Accelerating Transformer-based Deep Learning Models on FPGAs using Column Balanced Block Pruning

*Hongwu Peng[1], Shaoyi Huang[1], Tong Geng[2], Ang Li[2], Weiwen Jiang[3],
Hang Liu[4], Shusen Wang[4], and Caiwen Ding[1]
[1]University of Connecticut, Storrs, CT, USA
[2]Pacific Northwest National Laboratory, Richland, WA, USA
[3]University of Notre Dame, Notre Dame, IN, USA
[3]Stevens Institute of Technology, Hoboken, NJ, USA
*hongwu.peng@uconn.edu

2021 22nd International Symposium on Quality Electronic Design (ISQED) | 978-1-7281-7641-3/20/$31.00 ©2021 IEEE | DOI: 10.1109/ISQED51717.2021.9424344

***Abstract*—Although Transformer-based language representations achieve state-of-the-art accuracy on various natural language processing (NLP) tasks, the large model size has been challenging the resource constrained computing platforms. Weight pruning, as a popular and effective technique in reducing the number of weight parameters and accelerating the Transformer, has been investigated on GPUs. However, the Transformer acceleration using weight pruning on field-programmable gate array (FPGAs) remains unexplored. This paper investigates the column balanced block-wise pruning on Transformer and designs an FPGA acceleration engine to customize the balanced block-wise matrix multiplication. We implement the Transformer model with proper hardware scheduling, and the experiments show that the Transformer inference on FPGA achieves 10.35 ms latency with the batch size of 32, which is 10.96 × speed up comparing to CPU platform and 2.08 × speed up comparing to GPU platform.**

***Index Terms*—Transformer, deep learning, pruning, acceleration, FPGA**

I. Introduction

In the field of natural language processing (NLP), the recurrent neural network (RNN) [1], and long short term memory (LTSM) model [2] have been well deployed to different tasks in the past. However, the RNN and LTSM models' training and inference are intrinsically sequential computation tasks, making it difficult to accelerate on today's hardware, e.g., GPUs and field-programmable gate array (FPGAs) [3], [4]. In 2017, the Transformer architecture, which relies on self-attention mechanisms [5] was proposed. The Transformer model enables a high level of computation parallelism on both training and computation. It outperforms RNN and LSTM in major NLP tasks, such as language inference, question answering, and sentiment analysis.

On the other hand, weight pruning methodology such as irregular pruning [6], structured pruning [7], pattern pruning [8] have been widely used for deep neural networks in the computer vision field. It has also been used to accelerate Transformer-based DNNs due to the enormous parameters or model size of the Transformer. With weight pruning, the size of the Transformer can be significantly reduced without much prediction accuracy degradation [9]. Therefore, we can accommodate the compressed and high accurate Transformer model into FPGAs. In recent years, because of its extremely low-latency, high energy efficiency, and flexible reprogrammability for easy prototyping, FPGAs have received much attention as an alternative accelerating solution to GPU for ML and real-time data analytics applications [10]. However, the current state-of-the-art Transformer acceleration focus on the GPUs [11]–[13]. There lacks of comprehensive investigation on Transformer acceleration using hardware-aware weight pruning FPGA.

In this paper, we focus on the acceleration of the Transformer model on FPGAs with the balanced block pruning technique. We further propose hardware design and the resource scheduling to achieve high parallelism and high throughput on FPGA device. Our contributions are summarized as follows:

- A column balanced block pruning technique and its storage format are developed. The indices pointer matrix has much lower memory storage overhead than that of other sparse matrix formats.
- A specialized process element (PE) is introduced for the sparse matrix multiplication accelerator, and multiple PEs can be used to increase the accelerator throughput.
- The hardware resource scheduling for the encoder/decoder accelerator on FPGA is discussed, and we achieve a high overall hardware throughput.

We implement the proposed techniques on different hardware platforms (Intel i5-5257U (2.7 GHZ) CPU, Nvidia Jetson TX2 GPU, and Xilinx Alveo U200 FPGA) for further comparison of latency and throughput. Experimental results show that the FPGA hardware design enables a 10.96 × speed up on the FPGA platform comparing to the CPU platform and 2.08 × speed up compared to the GPU platform.

The organization of the work is as follows. Section II gives the basic Transformer model and DNN model compression knowledge, and the structure of the encoder is given. Section III proposes a column balanced block-wise pruning structure and its pruning algorithm. Section IV gives the hardware design for the sparse matrix multiplication accelerator and the overall Transformer structure. Section V gives the Transformer model's training, the hardware scheduling result of the encoder/decoder layer, and the inference speed for different hardware platforms. Section VI gives the overall conclusion for the hardware design and experiments.

II. Background and Related Work

This section focuses on the background of the Transformer model and the pruning methods for the hardware acceleration.

A. Transformer Model

The Transformer model was firstly proposed in 2017 [5] for NLP tasks. Different Transformer-based models have been proposed, such as BERT [14], RoBERTa [15], which utilizes more layers and heads to achieve better performance on major NLP tasks. To reduce the model size, DistilBERT [16] was invented based on knowledge distilling [16], and achieves 40% model size reduction from BERT without much accuracy degradation. Moreover, the Transformer model has been utilized in the computer vision field, such as the Transformer for image recognition [17] and the Transformer for object detection [18].

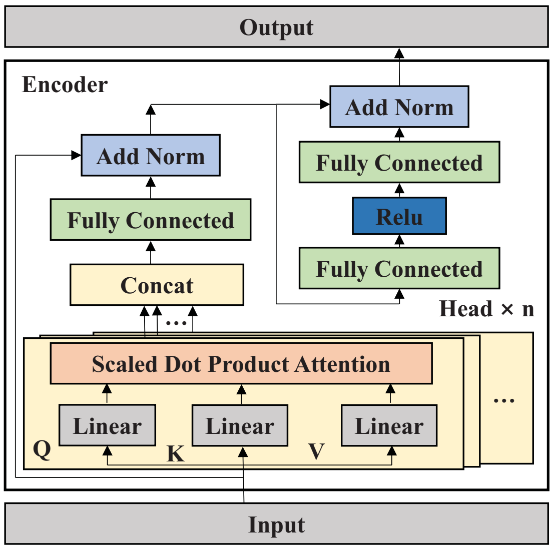


Fig. 1: Encoder structure.

Our research will focus on the Transformer model inference acceleration on FPGAs. The encoder structure of our Transformer is shown in Fig. 1. Our Transformer has 2 layers of encoder and 4 heads. The decoder is simply a linear layer on top of the encoder stack. The scaled dot product attention is used for the self-attention mechanism, and the equation is shown below:

$$Attention(\mathbf{Q},\mathbf{K},\mathbf{V}) = \text{Softmax}(\text{Mask}(\frac{\mathbf{Q}\times\mathbf{K}^T}{\sqrt{d_k}}))\times\mathbf{V} \quad (1)$$

B. DNN Model Compression and Sparsity Pattern

Different model compression techniques have been studied to reduce the parameter size and computation burden of the deep neural networks (DNNs). The main challenge of the state-of-the-art model compression techniques is maintaining the model accuracy while improving the model execution efficiency on hardware devices. Irregular pruning methods [11] enables high prune ratio without much performance degradation. However, it is not easy to be accelerated on GPUs and FPGAs due to its random memory access pattern and extra indices overhead. For instance, when storing an irregular sparse matrix using Coordinate (COO) format, we store the subsequent nonzero and related coordinates in memory. Three vectors are needed: row, col, data, where data[i] is value at (row[i], col[i]) position. The regular pruning features its relatively higher regularity on non-zero elements, and the regularity can help speed up the sparse matrix multiplication on FPGAs and GPUs. Regular pruning, such as bank balanced pruning [19]–[21], block-circulant matrix pruning [22], and block pruning [11], have been developed. However, the block pruning structure has only been implemented on GPUs in the past; the block pruning structure's acceleration on FPGAs remains unexplored.

III. Sparse Matrix Format and Pruning Algorithm

In this section, we will evaluate different sparse matrix formats. A row balanced block-wise sparsity is proposed, which strikes a better balance between index storage overhead and high accuracy. It also achieves high hardware parallelism among different pruning methodologies.

A. Comparison of Different Sparse Matrix Formats

Fig. 2 shows an example of four different types of pruning pattern for sparse matrix. The first one is the irregular pruning technique. The irregular pruning pattern puts a weight threshold for the specific pruning ratio and prunes out the elements below the weight threshold. The pruning is done in an element-wise pattern so that the irregular pruning pattern has the lowest accuracy drop as the pruning ratio increases. The irregular sparsity matrix indices can be stored in Compressed Sparse Row (CSR) format or Coordinate list (COO) format, both of which will occupy extra storage space. When the pruning ratio is low, the required memory space for irregular sparsity matrix storage might be higher than that of a dense matrix. Furthermore, the irregular sparsity matrix imposes a great challenge on the hardware design. The non-zero element of the irregular sparsity matrix is randomly distributed over the entire matrix, which results in an irregular memory access pattern and stalls the hardware parallelism. Thus, the speedup for the irregular sparsity matrix's operation can be negative. As a result, other types of regular pruning structures were proposed.

The bank balanced pruning has been widely used in different DNN applications. [19] discussed the hardware acceleration of bank balanced pruning sparse matrix operation on GPUs, and [20], [21] implement the bank balances structure

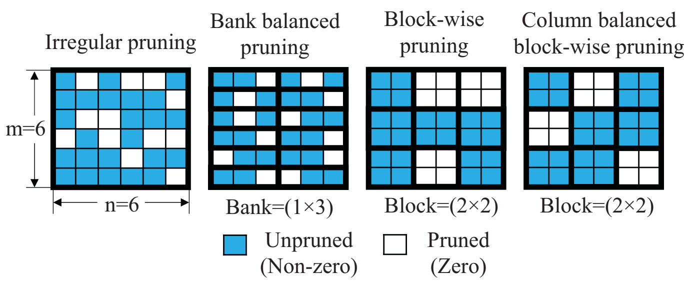


Fig. 2: Four types of pruning pattern with 0.33 pruning ratio: irregular pruning, bank balanced pruning, block-wise pruning, and column balanced block-wise pruning.

matrix operation on FPGAs. Both of the papers showed detailed hardware design and performance evaluation for bank balanced pruning. [20] proposed a Compressed Sparse Banks (CSB) for the sparse matrix storage. However, both the CSR and CSB formats require at least one index pointer for each non-zero element, which leads to extra memory occupation overhead. Banks balanced pruning may require more memory space at a low pruning ratio than the dense matrix because of the index pointer storage overhead.

The third pattern, block-wise pruning [22], calculates the block's L2 norm and prunes the block with a lower L2 norm. The block-wise pruning is similar to irregular pruning except that the weight importance is calculated for each block rather than individual elements. Block Compressed Sparse Row (BCSR) format can be used to store the weight matrix, which significantly reduces index pointer storage overhead since the whole block of elements share the same index. However, the blocks are randomly pruned over the entire matrix, making the hardware design for intra-block parallelism a great challenge.

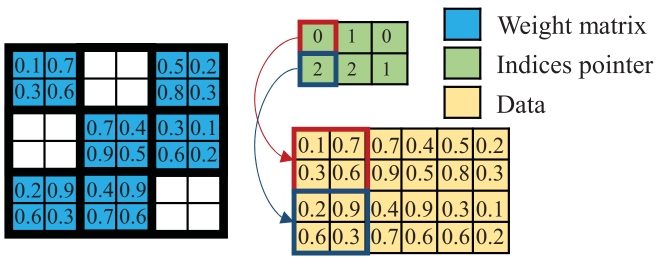


Fig. 3: An example of CSCB format for column balanced block-wise sparse matrix, where the block size is 2×2.

The column balanced block-wise pruning combines the key features of both bank balanced pruning and block-wise pruning. The column balanced block-wise pruning ranks the blocks' L2 norm by each column to get the pruning thresholds and prunes blocks for each column. A detailed algorithm for column balanced block-wise pruning is given in the next section. By combining the CSB format and BCSR format, a Compressed Sparse Column Block (CSCB) is formed. An example of the CSCB format is shown in Fig. 3. Only one index pointer for each block is needed, which leads to a much lower memory consumption than the previous sparse matrix pattern. Moreover, the column balanced block-wise pruning enables both inter-block and intra-block parallelism for efficient hardware design. The hardware design details are revealed in later sections.

B. Column Balanced Block-wise Pruning Algorithm

We propose a column balanced block-wise weight pruning algorithm, where column balanced is achieved by pruning the same number of block column elements in each column of weight matrices, which are weights of linear layers with the shape of 2-D for transformer model in our experiments.

Algorithm 1: Column balanced block-wise weight pruning algorithm.

```
1  Input: weight matrix $\boldsymbol{W}$, matrix width $n$, matrix height $m$,
   block row $r$, block column $c$, percentile $perc$
2  Output: pruned weight matrix $\boldsymbol{W}_p$
3  Set $\boldsymbol{W}_p$ = $\boldsymbol{W}$
4  Set column division $j$ = $n$ / $c$, row division $j'$ = $m$ / $r$
5  Divide $\boldsymbol{W}_p$ into $j$ matrices: $\boldsymbol{W}_1$,$\boldsymbol{W}_2$,...,$\boldsymbol{W}_j$
6  foreach $\boldsymbol{W}_i$ in $\boldsymbol{W}_1$,$\boldsymbol{W}_2$,...,$\boldsymbol{W}_j$ do
7      Divide $\boldsymbol{W}_i$ into $j'$ matrices
8      Calculate $l_2$ norm of each block
9      Setting the value in $perc$ of the lowest $l_2$ norm of blocks
         of $\boldsymbol{W}_i$ as zero
10 end
11 $\boldsymbol{W}_p$ = concatenate($\boldsymbol{W}_1$,$\boldsymbol{W}_2$,...,$\boldsymbol{W}_j$)
```

We denote weight matrix as $\boldsymbol{W}$ with width n and height m while row size and column size of pruning block are denoted by r and c respectively and $perc$ is the percentile of weights will be excluded within $\boldsymbol{W}$. Algorithm 1 illustrates column balanced block-wise weight pruning method. We (a) first divide a Transformer weight matrix into j sub-matrices where $j = n/c$. (b) For each sub-matrix $\boldsymbol{W}_i$, we divide it into j' sub-matrices where $j' = m/r$ and $j \times j'$ blocks with size of r by c are generated in this step. (c) We calculate l_2 norm of each block. (d) For each $\boldsymbol{W}_i$, we set the value of $perc$ of blocks with lower l_2 norm to zero. (e) Finally, we concatenate the resulting sub-matrices $\boldsymbol{W}_1$,$\boldsymbol{W}_2$,...,$\boldsymbol{W}_j$ horizontally to form the pruned matrix $\boldsymbol{W}_p$.

IV. Transformer Accelerator Design

Most of the Transformer model, even the one with some shallow encoder and decoder layers, is too large to fit into the FPGA on-chip block RAM (BRAM). Thus, the model compression technique is needed to compress the model size and fit it into the existing FPGA devices. The hardware design for compressed matrix operation is a great challenge, and it directly determines how much speedup we can achieve on hardware design. The embedding layer is normally a lookup table, and it contributes 30% of parameters, so it's loaded into external memory instead of on-chip BRAM.

In this section, we first come up with holistic hardware architecture for the Transformer hardware accelerator. Then we dive into details of the sparse matrix multiplication accelerator as well as the encoder framework. We aim to reduce the inference latency by add more hardware parallelism and enable the efficient pipeline on the hardware data flow.

A. Overall Hardware Architecture

As shown in Fig. 4, the hardware architecture of the Transformer is composed of the host PC which is in charge of generating and sending the tokenized sentence, the off-chip DDR memory and its' controller for the embedding layer, and the FPGA accelerator for encoders and decoder layer. As mentioned in the previous section, our transformer model is composed of 2 layers of encoders and 1 layer of the decoder. The decoder is simply a linear layer that takes the encoder's input and generates the final output. The hardware resources for the encoder can be reused for different layers. However, we only have 2 layers of the encoder, so hardware resources re-utilization is not necessary.

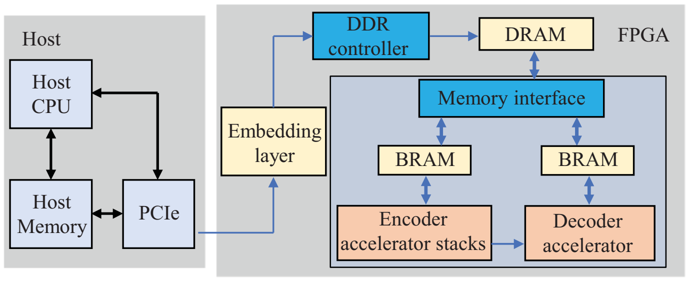


Fig. 4: Overall structure of the Transformer accelerator

The hardware data flow is as follows. Firstly, the host PC generates the tokenized sentence and sends it to FPGA through the PCIe interface. Then, the DDR controller will fetch the embedding from DRAM for each word of the input sentence. The word embedding sequence is then fed into FPGA on-chip resources for encoders and decoder inference.

B. Sparse Matrix Multiplication Accelerator Design

The sparse matrix multiplication accelerator design is shown in Fig. 5. The compressed sparse matrix is stored in CSCB format, and there are 2 matrices need for CSCB format; one is the data matrix, and one is the indices pointer matrix. Each of the indices pointer matrix element indicates the original location of the block in the data matrix. To exploit hardware parallelism, we design a processor element (PE) for each task, and multiple PEs can be used in parallel to increase the hardware bandwidth.

The PE design is as follows. For each PE shown in Fig. 5, it will have one copy of the single row of the dense matrix input as register buffer array. And for each operation, the PE will use the indices pointer to fetch two inputs: a single bank from dense matrix row and a whole data block from the compressed sparse matrix. After the data fetching procedure, a dense general matrix multiply (GEMM) can be performed within the PE. The GEMM accelerator design has already been exploited in the previous work [23], and we can follow the basic procedure to design a highly paralleled GEMM hardware accelerator. Lastly, the GEMM result will

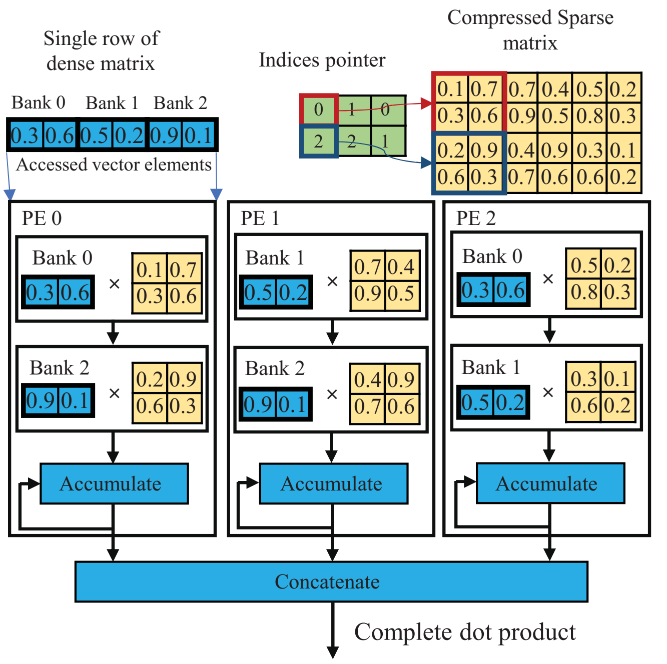


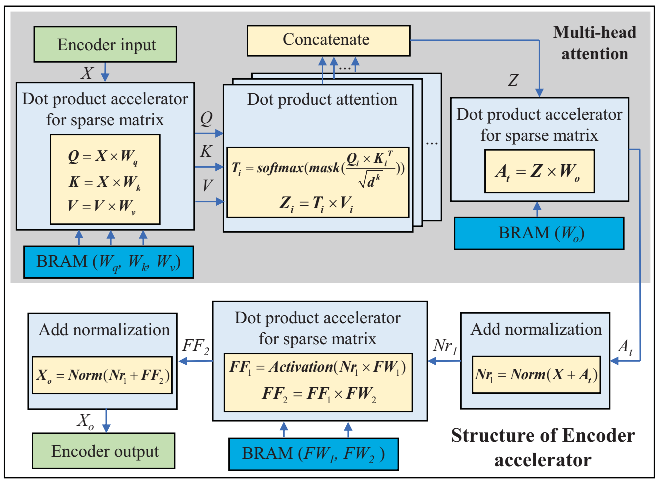
Fig. 5: Dot product accelerator for sparse matrix with column balanced block-wise sparsity pattern

be accumulated to obtain the final output. Inside the PE, the data fetching, GEMM, and accumulation processes execute in serial with an elaborate pipeline structure. It will iterate over the entire block column of the compressed sparse matrix to obtain the final output. To increase data throughput and enable intra-block parallelism, multiple PEs can be used in parallel. With multiple PEs, the BRAM memory partition of the compressed sparse matrix is required to aid the parallel computation. After each PE finished accumulation and iterates over the entire compressed sparse matrix, the result can be concatenated together to generate the final dot product output. The PE design concept can be applied to both FPGA hardware design and GPU kernel design.

C. Encoder Accelerator Structure

The overall structure of a single encoder layer is shown in Fig. 6. The encoder layer comprises a 4 sparse matrix dot product accelerator, an activation layer, 4 dot product attention hardware blocks for 4 heads, and 2 add normalization layers. 5 sparse weight matrices with their index matrices will be stored in BRAM to speed up the computation.

The data flow of the encoder layer is as follows. The encoder input X is firstly fed into the dot product accelerator to calculate matrix Q K, and V, and the accelerator output is fed into dot product attention hardware. The output matrices of the dot product attention are then concatenated into a single matrix Z. Z is fed into another dot product accelerator and generate matrix A_t. The matrix A_t is then passed through an add normalization layer, and the result is stored in the matrix Nr_1. Again, the matrix Nr_1 is fed into a special design dot product accelerator for two matrix multiplication and one



Encoder input
Concatenate
Multi-head attention
X
Dot product accelerator for sparse matrix
$Q = X \times W_q$
$K = X \times W_k$
$V = V \times W_v$
Q
K
V
Dot product attention
$T_i = softmax(mask(\frac{Q_i \times K_i^T}{\sqrt{d^k}}))$
$Z_i = T_i \times V_i$
Z
Dot product accelerator for sparse matrix
$A_t = Z \times W_o$
BRAM (W_q, W_k, W_v)
BRAM (W_o)
Add normalization
$X_o = Norm(Nr_1 + FF_2)$
FF_2
Dot product accelerator for sparse matrix
$FF_1 = Activation(Nr_1 \times FW_1)$
$FF_2 = FF_1 \times FW_2$
Nr_1
Add normalization
$Nr_1 = Norm(X + A_t)$
A_t
X_o
Encoder output
BRAM (FW_1, FW_2)
Structure of Encoder accelerator

Fig. 6: Encoder structure

activation function. In our design, we are using RELU for the activation function. The dot product accelerator output FF_1 is fed into another add normalization layer and generates the final output X_o.

D. Resource Scheduling

To allocate a reasonable amount of resources for each function, we adopt operation scheduling methods [24] for hardware design. The optimization task goal is:

$$\begin{aligned} \min_{\{\mathbf{W}_n\},\{\mathbf{b}_n\}} \quad & min(T_1, T_2, ...T_k) \\ \textbf{subject to} \quad & R_t \geq M \sum_{i=1}^{k} R_i + R_m \end{aligned} \tag{2}$$

In the function, $R_t = [R_{DSP}, R_{FF}, R_{LUT}, R_{BRAM}]$ is the total available resource on FPGA chip. R_i is the resource used by each function with an encoder/decoder. R_m is the resource used by the DDR controller, PCIe controller, or other types of miscellaneous function within the FPGA system. We firstly begin with a hardware design without any parallelism. Then we start to add hardware parallelism to the slowest function or loop and check the resource constraint. If the resource constraint is satisfied, we will add hardware parallelism to the slowest function or loop and do it over until the hardware resources and latency are optimized.

V. Experiments

A. Training of the Transformer Model

We conduct our experiment using the Transformer model on Wikitext-2 dataset [25], on which we use the accuracy of next word prediction for the benchmark. The Transformer model illustrated in the paper is a shallow well pre-trained, and unpruned model in Pytorch and GPU version, and it has two encoder layers and one decoder layer with hyperparameters of embedding layer size (emsize)=800 and number of hidden layer dimension (nhid)=200. The decoder layer of the Transformer is a fully connected layer with a 28,785 output dimension. In the experiment, we are using 32 as our batch size for both training and inference.

The Transformer model training is conducted on 4 RTX 2080Ti GPU servers (each with 4 or 8 GPUs). Experiments are implemented using python 3.6.10, GCC 7.3.0, PyTorch 1.4.0, and CUDA 10.1.

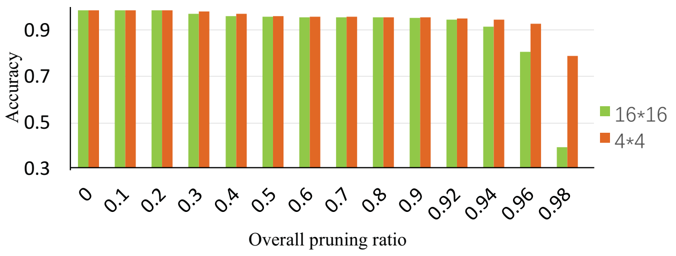


Fig. 7: Accuracy of transformer model at different pruning ratio using column balanced block-wise pruning method, two different block sizes (16*16 and 4*4) are compared

We first run 50 training epochs for training and obtain the model with the best accuracy as our pre-trained model. We load the pre-trained model and run 50 epochs iteration of admm-based training [26], pruning and retraining. For admm-based training, we set the initial learning rate as 3.0 and admm epoch as 9 and adjust the learning rate periodically. To be more specific, if the epoch is dividable by admm epoch, we reset the learning rate to the original one. Otherwise, we decay the learning rate every 1/3 admm epoch. For pruning, we set the prune ratio between 0 to 1 and employ the column balanced block-wise pruning method to prune the former trained model. Finally, we retrain the pruned model. For the prune ratio, we first set it in the range of 0 to 0.9 with an interval of 0.1. As we could easily observe from Fig. 7, there is little accuracy drop with the increasing prune ratio in this region with both block sizes. To find the turning point or the ”sweet spot”, we further explored the accuracy and sparsity curve between 0.9 to 1 with a smaller interval of 0.02. We finally choose prune ratio = 0.9 as the operation point, where the model has a high pruning ratio with a low accuracy drop. And at this point, the Transformer model with 16*16 block size has accuracy = 0.9512, comparable to accuracy = 0.9535 for 4*4 block size.

B. Hardware Performance

1) Hardware Platform: The Xilinx Alveo U200 board, which has 4,320 of 18k BRAM, 6,840 DSPs, and 1,882.2k logic cells (LUT), is used in the experiment. The FPGA board is connected with the host machine through PCIe for fetching the input word with 18 batch size. Xilinx SDX 2020.1 and the high-level synthesis tool (C/C++) are used for hardware development. The hardware inference performance is compared between Intel i5-5257U(2.7 GHZ) CPU, Nvidia Jetson TX2 GPU, and Xilinx Alveo U200 FPGA platforms for latency and throughput (frame/sequence per second). The batch size is chosen as 32 for both platforms for fairness comparison.

2) Resource Scheduling: We apply the hardware resource scheduling concept from section IV-D to increase the hardware throughput. The hardware scheduling results of each operation (e.g., Matrix Multiplication (MM), dot product attention, add normalization) in encoder and decoder are shown in Table I.

TABLE I: Resource scheduling for the Transformer on FPGA (prune ratio = 0.9, block size = 16*16 and batch size = 32)

	DSP	FF	LUT	Latency
Total hardware resources	6,840	2,364.5k	1,882.2k	N/A
Encoder	DSP	FF	LUT	Latency
Sparse MM accelerator 1	331	150.4k	150.8k	1.152 ms
Dot product attention $\times$ 4	292	59.6k	101.7k	0.554 ms
Sparse MM accelerator 2	168	23.3k	31.2k	0.704 ms
Add normalization 1	62	17.9k	18.3k	0.321 ms
Sparse MM accelerator 3	172	28.0k	26.1k	0.393 ms
Add normalization 2	62	17.9k	18.3k	0.321 ms
Resources for 1 encoder	1025	279.3k	389.3k	3.446 ms
Percentage	15.0%	11.8%	20.7%	N/A
Decoder	DSP	FF	LUT	Latency
Sparse MM accelerator 4	1318	98.8k	83.3k	3.456 ms
Resources for 1 decoder	1318	98.8k	83.3k	3.456 ms
Percentage	19.3%	4.2%	4.4%	N/A

We observe that through the developed resource scheduling technique, we achieve high resource utilization, i.e., 49.3%, 27.8%, 45.8% for DSP, FF, LUT respectively. The resultant latency is 3.446 ms and 3.456 ms for the single encoder and decoder, satisfying the real-time requirements for various NLP tasks on resource-constrained devices.

3) Cross Platform Comparison: After the resource scheduling is done for each encoder/decoder layer, the Transformer model can be combined from those layers. The final hardware latency of the Transformer model implemented on FPGA is 10.35 ms for batch size = 32 and block size = 16*16. The comparison of the Transformer model inference speed is made for different platforms, and the result is shown in Table II.

TABLE II: Comparison between CPU and FPGA (prune ratio = 0.9, block size = 16*16 and batch size = 32)

Hardware	latency(ms)	Throughput (FPS)
Intel i5-5257U (2.7 GHZ) CPU	113.40	282.2
Jetson TX2 GPU	21.54	1485.6
Xilinx Alveo U200 FPGA board	10.35	3091.8

As shown in the table, We achieve 10.95 $\times$ speed up on FPGA comparing to CPU and 2.08 $\times$ speed up comparing to GPU. The overall pruning technique and hardware design concepts enable efficient Transformer neutral network acceleration on the FPGA platform.

VI. Conclusion

This paper introduces an efficient Transformer acceleration framework for FPGA application. A column balanced block-wise pruning method is proposed, which achieves low accuracy decay under a high pruning ratio. A specialized process element for sparse matrix multiplication is designed to enable both inter block hardware parallelism and intra-block parallelism, and it can be applied to both GPU and FPGA devices. Then, the training process, as well as the accuracy versus pruning ratio relationship, is demonstrated. Finally, the hardware resource scheduling is done for the Transformer model, and the inference latency is compared between different hardware platforms. The overall framework achieves a reduced NLP model size with only a little accuracy drop, and the FPGA implementation achieves 10.95 $\times$ speed up comparing to the CPU platform and 2.08 $\times$ speed up comparing to the GPU platform.

References

[1] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio, “On the properties of neural machine translation: Encoder-decoder approaches,” *arXiv preprint arXiv:1409.1259*, 2014.

[2] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[3] A. X. M. Chang, B. Martini, and E. Culurciello, “Recurrent neural networks hardware implementation on fpga,” *arXiv preprint arXiv:1511.05552*, 2015.

[4] E. Nurvitadhi, J. Sim, D. Sheffield, A. Mishra, S. Krishnan, and D. Marr, “Accelerating recurrent neural networks in analytics servers: Comparison of fpga, cpu, gpu, and asic,” in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2016, pp. 1–4.

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[6] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding,” *arXiv preprint arXiv:1510.00149*, 2015.

[7] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, “Learning structured sparsity in deep neural networks,” in *Advances in Neural Information Processing Systems*, 2016, pp. 2074–2082.

[8] X. Ma, F.-M. Guo, W. Niu, X. Lin, J. Tang, K. Ma, B. Ren, and Y. Wang, “Pconv: The missing but desirable sparsity in dnn weight pruning for real-time execution on mobile devices.” in *AAAI*, 2020, pp. 5117–5124.

[9] Z. Wang, J. Wohlwend, and T. Lei, “Structured pruning of large language models,” *arXiv preprint arXiv:1910.04732*, 2019.

[10] Y. Guan, Z. Yuan, G. Sun, and J. Cong, “Fpga-based accelerator for long short-term memory recurrent neural networks,” in *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*. IEEE, 2017, pp. 629–634.

[11] C. Guo, B. Y. Hsueh, J. Leng, Y. Qiu, Y. Guan, Z. Wang, X. Jia, X. Li, M. Guo, and Y. Zhu, “Accelerating sparse dnn models without hardware-support via tile-wise sparsity,” *arXiv preprint arXiv:2008.13006*, 2020.

[12] S. Zheng, H. Lin, S. Zha, and M. Li, “Accelerated large batch optimization of bert pretraining in 54 minutes,” *arXiv preprint arXiv:2006.13484*, 2020.

[13] J. Xin, R. Tang, J. Lee, Y. Yu, and J. Lin, “Deebert: Dynamic early exiting for accelerating bert inference,” *arXiv preprint arXiv:2004.12993*, 2020.

[14] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint arXiv:1810.04805*, 2018.

[15] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, “Roberta: A robustly optimized bert pretraining approach,” *arXiv preprint arXiv:1907.11692*, 2019.

[16] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, “Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter,” *arXiv preprint arXiv:1910.01108*, 2019.

[17] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, “An image is worth 16x16 words: Transformers for image recognition at scale,” *arXiv preprint arXiv:2010.11929*, 2020.

[18] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, “End-to-end object detection with transformers,” *arXiv preprint arXiv:2005.12872*, 2020.

[19] Z. Yao, S. Cao, W. Xiao, C. Zhang, and L. Nie, “Balanced sparsity for efficient dnn inference on gpu,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 5676–5683.

[20] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu, and L. Zhang, "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*, 2019, pp. 63–72.

[21] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang *et al.*, "Ese: Efficient speech recognition engine with sparse lstm on fpga," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 75–84.

[22] B. Li, S. Pandey, H. Fang, Y. Lyv, J. Li, J. Chen, M. Xie, L. Wan, H. Liu, and C. Ding, "Ftrans: energy-efficient acceleration of transformers using fpga," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, 2020, pp. 175–180.

[23] J. de Fine Licht, S. Meierhans, and T. Hoefler, "Transformations of high-level synthesis codes for high-performance computing," *arXiv preprint arXiv:1805.08288*, 2018.

[24] S. Wang, Z. Li, C. Ding, B. Yuan, Q. Qiu, Y. Wang, and Y. Liang, "C-lstm: Enabling efficient lstm using structured compression techniques on fpgas," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 11–20.

[25] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," in *5th International Conference on Learning RepresentationsICLR*, 2017.

[26] T. Zhang, S. Ye, K. Zhang, J. Tang, W. Wen, M. Fardad, and Y. Wang, "A systematic dnn weight pruning framework using alternating direction method of multipliers," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 184–199.