Algorithm-hardware Co-design of Attention Mechanism on FPGA Devices

XINYI ZHANG, YAWEN WU, PEIPEI ZHOU, XULONG TANG, and JINGTONG HU,
University of Pittsburgh, USA

Multi-head self-attention (attention mechanism) has been employed in a variety of fields such as machine translation, language modeling, and image processing due to its superiority in feature extraction and sequential data analysis. This is benefited from a large number of parameters and sophisticated model architecture behind the attention mechanism. To efficiently deploy attention mechanism on resource-constrained devices, existing works propose to reduce the model size by building a customized smaller model or compressing a big standard model. A customized smaller model is usually optimized for the specific task and needs effort in model parameters exploration. Model compression reduces model size without hurting the model architecture robustness, which can be efficiently applied to different tasks. The compressed weights in the model are usually regularly shaped (e.g. rectangle) but the dimension sizes vary (e.g. differs in rectangle height and width). Such compressed attention mechanism can be efficiently deployed on CPU/GPU platforms as their memory and computing resources can be flexibly assigned with demand. However, for Field Programmable Gate Arrays (FPGAs), the data buffer allocation and computing kernel are fixed at run time to achieve maximum energy efficiency. After compression, weights are much smaller and different in size, which leads to inefficient utilization of FPGA on-chip buffer. Moreover, the different weight heights and widths may lead to inefficient FPGA computing kernel execution. Due to the large number of weights in the attention mechanism, building a unique buffer and computing kernel for each compressed weight on FPGA is not feasible. In this work, we jointly consider the compression impact on buffer allocation and the required computing kernel during the attention mechanism compressing. A novel structural pruning method with memory footprint awareness is proposed and the associated accelerator on FPGA is designed. The experimental results show that our work can compress Transformer (an attention mechanism based model) by 95x. The developed accelerator can fully utilize the FPGA resource, processing the sparse attention mechanism with the run-time throughput performance of 1.87 Tops in ZCU102 FPGA.

CCS Concepts: • **Hardware → Hardware-software codesign**; • **Computing methodologies → Machine translation**; • **Computer systems organization** → *Embedded hardware;*

Additional Key Words and Phrases: Co-design, algorithm, hardware, attention, transformer, FPGA, tops

This work is partially supported by National Science Foundation under Grant Numbers NSF IIS-2027546.
This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2021.
Authors' addresses: X. Zhang, Y. Wu, P. Zhou, and J. Hu, Swanson School of Engineering, Electrical and Computer Engineering, University of Pittsburgh, Pittsburgh, PA, USA, 15260; emails: {xinyizhang, yawen.wu, peipei.zhou, jthu}@pitt.edu; X. Tang, School of Computing and Information, Computer Science, University of Pittsburgh, Pittsburgh, PA, USA, 15260; email: xulongtang@pitt.edu.

1539-9087/2021/09-ART71 $15.00
https://doi.org/10.1145/3477002

ACM Reference format:
Xinyi Zhang, Yawen Wu, Peipei Zhou, Xulong Tang, and Jingtong Hu. 2021. Algorithm-hardware Co-design of Attention Mechanism on FPGA Devices. *ACM Trans. Embedd. Comput. Syst.* 20, 5s, Article 71 (September 2021), 24 pages.
https://doi.org/10.1145/3477002

1 INTRODUCTION

Attention mechanism has been widely adopted as the backbone in neural network models such as Transformer [1] in machine translation, BERT [2] in language modeling, GPT [3] in the general language model, and DETR [4] in image processing. The size of the attention-based model ranges from million bytes (Transformer) to billion (GPT-3) bytes.

While large scale attention-based neural network models are developed to break the records in the Natural Language Processing (NLP) tasks, the computations become more and more intensive. As the majority of the computations in the attention mechanism are matrix multiplications, it has been reported that 10 Giga multiply-accumulate operations (MAC) are needed when translating a short sentence via Transformer [5]. With such a large number of weights and MACs, a large memory footprint and high computational cost are demanded when deploying the mechanism.

Existing works proposed to structurally prune the model weight to keep computing efficiency. The memory footprint and workload are largely reduced after pruning. Such method focuses on removing the redundant weights without hurting the robustness of the model architecture, which also avoids the efforts in proposing new models. The efficient pruning method for attention mechanism has been observed in TransformerZip [6], HAT [7], and FTRANS [8]. TransformerZip [6] utilizes magnitude-based pruning to reduce weight size; HAT [7] crops the weights in both dimensions to reduce weight size and form regularly shaped weights. FTRANS [8] utilized block-circulant matrix to replace selected weights, which reduces the model memory footprint.

When programming and deploying weight-pruned models in the inference stage on generic processors, e.g., CPUs and GPUs, it incurs little effort. However, it poses significant programming efforts and design challenges on hardware accelerators like FPGA for the following two reasons: First, the dimension size of different weights can be arbitrary after compression as the significance or absolute value differs among the weight elements. It is challenging to efficiently allocate on-chip buffers for different shapes of weight under on-chip hardware constraints to maximize buffer utilization and improve inference throughput. Second, the FPGA accelerator is usually computing pattern-specific. While the compression causes arbitrary-sized weight and its associate computing pattern, building a dedicated computing kernel for each pattern is not feasible. As buffer allocation and computing kernel design on FPGAs are usually fixed in size and limited in number for specific computations, these two issues may severely hurt the FPGA efficiency. Therefore, the attention mechanism's memory footprint and hardware computing pattern should be jointly considered and optimized when deploying the attention mechanism on FPGAs.

In this paper, we propose a novel algorithm-hardware co-design framework to address these issues. To illustrate the framework, we take Transformer as a vehicle since it is a complete set of attention mechanism modules. First, in the algorithm design, we propose a novel structural weight pruning method for training Transformer models on generic processors by applying tighter restrictions on both weight shape and size. The size variation among the compressed weights is controlled in this step. Then, in hardware design, we propose a unified computing pattern that can process the sparse matrix multiplications in deploying Transformer inference stage. Based on the compressed model and the unified computing pattern, we develop an FPGA accelerator that can fully utilize FPGA resources and achieve 1.87 Tops (Tera Operations Per Second) throughput performance. Our contributions are summarized as follows:

- **Memory footprint aware compression.** We propose a novel pruning method that prunes the weight column by column, leaving the weight height unchanged. During the compression, the size variation among weights is controlled and minimized. After adequately pruning the weights, the compressed Transformer is then trained and applied with post-training quantization, forming a sparse INT8 Transformer.
- **Unified computing pattern.** We propose a novel computing pattern for the compressed model that converts the MACs in Transformer to unified element-wise vector multiplications and additions. The computing pattern naturally eliminates the unnecessary computation caused by sparsity. In addition to its arithmetic efficiency, the hardware efficiency is largely improved by packing multiple operations into a single FPGA DSP with the computing pattern.
- **Efficient FPGA accelerator.** We build an accelerator with pipelined processing element (PE) array according to the computing pattern. The accelerator efficiently processes the stream-in data in high parallelism. The accelerator can be recursively called to process the matrices in Transformer with near-zero resource under-utilization and high throughput.
- **Performance validation and accelerator implementation on FPGA.** We validate that the proposed compression method can compress a Transformer model by 95X without accuracy loss. The proposed accelerator is deployed on Xilinx ZCU102 FPGA, achieving 99% computing resource utilization and run-time throughput at 1.87 Tops.

The remainder of the paper is organized as follows. Section 2 and Section 3 presents the background of the attention mechanism and the related works. Section 4 presents the motivation of our proposed algorithm-hardware co-design framework for the attention mechanism. Section 5 presents the memory footprint aware compression algorithm. Section 6 presents the unified computing pattern and its benefits. Section 7 presents the accelerator hardware design. Section 8 presents the performance, resource, and energy efficiency evaluation of our proposed framework.

2 BACKGROUND

Attention mechanism has drawn the focus of the NLP community in recent years. With the capability of processing the long sequence in parallel, the attention mechanism becomes the most popular method in sequence modeling. The Transformer is the first self-attention based model. GPT series, BERT, and DETR further extend the model capacity and improve the precision in the NLP tasks including machine translation, information extraction, conversational agents, etc. However, they all share the same attention mechanism as Transformer. In this section, we take Transformer as the demonstration model to study the attention mechanism.

The core of the attention mechanism is self-attention, which is also called scaled dot-product. The scaled dot-product is shown in Equation (1) [1], in which, an input $X^{in} \in \mathbb{R}^{N \times d_{model}}$ is mapped to an output $O^{attn} \in \mathbb{R}^{N \times d_k}$ via Query (Q), Key (K), and Value (V). The Q, K, and V are intermediate results that are acquired by multiplying X^{in} with corresponding weights Q^w, K^w, and V^w in the same size ($w \in \mathbb{R}^{d_{model} \times d_k}$). Therefore, the computations can be roughly divided into two parts: the Q, K, and V mapping and scaled dot-product as shown in Figure 1(a). In a self-attention, N and d_{model} are determined by input dataset and d_k is one of the model hyperparameters that vary in different models. The X^{in} multiplies with corresponding weights to acquire Q, K, and V. Then, the intermediate result Q multiplies with the transpose of K (K^T). After division and softmax, the result QK multiplies with V to get O^{attn}. The overview of self-attention operations is shown in Figure 1(b), in which, the "linear" in Figure 1(a) is denoted by MatMul. The multiple parallel instances of self-attention form multi-head self-attention. Its main computations and the

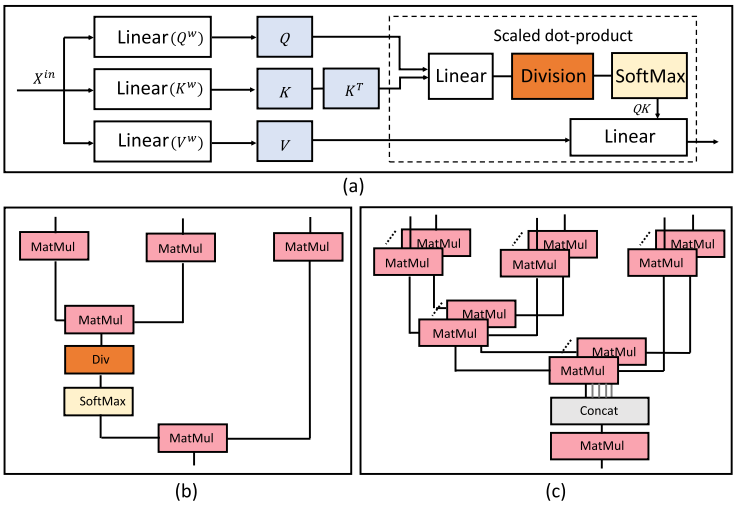

Fig. 1. (a) Computations in self-attention. (b) Single head self-attention. (c) Multi-head self-attention.

data flow of the multiple heads is shown in Figure 1(c). Multiple heads share the same input X^{in} but own private Q^w, K^w, V^w. Generally, Q^w, K^w, V^w in an encoder/decoder layer are concatenated in width direction in computation, forming $Q^w_{layer}, K^w_{layer}, V^w_{layer} \in \mathbb{R}^{d_{model} \times (N_{head} * d_k)}$. After the computation for each head, the individual head output O^{attn} are concatenated and mapped to multi-head output O^{head} via weight O^w.

$$O^{attn} = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

The attention mechanism is further assembled as an encoder and decoder, forming the main blocks of a Transformer as shown in Figure 2. In an encoder, addition&normalization and feed-forward network (FFN) are also placed. The FFN consists of two stacked linear modules. Compared to encoder in model architecture, decoder has an extra masked-attention module to build the data dependency in the output sequence. In [1], the d_k is 64, the d_{model} is 512, the number of heads (N_{head}) is 8, the number of the encoders (N_{enc}) is 6, and the number of the decoders (N_{dec}) is 6. The encoders are sequentially connected and decoders are connected likewise. The last encoder feeds the intermediate data into the decoders. The input data $X = (x_1, x_2, ..., x_N)$ is embedded into X^{in} and processed by encoders first. The intermediate representation $Z^{in} \in \mathbb{R}^{N \times d_{model}}$ generated by the last encoder is further fed into N_{dec} decoders.

As a result, the weights in a Transformer encoder/decoder layer can be summarized in **six types**: $Q^w_{layer}, K^w_{layer}, V^w_{layer} \in \mathbb{R}^{d_{model} \times (N_{head} * d_k)}$, $O^w \in \mathbb{R}^{d_{model} \times d_{model}}$, $FFN1^w \in \mathbb{R}^{d_{model} \times 4d_{model}}$, and $FFN2^w \in \mathbb{R}^{4d_{model} \times d_{model}}$. With the multi-head self-attention, encoders, and decoders, the number of weights can be up to hundreds and the total memory footprint size for the Transformer is 176 MB [1]. The large size of memory requirement and various sizes of the weights make it challenging when deploying Transformer on embedded devices as the memory size and computing resource are usually limited on such platforms.

3 RELATED WORKS

To reduce the Transformer weight size, existing works focus on compressing the weights Q^w, K^w, V^w, and FFN^w. In compression, structured pruning is widely adopted as it forms a regularly

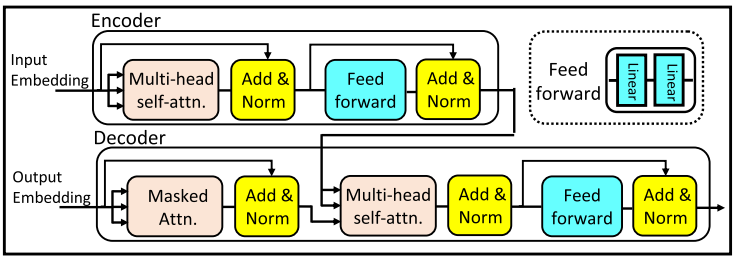

Fig. 2. Transformer encoder and decoder.

shaped weight that is efficient in computing [6–11]. Wang et al. [7] proposed structured pruning for Transformer targeting on CPU and GPU. By adopting neural architecture search, it structurally cropped each weight in Transformer and generated weights in regular shape but with arbitrary dimension size. The compressed weights Q^w, K^w, V^w, and FFN^w in [7] are shown in Figure 3(a) and (c). The pruned weight elements are white-colored. While the model size can be efficiently reduced, the compressed weigh size differs significantly. In computation, such a method heavily relies on the memory and computing resource flexibility of generic processors.

Li et al. [8] utilized block-circulant matrix to replace selected weights, efficiently reducing the model size. However, extra FFT processing kernel is needed. It also builds dedicated computing kernels for weight Q^w, K^w, and V^w on FPGA, leading to heterogeneous computing kernels. The overview of accelerator [8] is shown in Figure 4(a). In this design, much effort in building heterogeneous computing elements and balancing the pipeline stage is observed while its system performance is inferior. Xilinx [12] accelerated the matrix multiplications in attention mechanism on an industry-level FPGA with high bandwidth memory (HBM). The data movement overhead caused by large model size is relieved by HBM. During computing, a DSP systolic array is recursively invoked to process the matrix multiplications. However, the HBM technique is usually not available in embedded FPGAs. The large model size still needs to be addressed before deploying on embedded FPGAs.

When accelerating the large deep learning models, recursively calling the accelerator is efficient for resource constraint FPGA. Such mechanism is widely adopted in Convolution Neural Network (CNN) accelerator designs [13–20]. Such designs usually rely on the processing element (PE) array as the computing kernel. In this way, the accelerator can be re-purposed for different CNN applications. Two typical designs are shown in Figure 4(b) and (c). Design [14] proposed a throughput optimized accelerator that builds a PE as a multiply-accumulate unit which is shown in Figure 4(b). The PEs operate as a systolic array, in which, each PE has fixed communication availability with its neighbor PE. Design [20] proposed a latency and energy-optimized in-sensor accelerator which builds a PE as a group of multiply-accumulate. The data flow is further optimized to achieve low latency. Despite the existing CNN accelerators achieve superior performance in throughput or latency, they can not be directly applied in Transformer applications due to the significant difference in model architecture. How to efficiently accelerate Transformer application on FPGA is still not fully solved.

4 MOTIVATION EXAMPLE

The existing works proposed efficient methods to address the large memory footprint size of weights. However, the weight compression and computation are not jointly considered, which leads to inefficient utilization of on-chip memory and compute resources, resulting in computing inefficiency. Figure 3(a) and (b) show the compression result of one self-attention in [7]. The in-

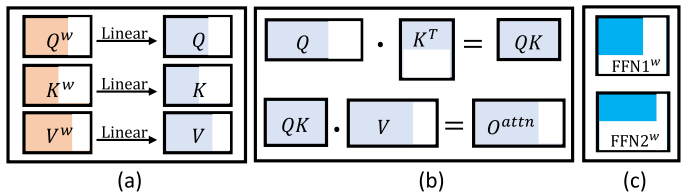

Fig. 3. Compression visualization in related works. (a) Compressed weights Q^W, K^W, V^W and its result Q, K, and V. (b) Compressed $Q * K^T$ and $QK * V$. (c) Compressed FFN layers.

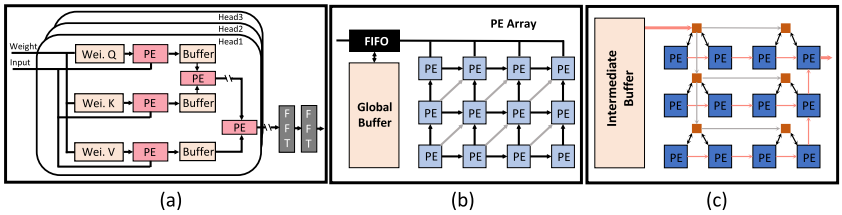

Fig. 4. Existing accelerator architectures. (a) FTRANs accelerator [8]. (b) Eyeriss CNN accelerator [14]. (c) In-sensor CNN accelerator [20].

termediate results related to the pruned weights are also white-colored for better illustration. [7] crops the columns of Q^w, K^w, and V^w. However, the width of compressed weight is not controlled, which also directly influences the MAC size of $Q * K^T$ as shown in Figure 3(b). The linear layer weights in FFN are even arbitrary in both height and width which are shown in Figure 3(c). As an attention mechanism consists of 8 heads and the Transformer consists of 6 encoders and 6 decoders, the number shapes of the compressed weights and the associated computation size can be as high as several hundred. Such compression method causes severe buffer inefficient usage on FPGA. Since MAC is usually executed in parallel on FPGAs, the suitable computing kernel size also varies for those compressed weights. As a result, the solution adopted in [8] will cause extreme unbalance computing kernel design and difficult pipeline arrangement. The accelerator system performance may be largely degraded.

In this work, targeting on machine translation task in NLP domain, we propose to compress the Transformer model with weight size awareness, leaving weights in similar size. The similarity of the compressed weights brings more efficient FPGA buffer utilization. A novel computing pattern is also proposed to address the computing heterogeneity and inefficiency in the Transformer after compression. With the compression methodology and the unified computing pattern, an accelerator is designed to accelerate the sparse matrix multiplications of Transformer ('MatMul' and Linear layers in Figure 1 and Figure 2). The accelerator is recursively called in a mode of streaming in, computing, and streaming out. In this way, the accelerator with the fixed buffer and the unified computing core is efficient to handle the deployment of sparse Transformer.

5 MODEL COMPRESSION

In this section, we present a novel compression method that structurally prunes the weights. This method forms compressed weight in similar size to achieve efficient deployment of the buffer and computing engine. The pruned model can maintain accuracy when quantizing weight from floating-point to INT8 while reducing memory footprint and MACs significantly. The compression involves two aspects: the weight significance analysis and memory footprint aware pruning.

5.1 Weight Significance Analysis

The weight significance is first acquired according to the means of "Winning Ticket Hypothesis' [21]. 'Winning ticket' theory proves that the sub-network of a model can have comparable accuracy by correctly removing the smallest-magnitude weights and training from original initialization (one-shot pruning [21]). The 'winning ticket' theory has been adopted and validated in the existing works for CNN models [21–24], in which, [22] identifies Normalization is efficient in identifying the CNN weight magnitude channel-wise. However, Transformer has a different architecture with CNN models, the existing techniques can not be directly applied to Transformer. In this work, we propose a novel 'winning ticket' finder, in which, layer normalization (LayerNorm) [25] is adopted to firstly analyze the weight significance in column-wise.

ALGORITHM 1: Weight significance analysis

Input: dataset (*data*), Transformer (*model*),
Transformer total layer L, $LayerNorm$.
Output: $model_{norm}$.
$model_{norm} = model$;
for $l \leftarrow 1$ **to** L **do**
  $model_l = model$;
  for $module \in model_l.layer[l]$ **do**
    if $module.weight$ **then**
      $module.attach(LayerNorm)$;
    end
  end
  while $LayerNorm^* \in model_l$ ***Not*** *Converged* **do**
    $model_l.train$;
  end
  $model_{norm}.layer[l] = model_l.layer[l]$;
end

The developed analysis workflow is shown in Algorithm 1. LayerNorm is exclusively applied to one encoder or one decoder layer at a time (denoted as $model.layer$ in the algorithm), in which, LayerNorm is attached to any matrix multiplication that contains weight ($module.weight$ in the algorithm) in the layer. The Transformer is then trained until the scaling factors γ in the newly attached LayerNorms are converged. The converged γ is collected as the column significance indicator for the weight. After repeatedly applying LayerNorms to each encoder and decoder layer and training the Transformer, the scaling factors γ in all of the attached LayerNorms are collected and stored in a separated model $model_{norm}$. The visualization of an encoder before and after LayerNorm attachment is shown in Figure 5(a) and (b), in which, the newly attached LayerNorm is labeled as $LayerNorm^*$.

The LayerNorm takes a vector or a matrix as input and scales the row elements individually which is shown in Equation (2). In a LayerNorm, the row elements expectation E and variation Var are employed; the scaling factors γ and β are dedicated for each row element but shared among rows. As visualized in Figure 5(c), a pair of the γ and β can scale a column of the inputs up or down. When attaching the LayerNorm to matrix multiplication, a column of the results will be normalized by the same γ and β. If looking backward, which is shown in Figure 5(d), a column of the result is determined by the corresponding column of weight. Therefore, a pair of the scaling factor γ and β can reflect the significance of the corresponding column of the weight. As γ dominates the scaling

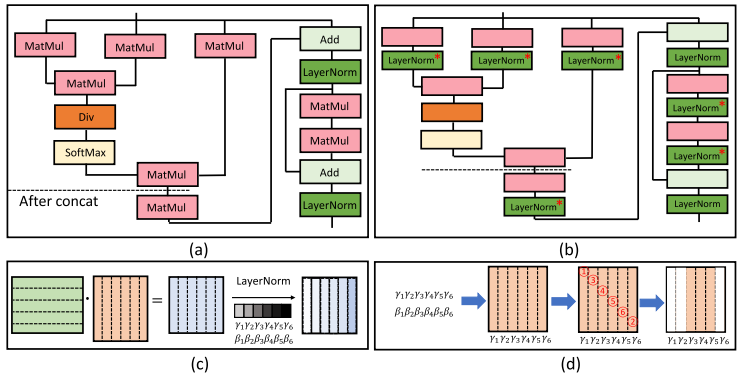

Fig. 5. (a) Encoder without LayerNorm insertion. (b) LayerNorm insertion of the encoder. (c) LayerNorm scaling factor γ and β. (d) Weight significance based on γ.

operation, we take γ value as a significance indicator. The scaling degree can be reflected by γ value. The weight significance can be ranked via its corresponding γ while weight with larger γ is more important. According to the significance ranking, the columns of the weights with smaller γ will be pruned. In the next step, the method to prune the weight in columns-wise with memory footprint awareness will be illustrated.

$$x_{scaled} = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} * \gamma + \beta \tag{2}$$

5.2 Pruning Strategy

To ensure the hardware efficiency after pruning, a two-stage one-shot pruning is performed: coarse-grained pruning to keep the weights of the same type in similar size; fine-tune to adequately remove the redundant weights without losing accuracy. In each stage, pruning and training are pairwise performed until the model is adequately pruned within the stage.

Coarse-grained: the coarse-grained pruning prunes each of the weights in the Transformer with the same ratio. During this stage, the memory footprint size of the weights is evenly reduced. Based on the dataset $data$, the Transformer $model$, the collected γ data in $model_{norm}$, and a pre-set pruning speed $incr_{coarse}$, coarse-grained pruning will determine the maximum even pruning ratio. As shown in Algorithm 2, the baseline accuracy is acquired first. Then, the pruning ratio increases by $incr_{coarse}$ from 0%. At each pruning ratio, **GetIndex** will locate the index of γ ranging in the least $ratio$% in a LayerNorm. Such index is equivalent to the index of weight columns. According to the selected index, **Mask** will mask the indexed weight columns to enable zero gradient descent during training. The increment of $ratio$ stops when the sparse model accuracy drops below the baseline accuracy ($accuracy_{base}$). After this stage, the maximum size of each weight is bounded.

Fine-tune: In this stage, the masked columns in stage one are considered as already been removed in both $model$ and $model_{norm}$. As shown in Algorithm 3, fine-tune takes $data$, coarsely pruned $model$, $accuracy_{base}$, $model_{norm}$, and $incr_{fine}$ as input, performing across layer pruning based on the rest γ. In fine-tune, encoders and decoders are pruned separately. Algorithm 3 takes encoders as an example since encoder and decoder are similar in architecture. The γ of LayerNorms in $model_{norm}$ for the same type weights cross all encoder layers are grouped first. For example, the γ for Q^w ($layer.module.LayerNorm$ in the algorithm) in six encoder layers are grouped. After collecting the cross-layer γ for each type of weight, γ for corresponding weights are stored

ALGORITHM 2: Coarse-grained pruning

Input: $data$, $model$, $model_{norm}$, $incr_{coarse}$.
Output: Pruned $model$, $accuracy_{base}$, $ratio_{coarse}$.
$model.train$;
$accuracy_{base} = model.accuracy$;
$ratio = 0\%$;
while $True$ **do**
  $ratio = ratio + incr_{coarse}$;
  for $l \leftarrow 1$ **to** L **do**
    $Norm_{layer} = model_{norm}.layer[l]$;
    for $module \in model.layer[l]$ **do**
      if $module.weight$ **then**
        $GetIndex(Norm_{layer}.module.LayerNorm, ratio)$;
        $Mask(model.layer[l]).module.weight)$;
      end
    end
  end
  $model.train$;
  if $model.accuracy < accuracy_{base}$ **then**
    $ratio_{coarse} = ratio - incr_{coarse}$;
    break;
  end
end
/* **GetIndex**: locating the index of γ ranging in the least $ratio$ percent. */
/* **Mask**: masking the selected column to ensure zero gradient descent in training. */

in $EncQ_{norm}$, $EncK_{norm}$, $EncV_{norm}$, $EncO_{norm}$, $EncFn1_{norm}$, and $EncFn2_{norm}$. The pruning and training will be executed similarly as in Algorithm 2. However, in this algorithm, **GetIndex** will locate the index of rest γ ranging in the least ratio% in the group. For example, if the *model* consists of six encoders, the γ in all of the encoders are compared. **Mask** also performs cross-layer masking in this stage. After this stage, the Transformer model size is further reduced with slight size variation.

The two-stage pruning adequately prunes the model without accuracy loss. The pruned weights in Transformer are illustrated in Figure 6(a), (b), (c), and (d), in which, the pruned weights and corresponding results are white-colored for better illustration. As a result, the weights in same type (e.g., $Q^w_{layer}, K^w_{layer}, V^w_{layer}, O^w, FFN1^w$, and $FFN2^w$) in the Transformer will be in similar size. This will maximize the utilization efficiency of the data buffer on hardware. The well-trained sparse *model* can be directly quantized to INT8 data via dynamic quantization with Pytorch [26] without accuracy loss. After quantization, the model size can be further reduced by 4x, which benefits a smaller memory footprint and more efficient hardware computing.

6 UNIFIED COMPUTING PATTERN DESIGN

In this section, we first present the optimization objectives for the computations in compressed Transformer. Then, the proposed unified computing pattern and its benefits are presented.

6.1 Optimization Objectives

After compression, the memory footprint and workload can be efficiently reduced. Though the proposed compression method structurally prunes the weight which minimizes the impacts to

ALGORITHM 3: Fine-tune pruning

Input: *data*, coarsely pruned *model*, $accuracy_{base}$, $model_{norm}$, $incr_{fine}$.
Output: Fine pruned *model*.

```
/* Encoders grouping.;                                                    */
EncQ_norm; EncK_norm; EncV_norm; EncO_norm; EncFn1_norm; EncFn2_norm;
for layer in model_norm.EncoderLayers do
    for module ∈ layer do
        if module.weight then
            switch module.Name do
                case Q do
                    EncQ_norm.append (layer.module.LayerNorm);
                case K do
                    EncK_norm.append (layer.module.LayerNorm);
                case V do
                    EncV_norm.append (layer.module.LayerNorm);
                case O do
                    EncO_norm.append (layer.module.LayerNorm);
                case Fn1 do
                    EncFn1_norm.append (layer.module.LayerNorm);
                case Fn2 do
                    EncFn2_norm.append (layer.module.LayerNorm);
            end
        end
    end
end
/* End Encoders grouping.;                                                */
/* Decoders grouping is similar to encoder (skipped).;                    */
...;
/* End Decoders grouping.;                                                */
ratio = 0%;
while True do
    ratio = ratio + incr_fine;
    /* Encoders tuning;                                                   */
     GetIndex(EncQ_norm, ratio); Mask(model.EncQ.weight);
     GetIndex(EncK_norm, ratio); Mask(model.EncK.weight);
     GetIndex(EncV_norm, ratio); Mask(model.EncV.weight);
     GetIndex(EncO_norm, ratio); Mask(model.EncO.weight);
     GetIndex(EncFn1_norm, ratio); Mask(model.EncFn1.weight);
     GetIndex(EncFn2_norm, ratio); Mask(model.EncFn2.weight);
    /* Decoders tuning skipped;                                           */
    /* End tuning;                                                        */
    model.train;
    if model.accuracy < accuracy then
        ratio = ratio − incr_fine;
        break;
    end
end
/* GetIndex: locating the index of γ ranging in the least ratio% in the group.  */
/* Mask: masking the selected column in the group to ensure zero gradient descent in
   training.                                                              */
```

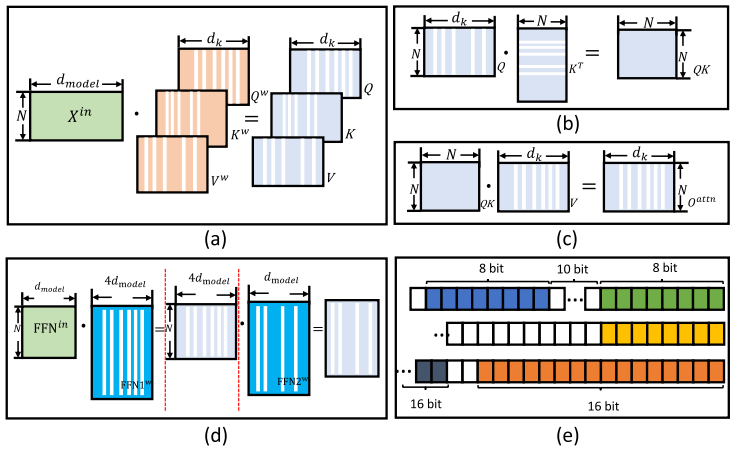

Fig. 6. (a) Sparse pattern of weight Q, K, and V in the proposed technique. (b) Sparse pattern of $Q * K^T$. (c) Sparse pattern of $QK * V$. (d) Sparse pattern in FFN operations. (e) INT8 multiplication encoding overview.

weight shape and the size difference among weights, the computing inefficiency caused by diverse computation sizes is still unresolved. And the side effect in computation caused by sparsity exists. In summary, there are four challenges in accelerating the sparse Transformer.

***O*1: Multi-size multiply-accumulate**: as shown in Figure 6(a), (b), (c), and (d), different sizes of MAC (d_{model}, d_k, N, and $4d_{model}$) still exist in corresponding matrix multiplications. As the value of such parameters differs significantly (d_{model} = 512, d_k = 64, N = 100 [1]), if the hardware multiply-accumulate kernel is not carefully designed, mapping different sizes of MAC in a network onto the same piece of hardware may lead to unsatisfactory overall performance.

***O*2: Inefficient INT8 computation**: DSP components in modern FPGAs are usually designed with high bitwidths such as $27b * 18b \rightarrow 45b$ [27], which is redundant for an INT8 operation. The DSP can only be fully utilized by encoding INT8 multiplication as $((a <<) + b) * c$ [27] which is shown in Figure 6(e). However, the shared multiplier c for multiplicands a and b do not exist in the conventional multiply-accumulate.

***O*3: Multiplying with zero**: since both Q and K are sparse in columns, the *multiplying with zero* occurs when multiplying Q with K^T as illustrated in Figure 6(b). When the compression ratio goes higher, *multiplying with zero* can cause severe under-utilization of computing resources.

***O*4: Compressed matrix restoration**: though the memory footprint can be reduced by only keeping the non-pruned weights, the sparse result needs to be restored to ensure the data consistency between matrices such as $Q * K^T$. The matrix restoration may diminish the benefit of the smaller memory footprint after model compression.

6.2 Computing Pattern Optimization

With the observation that compression forms fixed height (N) for multiplicand of the matrix multiplications in the model, the MAC of different sizes can be replaced by a sequence of uniform element-wise vector multiplication and addition. The unified computing pattern is shown in Figure 7, in which IN and WEI represents the multiplicand and multiplier, and OUT represents the product. OUT is computed column by column. Each column of OUT is partially computed by multiplying an element in WEI and its corresponding column of IN. The accumulation of column one is shown in this Figure (the element is fan-out to the vector). The pseudocode of this process is also shown in Figure 8, where the inner loop computes the partial result of a column in OUT and can execute in parallel on FPGA by loop unrolling. After the iterations of the middle loop,

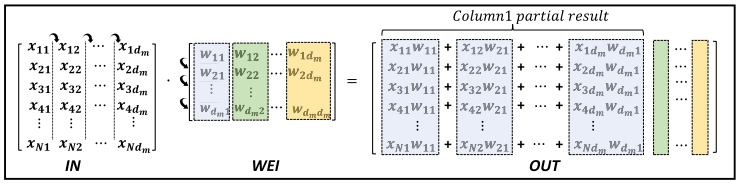

Fig. 7. Unified computing pattern in element-wise multiplication and addition.

```
for (col_wei=0; col_wei< C_WEI; col_wei++) {
   for (row_wei=0; row_wei< R_WEI; row_wei++) {
      for (row_in=0; row_in< R_IN; row_in++)        {
         OUT[row_in][col_in] +=IN[row_in][row_wei]*WEI[row_wei][col_wei]
} } }
/*  R represents the number of rows  */
/*  C represents the number of columns */
```

Fig. 8. The pseudocode of the computing pattern optimization in a manner of loop iteration.

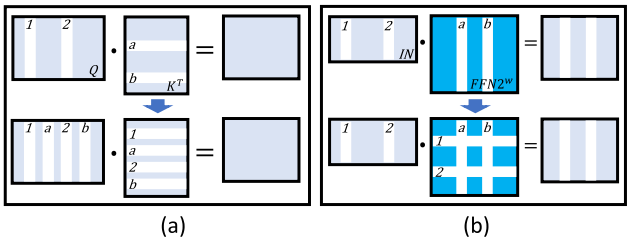

Fig. 9. Multiplicand and multiplier data alignment in (a) sparse $Q * K^T$ (b) sparse linear layer.

a column of OUT is computed. As highlighted in the loop iterations, the iterations of the inner loop are the height of the multiplicand which is fixed (N) cross all the matrix multiplications in Transformer; the $WEI[row_{wei}][col_{wei}]$ is a constant in the inner loop iterations. In this way, $\boldsymbol{O}1$ and $\boldsymbol{O}2$ are achieved. As for $\boldsymbol{O}1$, the MAC of different sizes is unified to single size element-wise vector multiplication first (inner loop). The accumulations are converted to single-size element-wise addition (middle loop). As for $\boldsymbol{O}2$, as the element $WEI[row_{wei}][col_{wei}]$ is constant in the inner loop iterations, it is shared by the multiplicand vector element. Therefore, the DSP bitwidth can be fully utilized by encoding the input vector as $((a <<) + b) * WEI[row_{wei}][col_{wei}]$.

$\boldsymbol{O}3$ and $\boldsymbol{O}4$ are naturally eliminated by the proposed computing pattern since computations are column-based and the pruned columns are not stored. Benefited from the computing pattern, the memory footprint can be further reduced as the column of the multiplicand needs to align with the multiplier's row. The corresponding two cases are shown in Figure 9. In this figure, the pruned columns in the multiplicand and multiplier are labeled in number and letter, respectively. Figure 9(a) illustrates the alignment between Q and K. As Q's column needs to align with K^T's row element to guarantee one-to-one mapping, columns a and b of Q are further removed; rows 1 and 2 of K^T are further removed. Case (b) shows $FFN2$, in which, the columns a and b of $FFN2^w$ were already pruned. Its rows 1 and 2 are further removed to align with its multiplicand's columns. As the sparsity is determined after compression, the alignment process can also be executed off-line before inference. Thus, the memory footprint and workload can be further reduced.

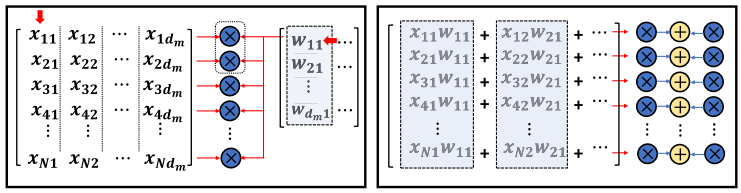

Fig. 10. (a) The PE1 architecture and PE1 mapping to the multiplication in the unified computing pattern. (b) The PE2 architecture and PE2 mapping to the addition in the unified computing pattern.

7 ACCELERATOR DESIGN

In this section, we first present the accelerator architecture which includes processing element, computing core, buffer allocation, buffer partition, and accelerator running schedule. Then, an analytical model is built to determine the accelerator parameters and predict the system performance.

7.1 Accelerator Architecture

Processing element. The diverse computations are unified into patterns: element-wise vector multiplication and addition, which can be conducted by only two types of Processing Elements (PE). The element-wise vector multiplication between a multiplicand's column and an element of the multiplier can be executed via multiplication unit $PE1$, in which, the parallelism of N ($N = R_{IN}$) can be performed (illustrated by the inner loop of Figure 8). The architecture of $PE1$ and $PE1$ to IN and WEI access is shown in Figure 10(a). In $PE1$, the multiplications are mainly implemented via FPGA DSPs and every two of the multiplications are packed into one DSP according to the INT8 data packing mechanism illustrated in Section 6.2. The partial results regarding all multiplicand's columns can be accumulated via addition unit $PE2$ in the parallelism of N as shown in Figure 10(b). $PE2$ is implemented via FPGA LUT as INT8 addition is more efficient with LUT resources. More $PE1$ can be built with FPGA LUT resource as long as the LUT resource is abundant. By building and connecting multiple $PE1$ and $PE2$, the unified computations can be efficiently processed.

Computing core. As the height of the multiplicand of matrix multiplications in Transformer is fixed at N, a homogeneous $PE1$ array and homogeneous $PE2$ array are utilized to build a multi-stage pipeline computing core. The PE array processes the unified computations in high parallelism. The multiplicand's columns can be processed simultaneously via $PE1$ array and accumulated by the following tree-structured $PE2$ array. At the lowest hierarchy of $PE2$ tree, an additional $PE2$ accumulates the partial results. A 6-stage computing core with 8 $PE1$ and 8 $PE2$ is shown in Figure 11(a), in which, the parallelism of $PE1$ and $PE2$ is set as N. 8 $PE1$ work in parallel and produce 8 vectors in length N. Each $PE2$ in the "adder-tree" structure accumulates two vectors. Buffers are placed between the hierarchies to support pipeline execution. The accumulation for all $PE1$ results is acquired after the data flow reaches the bottom $PE2$. As a result, after $R_{WEI}/8$ executions of the computing core, the first column of the output is acquired.

Since the size of the computing core is bounded by the device resource, a generic data flow of the proposed computing core is shown in Figure 12. The pipeline stage $M + 1$ is determined by the number of $PE1$, where the M stages are spent on computation within PE array and the 1 stage is spent on buffer access. A larger $PE1$ array only leads to a slightly deeper pipeline since each hierarchy in "adder-tree" halves the output of $PE1$. The computing core enter $INITIAL$ to read input and weight elements in the buffer. In $STAGE\_1$, $PE1$ array multiplies the corresponding input columns with weight elements. In the rest $STAGE$, $PE2$ at each hierarchy reads the intermediate results from its higher-level hierarchy and does the accumulation. Such design can

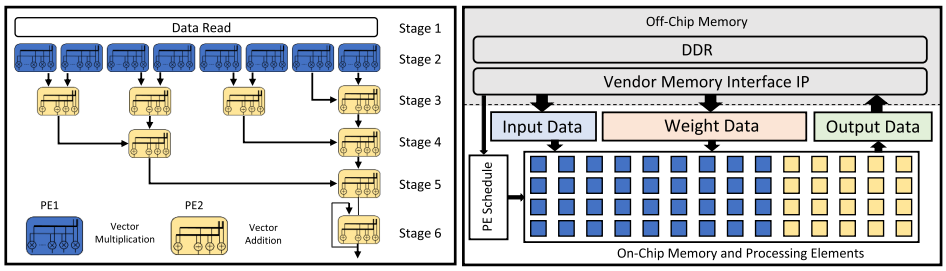

Fig. 11. (a) Computing core hierarchy. (b) Accelerator architecture overview.

Data Fetch	Input, weight	Input, weight	STAGE_1 output	STAGE_2 output	...	STAGE_M-1 output
Computation	N/A	Multiply (PE1)	Add (PE2)	Add (PE2)	...	Add (PE2)
STAGE	INITIAL	STAGE_1	STAGE_2	STAGE_3	...	STAGE_M

Fig. 12. The data flow of the computing core.

maximize the multiplication parallelism and pipeline efficiency. By halving the partial results at each $STAGE$, only $log(2)N_{PE1} + 2$ pipeline stages are needed for the computing core. The pipeline interval of the proposed computing core architecture can run at II=1 during on-board execution.

Accelerator system design. While the memory footprint and the workload are largely reduced by the proposed compression method, the compressed Transformer may still exceed FPGA capacity. Therefore, an accelerator system design that includes modules such as data swapping, data computing, and running scheduler is needed. The proposed full system design is shown in Figure 11(b) in consideration of off-chip memory (DDR), an input data buffer (buf_{in}), a weight data buffer (buf_{wei}), an output data buffer (buf_{out}), PE schedule register, and the computing core. Each buffer utilizes FPGA on-chip block-RAM (BRAM) and LUT-RAM to store the two-dimension data. At each on-chip buffer to DDR interface, a dedicated streaming bus with ping-pong buffer is placed to support continuous processing. The PE schedule register stores the data fetch information and the number of executions (R_{WEI}/N_{PE1}) to get a single column of output and the whole output columns (C_{WEI}). Per computing core execution, N_{PE1} partial results are accumulated. After R_{WEI}/N_{PE1} executions, a column of output is acquired. Therefore, after $(R_{WEI}/N_{PE1}) * C_{WEI}$ execution, the full output elements are acquired. By reading the execution information from PE schedule register, the corresponding address of the input buffer, and the associated elements of weight buffer, the computing core generates the full elements of the output. In this way, the proposed accelerator can be recursively called to process the matrix multiplications in Transformer while minimizing the system stall caused by transmission.

Accelerator buffer allocation. The size of three buffers can be determined by the compressed Transformer size and the computing core. Among the six types of weights, Q^w_{layer}, K^w_{layer}, V^w_{layer}, and O^w are in the same size as $N_{head} * d_k = d_{model}$ in Transformer. And the size $FFN1^w$ and $FFN2^w$ is quadruple of Q^w_{layer}, K^w_{layer}, V^w_{layer}, and O^w. Therefore, before compression, by allocating buf_{wei} of size $\mathbb{R}^{d_{model} \times d_{model}}$, the accelerator can recursively process the computations related to all these weights. During compression, the maximum size of Q^w_{layer}, K^w_{layer}, V^w_{layer}, O^w, $FF1^w$, and $FF2^w$ is determined by $ratio_{coarse}$ in the coarse-grained pruning stage. Though fine-tune pruning unevenly compresses the weights, after compression, the buf_{wei} of size $\mathbb{R}^{d_{model} \times (d_{model} * (1 - ratio_{coarse}))}$ is able to process different weights. After determining the size of buf_{wei}, the size of buf_{in} and buf_{out}

can be determined. The first dimension of buf_{in} is N since height of the multiplicand of matrix multiplications in Transformer is fixed at N. The second dimension of buf_{in} is d_{model} which is the same as buf_{wei}'s first dimension. The first dimension of buf_{out} is N, which is consistent to buf_{in}. The second dimension of buf_{out} is $d_{model} * (1 - ratio_{coarse})$ which is the same as buf_{wei}'s second dimension. The buffer allocation summary is shown in Equation (3).

In this way, the matrix multiplications in Transformer such as $X^{in} * Q^{w}_{layer}$, $X^{in} * K^{w}_{layer}$, $X^{in} * V^{w}_{layer}$, $Q * K^{T}$, $QK * V$, $O^{attn}_{layer} * O^{w}$, $x^{FFN1} * FFN1^{w}$, and $x^{FFN2} * FFN2^{w}$ can be processed via a pipeline processing pattern of streaming in, computing, and streaming out. When the on-chip buffer size is abundant, the capacity of buf_{wei} can be expanded in its second dimension. While the original part $d_{model} * (1 - ratio_{coarse})$ is still used for weight streaming, the expansion part of buf_{wei} can be used to cache selected weights on the chip to reduce off-chip transmission overhead.

$$\begin{aligned} buf_{wei}[R_{WEI}][C_{WEI}] &= buf_{wei}[d_{model}][d_{model} * (1 - ratio_{coarse})] \\ buf_{in}[R_{IN}][C_{IN}] &= buf_{in}[N][d_{model}] \\ buf_{out}[R_{OUT}][C_{OUT}] &= buf_{out}[N][d_{model} * (1 - ratio_{coarse})] \end{aligned} \tag{3}$$

Streaming interface design. As the Transformer's input and part of the weights are stored in the off-chip memory, the accelerator buffer design and transmission bandwidth are also optimized to minimize the transmission latency. Ping-pong buffer is applied on buf_{wei}, buf_{in}, and buf_{out}, forming buf_{wei}, buf_{weiDB}, buf_{in}, buf_{inDB}, buf_{out}, and buf_{outDB} at the interface. The off-chip memory transmission latency is hidden by alternately accessing the ping-pong buffers in two modes. Mode 1: off-chip to buf_{wei} and buf_{in} transmission, and buf_{out} to off-chip transmission while the PE array reads/writes data from/to buf_{weiDB}, buf_{inDB}, and buf_{outDB}. Mode 2: off-chip to buf_{weiDB} and buf_{inDB} transmission, and buf_{outDB} to off-chip transmission while the PE array reads/writes data from/to buf_{wei}, buf_{in}, and buf_{out}. In this way, the PE array access the ping-pong buffers alternately, hiding the off-chip transmission latency. In addition to the ping-pong buffer, the streaming interface bandwidth is also partitioned according to the buffer size. The streaming bandwidth for buf_{wei}, buf_{in}, and buf_{out} are assigned according to the ratio of a buffer capacity to total buffer capacity (e.g. streaming bandwidth for buf_{wei}: $B_{wei} = B_{total} * \frac{Capacity_{b_{wei}}}{Capacity_{buf_{wei}} + Capacity_{buf_{in}} + Capacity_{buf_{out}}}$).

Accelerator buffer partition. The input buffer (buf_{in}), weight buffer (buf_{wei}), and the output buffer (buf_{out}) store two-dimension data via BRAM and LUTRAM. The buffers should also be carefully partitioned to support the parallel access from the computing cores. For input buffer $\boldsymbol{buf_{in}}$, since $PE1$ array accesses multiple columns of input buffer simultaneously, the $buf_{in}[R_{IN}][C_{IN}]$ should be partitioned by parameter N_{PE1} in its second dimension. As each $PE1$ access all the elements in a column of buf_{in}, the buf_{in} should be fully partitioned in its first dimension. For weight buffer $\boldsymbol{buf_{wei}}$, since a $PE1$ access one element in $buf_{wei}[R_{WEI}][C_{WEI}]$ and the $PE1$ array access N_{PE1} elements in buf_{wei} simultaneously, buf_{wei} should be partitioned by parameter N_{PE1} in its first dimension. For output buffer $\boldsymbol{buf_{out}}$, as the last $PE2$ write its elements to the first dimension $buf_{out}[R_{OUT}][C_{OUT}]$ simultaneously, buf_{out} should be fully partitioned in its first dimension.

Accelerator running schedule. With the allocated three buffers and the computing core, the accelerator can continuously process the matrix multiplications (aka Linear modules). The running schedule of the accelerator is shown in Figure 13. With the adopted ping-pong buffer, the streaming of buf_{in}, buf_{wei}, and buf_{out} and the computing core work in parallel. In each iteration of the accelerator, a matrix multiplication is processed. The streaming performance for buf_{in}, buf_{wei}, and buf_{out} is determined by the bandwidth of the streaming interface. As the buffers are partitioned according to PE array's parallelism, there is no data fetch delay for the computing

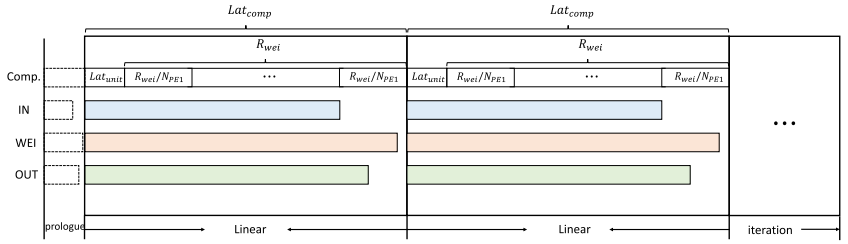

Fig. 13. The accelerator running schedule.

core. Benefiting from the pipeline design, the computing latency (Lat_{comp}) in an iteration is determined by three aspects, pipeline depth $Pipe_{depth}$ (Lat_{unit}), the number of executions for an output column (R_{WEI}/N_{PE1}), and the number of output columns (C_{WEI}). The computing latency for an accelerator iteration is described by Equation (4). At the run-time, the computing latency and streaming latency may vary from iteration to iteration as the matrices have different sizes.

$$\begin{aligned} Lat_{unit} &= Pipe_{depth} = \log_2 N_{PE1} + 2 \\ Lat_{comp} &= C_{WEI} * ceil\left(\frac{R_{WEI}}{N_{PE1}}\right) + Lat_{unit} \end{aligned} \tag{4}$$

7.2 Accelerator Analytical Model

As the parameters of attention mechanism vary in different applications while the DSP and LUT resources are flexible among FPGAs, an analytical model is developed to determine the design parameters of the accelerator and to analyze the system performance. The analytical model gives an estimation of the accelerator resource utilization and performance. The parameters in Figure 8 are used to illustrate the analytical model. The number of $PE1$ (N_{PE1}) is determined by DSP based $PE1$ (N^{dsp}_{PE1}) and LUT based $PE1$ (N^{lut}_{PE1}). In this model, the parallelism inside $PE1$ is set to the buf_{in}'s first dimension size N ($R_{IN} = N$). First, the N^{dsp}_{PE1} can be determined as the number of DSPs in an FPGA is fixed. After building the $PE1$ array in DSP, the number of $PE2$ can be determined ($PE2$ hierarchies=$\log_2 N^{dsp}_{PE1} + 1$). Next, if the FPGA LUT resource is still available, more LUT can be used in building $PE1$ and $PE2$ as INT8 operation is also efficient with FPGA LUT resources. The accelerator computing core's key parameters are summarized in Equation (5).

$$\begin{aligned} N_{PE1} &= N^{dsp}_{PE1} + N^{lut}_{PE1} \\ N^{dsp}_{PE1} &= floor(2 * N_{dsp}/N) \end{aligned} \tag{5}$$

The latency of the computing core is determined by the depth of the computing core which is $\log_2 N_{PE1} + 2$. The pipeline design and efficient data flow in the computing core achieve the minimum pipeline interval (II=1) to maximize the throughput. The run-time latency Lat^*_{comp} to process a matrix at the run-time is determined as $Lat^*_{comp} = C^*_{WEI} * ceil(\frac{R^*_{WEI}}{N_{PE1}}) + Lat_{unit}$, in which, C^* and R^* is the real size of the loaded weight at the run time.

As a result, the accelerator performance in processing a matrix can be modeled in Equation (6). The bottleneck is determined by the worst performance among the computing core and the buffer transmission. In Equation (6), Lat^{in}_{trans} can be determined by $Lat^{in}_{trans} = R^*_{IN} * C^*_{IN} * 8/B_{in}$, in which, B_{in} represents the streaming bandwidth assigned to buf_{in}. The Lat^{wei}_{trans} and Lat^{out}_{trans} can be acquired similarly. Then, the system performance Lat_{sys} is determined by Lat_{comp} and Lat_{trans}.

$$\begin{aligned} Lat_{trans} &= max\left(Lat^{in}_{trans}, Lat^{wei}_{trans}, Lat^{out}_{trans}\right) \\ Lat_{sys} &= max(Lat_{comp}, Lat_{trans}) \end{aligned} \tag{6}$$

Table 1. Transformer Parameters

Model	N_{enc}	N_{dec}	$head$	d_k	d_{model}
Transformer	6	6	8	64	512

While the proposed computing core achieves ultimate parallelism and pipeline performance, an adequate streaming bandwidth between off-chip and on-chip is also important. To avoid the system stall caused by data transmission, the minimum bandwidth requirement of each interface is also analyzed. After building the computing core and corresponding buffers, the required streaming bandwidth can be determined as $B_{in}^{min} \geq R_{IN} * C_{IN} * 8/Lat_{comp}$, $B_{wei}^{min} \geq R_{WEI} * C_{WEI} * 8/Lat_{comp}$, and $B_{out}^{min} \geq R_{OUT} * C_{OUT} * 8/Lat_{comp}$ for buf_{in}, buf_{wei}, and buf_{out}, respectively.

8 EXPERIMENTS

In this section, we first evaluate the compression performance for Transformer on machine translation tasks. Then, we evaluate the accelerator performance on Xilinx ZCU102 FPGA.

8.1 Compression Algorithm Evaluation

Compressed model performance. We use two datasets from language translation, i.e., the Multi30K [28] (a subset of WMT2014) and the IWSLT'17 [29], to evaluate the accuracy impact from our memory footprint aware compression. The key parameters of Transformer are listed in Table 1, which are consistent with the settings in [1, 6, 7]. The Transformer consists of 6 encoders and 6 decoders. The self-attention consists of 8 heads with $d_k = 64$ and $d_{model=512}$. The compression, training, and quantization processes are performed via Pytorch.

The evaluation is performed targeting on dataset Multi30K in English-German (En-De) and German-English (De-En) translation first. In Table 2, we report the achieved maximum compression ratio in the state-of-the-art works and our design that does not affect the accuracy. The Transformer is trained to get the baseline BLEU score first (29 for En-De and 25.8 for De-En). In En-De task, HAT [7] achieves 73% compression and TransformerZip [6] achieves 50% compression. Our method achieve 80% compression without accuracy loss, which is 7% and 30% higher than the state-of-the-art works respectively. The model compressed by our method can be directly quantized to INT8 without accuracy loss, achieving a final compression ratio of 95% (Ratio*). In De-En task, we also achieve 80% compression. After applying post-training quantization, we achieve a final compression ratio of 95%. As a result, the proposed compression method can reduce the number of computing operations by **80%** and the memory footprint by **95%** in Transformer for both tasks.

We further verify our method in dataset IWSLT'17 for English-German (En-De) and German-English (De-En) tasks. The baseline for the two tasks is trained as 13 for En-De and 15.5 for De-En. We achieve 70% and 75% compression for En-De and De-En tasks, respectively. The post-quantization does not affect the model accuracy and we finally achieve 92.5% and 93.75% compression ratio in the two tasks. As a result, the proposed compression method can reduce the number of computations by **70%/75%** and the memory footprint by **92.5%/93.75%** in Transformer for the two tasks.

Two-stage compression discussion. The two-stage compression process of En-De and De-En tasks for Multi30K is shown in Figure 14. For both tasks, the $incr_{coarse}$ is set to 10% in stage 1 and $incr_{fine}$ = 5% in stage 2. The accuracy is slightly improved at the early stage of stage 1 until reaching a 50% compression ratio for both tasks. However, a significant accuracy drop is observed when increasing the compression ratio from 50% to 60% in stage 1, which is shown by the red line. To prevent the sharp drop in the accuracy, the compression process exits stage 1 at a 50% compression ratio and enters stage 2. It is observed that the accuracy slightly drops when the compression ratio increases from 50% to 80% after entering stage 2 in both tasks. Further increasing the compression

Table 2. Performance of Compression and Quantization

			# Param (MB)	Compression Ratio (%)	BLEU	Quantized BLEU	Compression Ratio* (%)
Multi30K	En-De	Transformer	176	-	28.9	-	-
		HAT [7]	48	73% (-7%)	28.4	-	-
		TransformerZip [6]	86	50% (-30%)	26.4	-	-
		Ours	**8.8**	**80%**	**29**	**29**	**95%**
	De-En	Transformer	176	-	26	-	-
		Ours	**8.8**	**80%**	**25.8**	**25.8**	**95%**
IWSLT'17	En-De	Transformer	176	-	13	-	-
		Ours	**13.2**	**70%**	**13**	**13**	**92.5%**
	De-En	Transformer	176	-	15.5	-	-
		Ours	**11**	**75%**	**15.5**	**15.5**	**93.75%**

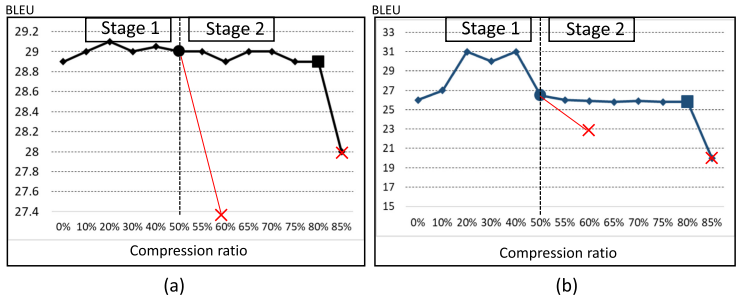

Fig. 14. Accuracy degradation during compression. (a) Multi30K En-De task. (b) Multi30K De-En task.

ratio incur a sharp drop in the accuracy again. The compression stops with a compression ratio of 80%. Therefore, the overall compression ratio is 80% and the $ratio_{coarse}$ is 50%.

Off-line compression overhead discussion. The off-line compression overhead consists of three parts, the weight analysis, the coarse pruning in stage 1, and the fine-tune pruning in stage 2. The weight analysis per Transformer encoder or decoder is about 22 GPU hours on 2080Ti GPU. The compression time for a given compression ratio is about 80 GPU hours on 2080Ti GPU in stage 1 or stage 2. When linearly exploring the compression ratio that we can achieve, the off-line compression time is about 1000 GPU hours for Multi30K tasks, 900 GPU hours for IWSLT'17 En-De task, and 1000 GPU hours for IWSLT'17 De-En task. However, the weight analysis of Transformer encoders and decoders can be executed in parallel; the exploration in both stages 1 and 2 can be executed in parallel. This results in less than a total of 200 hours wall-clock time.

Compression performance exploration. The proposed compression techniques show superior performance when compared with the existing designs. We explore the impact of weight dimension variation on the task IWSLT'17 En-De, which is the most challenging task (i.e. the lowest baseline BLEU and lowest achieved compression ratio) among the four tasks we evaluated. Therefore, we explore the compression performance with different weight variations for task IWSLT'17 En-De. The exploration results are shown in Figure 15(a). In this figure, the x-axis represents the overall compression ratio which combines stage 1 and stage 2. The $incr_{fine}$ is set to 2% per step. The y-axis shows the evaluated BLEU score at different compression ratios (the baseline BLEU score is 13). We explore 'Variation Large', which exits stage 1 at 20% compression ratio, leaves an 80% space for stage 2 fine-tuning; 'Variation Small', which exits stage 1 at 30% compression ratio,

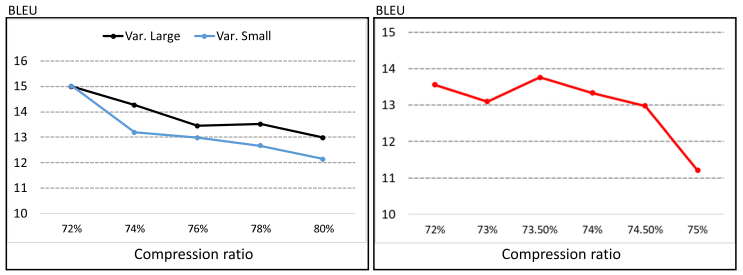

Fig. 15. (a) The compression performance exploration with different weight dimension control for task IWSLT'17 En-De. (b) The compression performance with more fine-grained $incr_{fine}$ for task IWSLT'17 En-De.

leaves a 70% space for stage 2 fine-tuning. We can observe that both 'Variation Large' and 'Variation Small' achieve higher compression ratio without accuracy drop while the extreme variation control achieves 70% which is reported in Table. 2. The 'Variation Large' shows better performance when compared with 'Variation Small' and it can reach an 80% overall compression ratio without accuracy drop. 'Variation Small' reaches a 76% overall compression without accuracy drop. Therefore, we can achieve better compression performance at the cost of slightly larger weight dimension variation.

We also explore the trend of accuracy drop by using more compression ratios with a smaller step size. Instead of increasing by 5% ($incr_{fine}$) in stage 2, we explore the accuracy drop by increasing the compression ratio by the step size of 2%, 1%, and 0.5%. The accuracy drop trend is shown in Figure 15(b). In this Figure, we explore the task IWSLT'17 En-De which originally achieves a 70% compression ratio with a base BLEU score of 13. In this exploration, the compression still exits stage 1 at 50% compression ratio and enters stage 2 with $incr_{fine}$ = 2%. We further reduce the $incr_{fine}$ to 0.5% when approaching the 75% overall compression ratio. The proposed compression technique can still maintain the BLEU over 13 until reaching 74.5% compression ratio. A drastic accuracy drop is observed at 75% compression ratio. Therefore, we can achieve a greater compression ratio even without accuracy drop by adopting more fine-grained $incr_{fine}$ in compression stage 2.

Compression impact on buffer size. As a result of the compression, the weight size upper bound for Q^w_{layer}, K^w_{layer}, V^w_{layer}, O^w_{layer}, $FFN1^w$, and $FFN2^w$ within Transformer is 50% of its original size. Therefore, the weight buffer buf_{wei} size is early estimated as buf_{wei}[512][256]. The input buffer buf_{in} size is early estimated as buf_{in}[100][512]. The output buffer buf_{out} size is early determined as buf_{out}[100][256]. In the buffer dimension, 100 is the value of N, which is the length of input sentence. The real on-chip buffer allocation will be co-determined with computing core parameters.

8.2 Hardware Accelerator Evaluation

An accelerator in INT8 towards Transformer compression $ratio_{coarse}$ = 50% is built with Xilinx HLS and synthesized in Vivado (v2019.1). The accelerator performance is tested with Vivado SDK.

Computing core implementation. On ZCU102 FPGA, we build the computing core with 74 $PE1$ (50 N^{dsp}_{PE1} and 24 N^{lut}_{PE1}) and 73 $PE2$. The parallelism of each PE is 100 which is the maximum sentence length of the dataset. In this design, the ideal "adder-tree" is fine-tuned in selected hierarchies to fit 74 $PE1$ outputs into $PE2$ array's data flow, resulting in a 10 stage pipeline with pipeline interval II=1. The resource utilization breakdown for $PE1$ and $PE2$ is shown in Table 3. 50 DSPs are used in building a $PE1^{dsp}$ and 5000 LUTs are used in building $PE1^{lut}$. 1500 LUTs are used in building a $PE2$. The developed computing core fully utilizes the FPGA DSP and LUT resources and can

Table 3. Processing Element Resource Breakdown

	Number	Parallelism	DSP	LUT	FF	BRAM
$PE1^{dsp}$	50	100	50	-	1608	0
$PE1^{lut}$	24	100	-	5000	1608	0
$PE2$	73	100	-	1500	2400	0
$Computing\ core$	1	7400	2500	-	-	0

Table 4. Accelerator Buffer Allocation

	buf_{in}	buf_{wei}	buf_{out}
Rows	100	518	100
Columns	518	(256+2048)	256
Stream. Bus	$\langle 1, 2, 1 \rangle$		

Table 5. Accelerator Performance on ZCU102

Resource					Bus. $\langle\rangle$	Freq. (MHz)	Comp. Through.	End2End Through.	Lat_{comp} (ms)	Lat_{sys} (ms)
	DSP	LUT	BRAM	FF						
Avail	2520	274080	912	548160						
design1	99.3%	91.8%	28%	29.3%	$\langle 1, 2, 1 \rangle$	150	1.87Tops	794Gops	0.014	0.033
design2	99.3%	91.9%	77%	26.4%	$\langle 1, 2, 1 \rangle$	125	1.7Tops	845Gops	0.015	0.031
design2*	99.3%	91.9%	77%	26.4%	$\langle 2, 0, 2 \rangle$	125	1.7Tops	1.4Tops	0.015	0.019

conduct 7400 multiply-accumulate operations per cycle. The theoretical throughput performance of the computing core is 2.22 Tops at 150 MHz.

Accelerator buffer allocation. The buffer allocation is conducted by considering both the computing core and the compressed Transformer size. The parameters of the allocated buffer buf_{in}, buf_{wei}, and buf_{out} in the accelerator are shown in Table 4. The buffers are implemented by BRAMs and LUTRAMs. Since 74 $PE1$ are built, the first dimension of buf_{wei} and the second dimension of buf_{in} is rounded to 518 for ease of PE array access. The second dimension of buf_{wei} is selected as 256 since the $ratio_{coarse}$ is 50%. After building the accelerator, the rest BRAMs on ZCU102 FPGA is utilized to expand the buf_{wei} in its second dimension. The expansion size of buf_{wei} is 2048, which can store 8.4Mb weights on the chip. At the interface between ZCU102 on-chip memory and the off-chip DDR, four streaming buses (128bit each) are available. Therefore, the streaming interface is determined according to buffer size: one streaming bus for buf_{in}; two streaming buses for buf_{wei}; one streaming bus for buf_{out}. The streaming bus allocation is represented by $\langle 1, 2, 1 \rangle$ in the table.

Accelerator performance analysis. Three versions of the accelerator are implemented to show the superiority of our design. Design1 is built with no buf_{wei} expansion. Design2 is built with buf_{wei} expansion. Design2* is built with buf_{wei} expansion and streaming interface tuning. In the three designs, we show the computing core's peak throughput (Computing Throughput), the accelerator system throughput considering data transfer (End2End Throughput), the computing latency (Lat_{comp}) in processing a unit matrix multiplication (buf_{in}[100][518], buf_{in}[518][256], and buf_{out}[100][256]), and the end-to-end latency (Lat_{sys}) in processing the unit matrix multiplication.

Design1 explores the highest throughput that the computing core can achieve. With the minimum active buffer, it achieves the highest working frequency at 150 MHz working in pure streaming in, computing, and streaming out mode (stream-in processing). It fully utilizes the DSP and LUT resource to build computing core, and 28% of BRAM resource to build buffers. The streaming

Table 6. End-to-end Accelerator Performance on Real Transformer

Standard Transformer (Our Work)								
Sparsity	Gop	DSP	LUT	BRAM	FF	Freq. (MHz)	Latency	Area Efficiency
80%	1.29	2500	251725	699	142723	125	6.8 *ms*	0.076 (**6.3x**)
75%	1.65						7.2 *ms*	0.092 (**7.6x**)
70%	2.04						8.4 *ms*	0.097 (**8.1x**)
Shallow Transformer (Existing design [8])								
-	0.205	5647	268933	-	304012	-	2.94 *ms*	0.012
Standard Transformer (Existing design [30])								
-	-	129	471563	498	217859	200	8.9 *ms*	-

interface is assigned as $\langle 1, 2, 1\rangle$. At the run-time, design1's computing throughput is measured at 1.87 Tops. This is only slightly lower than its theoretical value due to the initialization of FPGA IP cores. The end-to-end processing throughput is 794 Gops. The latency Lat_{comp} is 0.014 *ms* and the Lat_{sys} is 0.033 *ms*. The significantly lowered end-to-end throughput is due to the limited streaming interface bandwidth, which causes significant computing core stall. If the bandwidth is sufficient to support the data consumption of the computing core, the design1 can work under 150 MHz with a throughput of 1.87 Tops.

Design2 explores the generic acceleration solution which partially stores the model on the chip. It supports "stream-in processing" and "in-situ processing" modes, in which, "in-situ processing" consumes the on-chip weights. Compared with design1, design2 utilizes the spare BRAM resource to expand the buf_{wei} capacity and 77% BRAMs are utilized. Therefore, 8.5Mb spare buffer spaces are generated to keep selected weights on the chip. With more active buffer, design2 works under 125 MHz. By partially storing a model on the chip, only the unbuffered weights need to be streamed in. The streaming interface is assigned as $\langle 1, 2, 1\rangle$ to support both modes. We report "in-situ processing" performance in the table. The computing throughput is 1.7 Tops. The end-to-end throughput is 845 Gops. The latency Lat_{comp} is 0.015 *ms*. The Lat_{sys} is 0.031 *ms*. In this mode, the end-to-end system performance is moderately improved as buf_{wei} still occupies the streaming interface, leaving limited streaming bandwidth for other buffers. When working in "stream-in processing", design2's end-to-end throughput is slightly lower than design1's due to the lower working frequency. Therefore, in design2, the overall performance can benefit from less transmission with the on-chip weights.

Design2* explores the accelerator solution for the small model that can be fully stored on the chip, which works in pure "in-situ processing". Design2* tunes the streaming interface to $\langle 2, 0, 2\rangle$, more streaming bandwidth is assigned to buf_{in} and buf_{out} as no weight transfer is needed. After eliminating weight transfer, the end-to-end processing throughput is greatly improved to 1.4 Tops. The small gap between end-to-end throughput and computing throughput is due to the transmission initialization between DDR and FPGA.

To mitigate the gap between the computing throughput and end-to-end throughput, based on the analytical model for the streaming interface, the minimum bandwidth for buf_{in} needs to be 210 bit/cycle, the minimum bandwidth for buf_{wei} needs to be 539 bit/cycle, the minimum bandwidth for buf_{out} needs to be 104 bit/cycle. Such bandwidth settings can fulfill the throughput of our computing core.

Accelerator performance on real-life Transformer. We accelerate the compressed Transformer models discussed in Table 2, in which, 80%, 75%, and 70% compression ratios are achieved. This reduces the number of multiplyaccumulate operations to 1.29, 1.65, and 2.04 Giga Operations (Gop). Due to the size of the compressed models, design2 is adopted to accelerate the three

Table 7. Cross-platform Comparison

	FPGA ZCU102	GPU 2070S	GPU Jetson TX2	CPU 9700K	ARM A57
Latency (*ms*)	6.8	0.9	50	7	1350
Gop/s	190.1	1438	25.8	184.9	0.95
Power (W)	22.5	215	6.5	185	4.5
Energy Effiency (Gops/watt)	**8.44**	6.68	3.98	1.18	0.21

compressed models. The breakdown of accelerator resource utilization is listed in Table 6. In processing a full encoder layer to decoder layer inference, we achieve the latency of 6.8*ms*, 7.2*ms*, and 8.4*ms* for the three compression degrees. In this table, we also list the performance of the state-of-the-art Transformer accelerator [8] and [30]. [8] accelerated a shallow Transformer with 5.76MB weights and 0.205 Gop. Their accelerator is built on VCU118 FPGA in fix-point data via Vivado HLS, in which, 5647 DSPs are utilized. It achieves 2.94*ms* in accelerating the shallow Transformer. We compare the FPGA area efficiency via $Gop/Latency/N_{dsp}$ for fair comparison. As shown in the table, our accelerator achieves 6.3x, 7.6x, and 8.1x better efficiency at different compression ratios, which is significantly higher than the state-of-the-art accelerator [8]. Design [30] is built on XCVU13P FPGA in INT8 via hardware description language (HDL). The performance is measured by simulation in Vivado. We report its performance in accelerating a full Transformer, which achieves 8.9*ms* in latency. Its latency is significantly higher than our design. Furthermore, the design [30] can hardly utilize its device resource, causing severe resource under-utilization (27.% LUT utilization, 36.5% BRAM utilization, and 2% DSP utilization).

Performance trade-off with loosen weight dimension control. In the previous section, compression performance exploration, we explore the performance trade-off in weight dimension variation and compression ratio. In this section, we also explore the performance trade-off between the accelerator throughput and the weight dimension variation when we achieve an 80% compression ratio with 'Variation Large'. We implement the accelerator similar to design1, in which, the buffer size is re-designed to (buf_{in}[100][518], buf_{in}[518][410], and buf_{out}[100][410]). As a result of the larger buffer, in processing the same matrix which is evaluated in Table 5, we achieve the working frequency, computing throughput, end-to-end throughput, Lat_{comp}, and Lat_{sys} in 140 MHz, 1.77 Tops, 757 Gops, 0.015 *ms*, 0.035 *ms*, respectively. We can see that the accelerator performance is only slightly influenced by the larger weight dimension variation. This is because the weight buffer is expanded in the second dimension, which does not need to be partitioned to support the computing core. Thus, the timing performance of the placement and routing during the synthesis is not significantly influenced by the larger weight buffer. However, such strategy leads to severe buffer under-utilization in processing different matrices in the Transformer.

Cross-platform performance comparison. We compare the energy efficiency among FPGA ZCU102, GPU 2070 Super, GPU Jetson TX2, CPU i7-9700K, and ARM A57 in processing the Transformer with an 80% compression ratio in Table 7. On all platforms, the end-to-end throughput performance (Gop/s) is influenced by the extremely low compression ratio caused small matrix operations. Desktop GPU wins in latency due to its extreme computing power and the large batch size. Its latency is measured by batch processing time divided by the batch size. Our design outperforms mobile GPU and ARM in latency as we achieve extreme high computing parallelism for the pruned Transformer. The computing inefficiency caused by the small and different matrix is resolved by the proposed computing pattern and accelerator architecture. The desktop CPU has longer but comparable latency with our design. This is due to the high operating frequency of the CPU and its high-speed main memory. The desktop CPU 9700K with a 12MB cache works under

5.0GHz and is supported by a 3200MHz DDR memory. The working power of the CPU itself is 184.9W on average, while our design works under 22.5W on average. Among all platforms, our design achieves significantly higher energy efficiency (Gops/watt) than other platforms. We achieve 1.26x, 2.1x, 7x, and 40x improvement when compared with GPU 2070 Super, GPU Jetson TX2, CPU i7-9700K, and ARM A57 respectively.

9 CONCLUSION

In this work, we propose an algorithm-hardware co-design for the attention mechanism on FPGA. The proposed compression method effectively compresses the attention mechanism by 95% and minimizes the dimension size variation among weights. The proposed unified computing pattern eliminates the inefficiency caused by the sparse attention mechanism. As a result of the co-design, the developed hardware accelerator fully utilizes the FPGA's DSP and LUT resources, achieving a run-time computing throughput of 1.87 Tops.

REFERENCES

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *arXiv preprint arXiv:1706.03762* (2017).

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

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

[4] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. 2020. End-to-end object detection with transformers. In *European Conference on Computer Vision*. Springer, 213–229.

[5] Zhanghao Wu, Zhijian Liu, Ji Lin, Yujun Lin, and Song Han. 2020. Lite transformer with long-short range attention. *arXiv preprint arXiv:2004.11886* (2020).

[6] Robin Cheong and Robel Daniel. 2019. *transformers. zip: Compressing Transformers with Pruning and Quantization.* Technical Report. Technical report, Stanford University, Stanford, California.

[7] Hanrui Wang, Zhanghao Wu, Zhijian Liu, Han Cai, Ligeng Zhu, Chuang Gan, and Song Han. 2020. Hat: Hardware-aware transformers for efficient natural language processing. *arXiv preprint arXiv:2005.14187* (2020).

[8] Bingbing Li, Santosh Pandey, Haowen Fang, Yanjun Lyv, Ji Li, Jieyang Chen, Mimi Xie, Lipeng Wan, Hang Liu, and Caiwen Ding. 2020. FTRANS: Energy-efficient acceleration of transformers using FPGA. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*. 175–180.

[9] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2016. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710* (2016).

[10] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. 2017. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 13, 3 (2017), 1–18.

[11] Yawen Wu, Zhepeng Wang, Zhenge Jia, Yiyu Shi, and Jingtong Hu. 2020. Intermittent inference with nonuniformly compressed multi-exit neural network for energy harvesting powered devices. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[12] Xilinx. 2019. Supercharge Your AI and database applications with xilinx's HBM-enabled ultrascale+ devices featuring samsung HBM2. In *Xilinx white paper, WP508 (v1.1.2)*.

[13] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*. 161–170.

[14] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 367–379.

[15] Yongming Shen, Michael Ferdman, and Peter Milder. 2017. Maximizing CNN accelerator efficiency through resource partitioning. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 535–547.

[16] Weiwen Jiang, Edwin H.-M. Sha, Xinyi Zhang, Lei Yang, Qingfeng Zhuge, Yiyu Shi, and Jingtong Hu. 2019. Achieving super-linear speedup across multi-fpga for real-time dnn inference. *ACM Transactions on Embedded Computing Systems (TECS)* 18, 5s (2019), 1–23.

[17] Weiwen Jiang, Xinyi Zhang, Edwin H.-M. Sha, Lei Yang, Qingfeng Zhuge, Yiyu Shi, and Jingtong Hu. 2019. Accuracy vs. efficiency: Achieving both through fpga-implementation aware neural architecture search. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–6.

[18] Weiwen Jiang, Lei Yang, Edwin Hsing-Mean Sha, Qingfeng Zhuge, Shouzhen Gu, Sakyasingha Dasgupta, Yiyu Shi, and Jingtong Hu. 2020. Hardware/software co-exploration of neural architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 12 (2020), 4805–4815.

[19] Xinyi Zhang, Weiwen Jiang, and Jingtong Hu. 2020. Achieving full parallelism in LSTM via a unified accelerator design. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE, 469–477.

[20] Md Jubaer Hossain Pantho, Pankaj Bhowmik, and Christophe Bobda. 2021. Towards an Efficient CNN inference architecture enabling in-sensor processing. *Sensors* 21, 6 (2021), 1955.

[21] Jonathan Frankle and Michael Carbin. 2018. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635* (2018).

[22] Haoran You, Chaojian Li, Pengfei Xu, Yonggan Fu, Yue Wang, Xiaohan Chen, Richard G. Baraniuk, Zhangyang Wang, and Yingyan Lin. 2019. Drawing early-bird tickets: Towards more efficient training of deep networks. *arXiv preprint arXiv:1909.11957* (2019).

[23] Ari S. Morcos, Haonan Yu, Michela Paganini, and Yuandong Tian. 2019. One ticket to win them all: Generalizing lottery ticket initializations across datasets and optimizers. *arXiv preprint arXiv:1906.02773* (2019).

[24] Eran Malach, Gilad Yehudai, Shai Shalev-Schwartz, and Ohad Shamir. 2020. Proving the lottery ticket hypothesis: Pruning is all you need. In *International Conference on Machine Learning*. PMLR, 6682–6691.

[25] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450* (2016).

[26] [n.d.]. FBGEMM. Website. https://github.com/pytorch/FBGEMM.

[27] Xilinx. 2017. Deep Learning with INT8 optimization on xilinx devices. In *Xilinx WP486*.

[28] [n.d.]. Multi30K. Website. https://github.com/multi30k/dataset.

[29] [n.d.]. IWSLT. Website. http://workshop2017.iwslt.org/.

[30] Siyuan Lu, Meiqi Wang, Shuang Liang, Jun Lin, and Zhongfeng Wang. 2020. Hardware accelerator for multi-head attention and position-wise feed-forward in the transformer. *arXiv preprint arXiv:2009.08605* (2020).

Received April 2021; revised June 2021; accepted July 2021