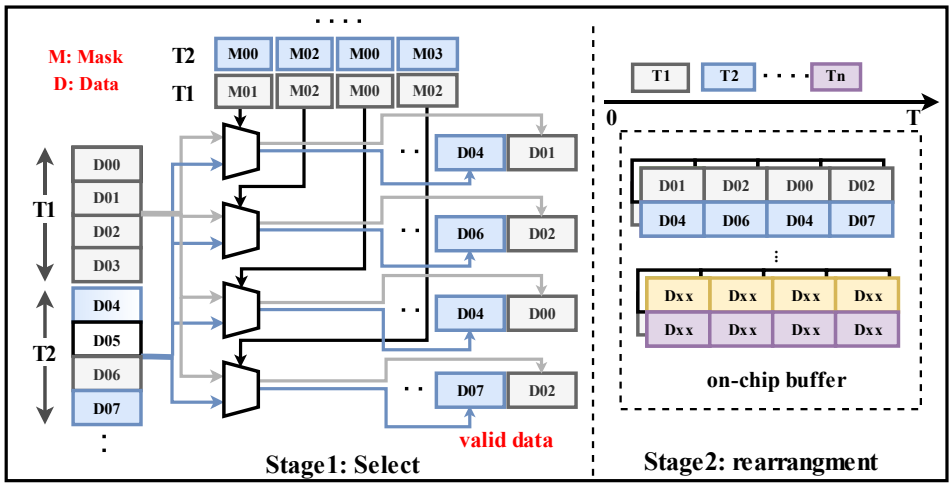

Figure 8. The schematic diagram of sparse preprocessing module.

3.6. Fine-Grained Scheduling

Encoding in transformers is a complicated dataflow. One of the key challenges for efficient hardware inference is to hide the off-chip memory access overhead. In this design, we carefully designed a control flow by using double buffering to match the data calculation and transfer cycles through fine-grained, task-level scheduling.

The control flow, which is shown in Figure 9, can complete the calculation and data handling of the transformer structure. The computation sequence is grouped into four states, where each state is associated with a set of computations (blue blocks) along with weight (pink blocks) to be prefetched in it. The second and fourth lines are data sizes for the prefetch and numbers of computation operations, and those in the fifth line are buffer names for saving. Operations in the horizontal direction have to be executed sequentially first, followed by the those in the vertical direction successively in each state. For example, in STATE2, we can calculate multihead attention layers *Q*, *K*, *V* and *softmax* operations of one head because the operations are dependent in the MHA. Then the MHA operations of the other heads are executed in turn until all the computing loads are completed and jump to the next state.

Note that the computation and transfer cycles can be estimated given workload and architecture parameters, as detailed in Section 4. By employing cycles analysis and by considering the data dependency between computation workload and prefetched data, we can balance the computation and transfer cycle well for all the states. As a result, we are able to achieve high performance through workload mapping on EFA-Trans.

STATE	STATE1	STATE2	STATE3	STATE4
Data Fetch/Load	X、WQ、WK、WV、WO	W_Ln1、Bias_Ln1、WF1、Bias_F1	WF2、Bias_F2、W_Ln2、Bias_Ln2	WQ、WK、WV、WO (for next layer)
Data Size	t*dmodel + 4*dmodel*dmodel	ffn1 * dmodel + 2*dmodel + dmodel	dmodel*ffn2 + 2*dmodel * dmodel	4*dmodel*dmodel
Computation	------	[WQ1、WK1、WV1]*X = Q、K、V ; Q*KT = Q1 ; Softmax(Q1) = Q ; V1*Q = Z1 ..... [WQn、WKn、WVn]*X = Q、K、V ; Q*KT = Q1 ; Softmax(Q1) = Q ; Vn*Q = Zn X ->IN_Reorder ; WO*Z + X = A0	Layer Norm（A0）= L0 ; Relu(WF1*L0 + BF1) = F0 ; X -> IN_Reorder	WF2*F0 + BF2 + A0 = A1 ; LayerNorm(A1) = L1
Number of Operations	------	3*dmodel*dmodel*t + dmodel*t*t + dmodel*dmodel*t + h*softmax(t*t)	LN(dmodel*t) + ffn1*dmodel*t	LN(dmodel*t) + ffn2*dmodel*t
Save	------	Q、K、V、Q1、IN_Reorder、IN0	IN1、IN2	IN_Reorder、IN1、IN0

Figure 9. The control flow of the EFA-Trans. Data fetch is colored in pink, and the computation process is colored in blue.

4. Performance Analytical Model

Performance and resource usage of accelerators will vary under different parameter sets. Therefore, establishing an analytical model is very helpful for the determination of architecture parameters and model deployment. We determined several important hardware parameters that affect accelerator performance according to our design, which form a set S. We define an acceleration architecture as $S_i = \langle p_i, m_i, n_i \rangle$, consisting of the pruning rate p_i, engine parallelism m_i, and MCA parallelism n_i. The space of all legal values is denoted by Formula (5). Based on these hardware parameters, we model the latency. The total runtime latency of the system is determined by the bottleneck of the transmission and computation.

$$\begin{cases} 0 < p < 1 \\ 0 < m \times n < (\sharp\, num\ of\ PEs) \end{cases} \tag{5}$$

Transmission Latency: All the transformer's input and part of the weights should be prefetched to the on-chip memory, and the results need to be moved from on-chip memory to off-chip memory. As Formula (6) shows, transmission overhead is related to the memory access amount and bandwidth, where the variables $\{M_{num}, DBW, TBW\}$ represent the data access, data bit width, and transmission bandwidth, respectively. The product of the data amount and the data bit width determines all memory access sizes, and when divided by the transmission bandwidth, the approximate number of cycles is obtained.

$$LAT_{trans} = \frac{M_{num} \times DBW}{TBW} \tag{6}$$

Computing Latency: As for the computing latency estimation, there are total of $m_i \times n_i$ multiply units in the MCA, and the parallelism of nonlinear modules is h_i . The latency of operations is decomposed in matrix and nonlinear functions, as shown in Formulas (7) and (8). Benefiting from the pipeline design, the matrix latency $LAT^{M}_{comp\_i}$ in each iteration is determined by MCA pipeline depth MPP_{depth}, matrix dimension $\{MA_i, MB_i, MC_i\}$, and architecture parameters $\{m_i, n_i\}$. The latency of the nonlinear layer is also related to the parallel parameter h_i and pipeline stages of nonlinear modules $NLPP_{depth}$ and preprocessing depth PRE_{depth}. At the runtime, the computing latency and the transmission latency may vary from iteration to iteration as the matrices have different sizes.

$$LAT^{M}_{comp\_i} = \frac{MB_i \times MC_i \times MA_i}{m_i \times n_i} + MPP_{depth} \tag{7}$$

$$LAT^{NL}_{comp_i} = \frac{MB_i \times MA_i}{h_i} + NLPP_{depth} + PRE_{depth} \tag{8}$$

Matching the data computed and load cycles carefully can reduce the overall latency. After the workloads are determined, we can explore the optimal hardware architecture parameters accordingly through estimation, and the computation and memory access in scheduling can be efficiently balanced. In order to achieve a "sweet spot", the dense EFA-Trans is configured as < 32, 32 >. In case of a dense–sparse operation pattern, architecture parameters are set to < 12.5%, 32, 16 >.

5. Evaluation and Experiments

5.1. Experimental Setup

To evaluate the performance of EFA-Trans, IWSLT 2016 German–English corpus [17], which is a representative benchmark data set for the transformer, was used. For the evaluation of accuracy degradation due to pruning, the bilingual evaluation understudy (BLEU) score [18] was used. We trained and tested the transformer (hidden layer size was 2048, head num was 8, and sentence length was set to 64) using the IWSLT 2016 German–English corpus. After pruning, the BLEU value changed from 30.41 to 31.05. Learning from [9], replacing FP32 with INT8 in the transformer can greatly reduce the computational complexity with limited accuracy loss. Thus, we quantized FP32 to INT8 for hardware deployment. After that, the BLEU score dropped to 30.65, proving that quantizing with INT8 in this network is acceptable. Vivado 2019.2 and the Xilinx ZCU102 board were used to evaluate the proposed hardware accelerators in the experiments.

5.2. Characteristics of Accelerators

When comparing the characteristics implemented on a reconfigurable device with other works, to our knowledge, there is no research that has been able to realize a configurable accelerator, which can complete the whole-layer network inference and have compatible dense and sparse characteristics simultaneously. Authors of [5] and [12] show how to deploy a complete single-layer network, but the corresponding hardware for each operator on the dataflow is arranged, which is not configurable, and both of which are aimed at hardware acceleration for specific sparse matrices. Designs included in [8] and [9] have generic arrays for matrix operations. However, Ref. [9] has only made design optimization for the pure computation modules of two building blocks and cannot map the entire transformer layer. Authors of [8] only implemented matrix operations without elaborating on the design of the nonlinear layers. Compared with these works, our work can simultaneously realize the configurable computing array and the whole inference. In addition, it can be selectively compatible with dense and sparse patterns, as described in Table 2.

Table 2. Comparison of Functional Implementation Circumstances.

	[5]	[8]	[9]	[12]	This Work
# Configurability	✘	✔	✔	✘	✔
# Complete layer	✔	✘	✘	✔	✔
# Compatibility [1]	✘	✘	✘	✘	✔

"Compatibility [1]" indicates the characteristic that supports both dense and sparse paradigms at run-time.

5.3. Resource Utilization

We used Vivado (2019.2) to synthesize and implement the accelerators and deploy them on a Xilinx ZCU102 field programmable gate arrays (FPGA) board, which has 912 BRAM18E, 2520 DSP, and 274.08 k logic cells (LUT). Taking a single-layer workload as an example, some more details of the network architecture containing the dimension of the operation matrix, the number of operands, etc., are listed in Table 3. The data flow in the network is mapped to the MCA, SMC, and LNC computing modules successively for implementation, and the matrix dimension column illustrates the feature map matrix dimension and the weight matrix dimension of the participating operations. Since the accelerator proposed in this paper is recursive, the MCA can be configured to accomplish

the relevant operations for different matrix dimensions. Moreover, the resource utilization of each submodule in the acceleration architecture (Figure 2) is also described in Table 3. The multiplication operations in the MCA exploit DSP hardware resources in the FPGA, which corresponds to the architecture parameters. Apart from consuming BRAM, the memory groups also occupy a number of LUT and FF hardware resources due to some of the memory cells being implemented using registers.

Table 3. The Operations and Resource Utilization of Submodules in Acceleration Architecture.

	Operations		**Resource**			
	Matrix Dimensions	**OPs**	**DSP**	**LUT**	**FF**	**BRAM**
MCA	X [1]—[64, 512] MHA—[512, 512] FFN1—[512, 2048] FFN2—[2048, 512]	0.41 G	1024	40,508	6114	/
SMC	Softmax—[64, 64]	0.13 M	0	4665	5774	/
LNC	LN—[64, 512]	0.46 M	0	2787	838	/
Memory Groups	/	/	/	16,685	17,936	539
Transmission	/	/	/	200	107	/

"X [1]" indicates the input, which corresponds to the computation in Table 1.

Table 4 shows the utilization report and latency of dense and compatible dense–sparse accelerators, which can reach frequencies of 185 MHz and 150 MHz, respectively. In compatible dense–sparse mode, LUT resources are occupied more, which accordingly affects the frequency. However, the latter can achieve better latency 0.87 ms with lower BRAM and DSP consumption.

Table 4. Resource Utilization Report and Latency of Accelerators.

	Resource				**Latency (ms)**
	DSP	**LUT**	**FF**	**BRAM**	
Available	2520	274,080	548,160	912	-
Dense	1024	65,385	31,739	539	1.47
Dense–Sparse	512	132,433	52,332	439	**0.87**

5.4. Performance Comparison

We completed comparison experiments on the CPU, GPU, and FPGA, respectively. In this experiment, the CPU was an Intel (R) Core (TM) CPU i5-4460 @ 3.20 GHz, and the GPU was Nvidia RTX-3060, both of whose results were obtained by running the same network structure using pytorch framework.

In comparison experiments with other works based on FPGA, Ftrans [5] was built on VCU118 FPGA via Vivado HLS, which achieves 2.94 ms in accelerating the shallow transformer. FQ-BERT was a fully quantized BERT proposed by [19]. Authors of [8] created an algorithm–hardware codesign for the attention mechanism, which can reach 1.87 Tops on the runtime throughput performance in ZCU102. Considering the size and structure of model, metrics such as latency are difficult to compare horizontally, so we chose energy efficiency and throughput per DSP as evaluation metrics to provide a fair comparison. Their results are not affected by the size and structure of models.

The experimental results are shown in Tables 5 and 6. Our energy efficiency is 23.74× that of CPU and 7.58× that of GPU. Desktop GPU wins in throughput due to its high computing parallelism and bandwidth, but its extreme computing power cannot be ignored. Due to the facts that we classify all operations of the model and we implement the corresponding control flow and underlying module for each type of workload, our design achieves certain customization and high performance. Our throughput is higher than that of other related works. Further, a more fine-grained mapping way and more universal

modules allows our DSP slices utilized to be lower than others. As can be seen from the Table 6, our throughput per DSP is more than 3.59× that of [8] and 21.07× the FQ-BERT in [19], much better than existing accelerators. This further illustrates the advantages of EFA-Trans in terms of resource utilization. In a word, EFA-Trans achieves a balance between performance and resource consumption.

Table 5. Comparison of Design against CPU and GPU.

	CPU i5-4460	GPU RTX-3060	This Work ZCU102
Latency (ms)	4.66	0.71	1.47
Throughput (GOPS)	88.2	579.3	279.8
Power (W)	41	86	5.48
Energy Efficiency (GOPS/W)	2.15	6.74	**51.06**

Table 6. Comparison of Design against related works.

	Ftrans [5]	**FQ-BERT [19]**	**Work [8]**	**This Work**
Device	VCU118	ZCU111	ZCU102	ZCU102
Throughput (GOPS)	101.8	22.7	190.1	279.8
DSP Slices Util.	6531	1751	2500	1024
DSP Efficiency (GOPS/DSP)	0.015	0.013	0.076	**0.274**
Relative Speedup	1.15×	1× (baseline)	5.84×	**21.07×**

6. Conclusions

This paper presents a custom hardware architecture, EFA-Trans, for accelerating the transformer neural network with high efficiency and flexibility. In order to perform the inference efficiently, computing process and workload were analyzed in detail. Furthermore, we exploited optimization strategies in the design of the MCA, nonlinear cores, and other key modules. Moreover, we proposed a sparse preprocessing unit and fine-grained control flow to achieve dramatic performance improvement. Finally, a performance pre-evaluate analytical model was utilized to guide the hardware design. The experiments showed that the deployment of EFA-Trans on FPGA significantly outperforms CPU, GPU, and other transformer accelerators.

Author Contributions: Data curation, X.Y.; Funding acquisition, T.S.; Methodology, X.Y. and T.S.; Project administration, X.Y. and T.S.; Software, X.Y.; Supervision, T.S.; Writing—original draft, X.Y.; Writing—review and editing, T.S. All authors have read and agreed to the published version of the manuscript.

Funding: The research was funded by the Science and Technology Program of Guangdong Province under Grant 2021B1101270007.

Data Availability Statement: The data presented in this study are available on request from the corresponding author. The data are not publicly available due to privacy.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention is All you Need. *Adv. Neural Inf. Process. Syst.* **2017**, *30*. [CrossRef]
2. Devlin, J.; Chang, M.; Lee, K.; Toutanova, K. BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding. *arXiv* **2019**, arXiv:1810.04805. [CrossRef]
3. Dosovitskiy, A.; Beyer, L.; Kolesnikov, A.; Weissenborn, D.; Zhai, X.; Unterthiner, T.; Dehghani, M.; Minderer, M.; Heigold, G.; Gelly, S.; et al. An Image Is Worth 16x16 Words: Transformers for Image Recognition at Scale. *arXiv* **2021**, arXiv:2010.11929. [CrossRef]

4. Carion, N.; Massa, F.; Synnaeve, G.; Usunier, N.; Kirillov, A.; Zagoruyko, S. End-to-end object detection with transformers. In Proceedings of the European Conference on Computer Vision (ECCV), Glasgow, UK, 23–28 August 2020; pp. 213–229.
5. Li, B.; Pandey, S.; Fang, H.; Lyv, Y.; Li, J.; Chen, J.; Xie, M.; Wan, L.; Liu, H.; Ding, C. FTRANS: Energy-efficient acceleration of transformers using FPGA. In Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design, Boston, MA, USA, 10–12 August 2020; pp. 175–180. [CrossRef]
6. Ham, T.J.; Jung, S.J.; Kim, S.; Oh, Y.H.; Park, Y.; Song, Y.; Park, J.-H.; Lee, S.; Park, K.; Lee, J.W.; et al. Aˆ3: Accelerating Attention Mechanisms in Neural Networks with Approximation. In Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), San Diego, CA, USA, 16 April 2020; pp. 328–341. [CrossRef]
7. Ham, T.J.; Lee, Y.; Seo, S.H.; Kim, S.; Choi, H.; Jung, S.J.; Lee, J.W. ELSA: Hardware-Software Co-design for Efficient, Lightweight Self-Attention Mechanism in Neural Networks. In Proceedings of the 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), Valencia, Spain, 4 August 2021; pp. 692–705. [CrossRef]
8. Zhang, X.; Wu, Y.; Zhou, P.; Tang, X.; Hu, J. Algorithm-hardware Co-design of Attention Mechanism on FPGA Devices. *ACM Trans. Embed. Comput. Syst.* **2021**, *20*, 1–24. [CrossRef]
9. Lu, S.; Wang, M.; Liang, S.; Lin, J.; Wang, Z. Hardware Accelerator for Multi-Head Attention and Position-Wise Feed-Forward in the Transformer. In Proceedings of the 2020 IEEE 33rd International System-on-Chip Conference (SOCC), Las Vegas, NV, USA, 6 September 2021; pp. 84–89. [CrossRef]
10. Cao, S.; Zhang, C.; Yao, Z.; Xiao, W.; Nie, L.; Zhan, D.; Liu, Y.; Wu, M.; Zhang, L. Efficient and Effective Sparse LSTM on FPGA with Bank-Balanced Sparsity. In Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA, 24–26 February 2019; pp. 63–72. [CrossRef]
11. Wang, H.; Zhang, Z.; Han, S. Spatten: Efficient sparse attention architecture with cascade token and head pruning. *arXiv* **2020**, arXiv:2012.09852. [CrossRef]
12. Qi, P.; Song, Y.; Peng, H.; Huang, S.; Zhuge, Q.; Sha, E.H.-M. Accommodating Transformer onto FPGA: Coupling the Balanced Model Compression and FPGA-Implementation Optimization. In Proceedings of the 2021 on Great Lakes Symposium on VLSI, Virtual Event, USA, 22–25 June 2021; pp. 163–168. [CrossRef]
13. Peng, H.; Huang, S.; Geng, T.; Li, A.; Jiang, W.; Liu, H.; Wang, S.; Ding, C. Accelerating Transformer-based Deep Learning Models on FPGAs using Column Balanced Block Pruning. In Proceedings of the 2021 22nd International Symposium on Quality Electronic Design (ISQED), Santa Clara, CA, USA, 7–9 April 2021; pp. 142–148. [CrossRef]
14. Khan, H.; Khan, A.; Khan, Z.; Bin Huang, L.; Wang, K.; He, L. NPE: An FPGA-based Overlay Processor for Natural Language Processing. *arXiv* **2021**, arXiv:3431920.3439477. [CrossRef]
15. Li, B.; Kong, Z.; Zhang, T.; Li, J.; Li, Z.; Liu, H.; Ding, C. Efficient Transformer-based Large Scale Language Representations using Hardware-friendly Block Structured Pruning. *arXiv* **2020**, arXiv:2009.08065.
16. Narang, S.; Undersander, E.; Diamos, G. Block-sparse recurrent neural networks. *arXiv* **2017**, arXiv:1711.02782.
17. Jean, S.; Firat, O.; Cho, K.; Memisevic, R.; Bengio, Y. Montreal Neural Machine Translation Systems for WMT'15. In Proceedings of the 10th Workshop on Statistical Machine Translation, Lisbon, Portugal, 17–18 September 2015; pp. 134–140.
18. Papineni, K.; Roukos, S.; Ward, T.; Zhu, W.-J. BLEU: A method for automatic evaluation of machine translation. In Proceedings of the 40th annual meeting of the Association for Computational Linguistics, Philadelphia, PA, USA, 6–12 July 2002; pp. 311–318.
19. Liu, Z.; Li, G.; Cheng, J. Hardware Acceleration of Fully Quantized BERT for Efficient Natural Language Processing. In Proceedings of the 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 1–5 February 2021.