Hardware Accelerator for Multi-Head Attention and Position-Wise Feed-Forward in the Transformer

Siyuan Lu, Meiqi Wang, Shuang Liang, Jun Lin, and Zhongfeng Wang
School of Electronic Science and Engineering, Nanjing University, Nanjing, China
Email: {sylu, mqwang, sliang}@smail.nju.edu.cn, {jlin, zfwang}@nju.edu.cn

2020 IEEE 33rd International System-on-Chip Conference (SOCC) | 978-1-7281-8746-4/20/$31.00 ©2020 IEEE | DOI: 10.1109/SOCC49529.2020.9524802

***Abstract*—Designing hardware accelerators for deep neural networks (DNNs) has been much desired. Nonetheless, most of these existing accelerators are built for either convolutional neural networks (CNNs) or recurrent neural networks (RNNs). Recently, the Transformer model is replacing the RNN in the natural language processing (NLP) area. However, because of intensive matrix computations and complicated data flow being involved, the hardware design for the Transformer model has never been reported. In this paper, we propose the first hardware accelerator for two key components, i.e., the multi-head attention (MHA) ResBlock and the position-wise feed-forward network (FFN) ResBlock, which are the two most complex layers in the Transformer. Firstly, an efficient method is introduced to partition the huge matrices in the Transformer, allowing the two ResBlocks to share most of the hardware resources. Secondly, the computation flow is well designed to ensure the high hardware utilization of the systolic array, which is the biggest module in our design. Thirdly, complicated nonlinear functions are highly optimized to further reduce the hardware complexity and also the latency of the entire system. Our design is coded using hardware description language (HDL) and evaluated on a Xilinx FPGA. Compared with the implementation on GPU with the same setting, the proposed design demonstrates a speed-up of 14.6× in the MHA ResBlock, and 3.4× in the FFN ResBlock, respectively. Therefore, this work lays a good foundation for building efficient hardware accelerators for multiple Transformer networks.**

***Index Terms*—Transformer, Natural Language Processing (NLP), Hardware Accelerator, FPGA, Neural Network**

I. Introduction

Recurrent neural networks (RNNs), long-short memory (LSTM) [7], and gated recurrent (GRU) [3], used to be the best solutions in the natural language processing (NLP) area. This situation was changed when the Transformer model [11] was invented in 2017, which outperforms previous RNN models in multiple tasks. By avoiding the recurrent calculations and taking full advantage of the attention mechanism, the Transformer and Transformer-based pre-trained language models (such as BERT [4], ALBERT [8], T5 [9], ERINE [10], and structBERT [14]) have achieved state-of-the-art accuracy in various NLP tasks.

In spite of making great progress in relative fields, the high computation complexity and huge memory requirements of these powerful Transformer networks are making them hard to be operated in mobile devices or embedded systems. More and more researchers are paying attention to this problem, and one way to solve it is through model compression [5]. Several techniques have been used to compress these networks, including data quantization [2], pruning, knowledge distillation and Architecture-Invariant Compression (AIC) [8].

Recently, building FPGA or ASIC hardware accelerators for deep neural networks (DNNs) has achieved great success in both academic and industrial societies, which makes us believe that designing efficient hardware architectures for these Transformer networks must be an important topic as well. By implementing them on hardware platforms, the inference systems of many NLP applications, such as machine translation, question answering, and sentiment analysis, are able to achieve higher speed or lower power consumption or both. However, intense matrix computations, complicated data flow, and complex nonlinear functions are making it hard to design efficient hardware architecture for the Transformer. To the best of our knowledge, we are the first to propose a specific hardware accelerator for the Transformer. In the open literature, the A^3 [6] is the only hardware architecture for accelerating the attention mechanism in various neural networks, which is not specifically designed for the Transformer.

As mentioned in [11] and [8], most of the trainable parameters and the computations are in the multi-head attention (MHA) ResBlock and the position-wise feed-forward network (FFN) ResBlock, which is discussed by Section II in detail. In this work, we design a reconfigurable hardware architecture based on systolic array (SA) for the MHA ResBlock and the FFN ResBlock, which are the two most complex layers in the Transformer.

Main contributions of this work can be summarized as follows:

- We provide an efficient method to partition the huge matrices in the Transformer, which allows the MHA ResBlock and the FFN ResBlock to share most of the hardware resources.
- We propose the first hardware architecture design which can complete the calculations for both these two ResBlocks. To ensure the high hardware utilization of the SA, which is the biggest module in our design, the computation flow is well designed.
- Two most complicated nonlinear functions, including the scaled masked-softmax and the layer normalization, are highly optimized to become more hardware-friendly. As the "bottle-neck" in the proposed architecture, the latency of layer normalization is reduced as much as possible.

This work was supported by the National Natural Science Foundation of China under Grant 61604068, the Fundamental Research Funds for the Central Universities under Grant 021014380065, the Key Research Plan of Jiangsu Province of China under Grant BE2019003-4. (Corresponding authors: Jun Lin; Zhongfeng Wang.)

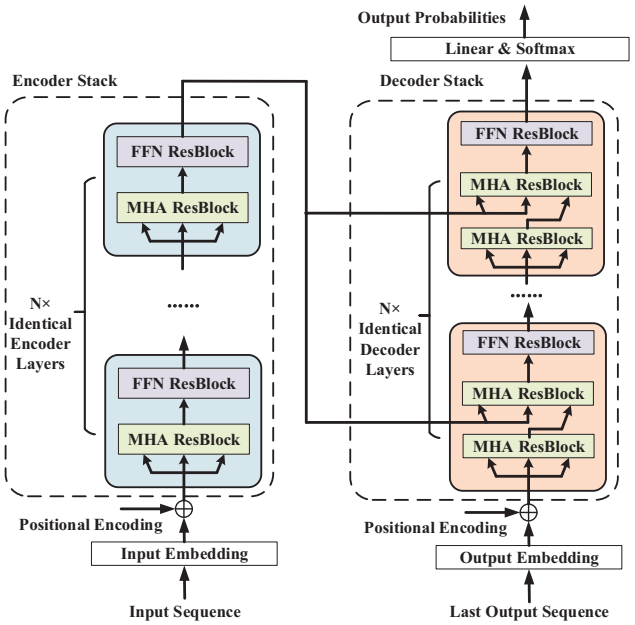


Fig. 1. The model architecture of the Transformer.

After quantizing the Transformer base model in [11] (distinguished from the Transformer big model) with 8-bit integers (INT8), we also evaluate our design on the Xilinx xcvu13p-fhga2104-3-e FPGA, when the max sequence length (denoted as s) is equal to 64 and the batch size is equal to 1. The hardware experimental results demonstrate a speed-up of 14.6× in the MHA ResBlock, and a speed-up of 3.4× in the FFN ResBlock, compared to a GPU implementation on an NVIDIA V100.

The rest of this paper is organized as follows. Section II gives a brief review of the Transformer networks, and explains the importance of accelerating the MHA ResBlock and the FFN ResBlock. Section III presents the method of matrix partitioning. Section IV describes the proposed hardware architecture. Experimental results are given in Section V. Section VI concludes this paper.

II. Background and Motivation

A. The Model Architecture of the Transformer

The model architecture of the Transformer is described in Fig. 1, containing an encoder stack and a decoder stack. Notice that most of the trainable parameters and the computations are in these two stacks, and other components beside the stacks such as the embedding layers and the softmax output layer have not been taken into account by this work. As is shown in Fig. 1, all the encoder layers and the decoder layers are composed of two kinds of ResBlocks, the MHA ResBlock and the FFN ResBlock.

Fig. 2 shows the structure of the MHA ResBlock. An MHA ResBlock has h "Attention Heads", and the input of each Head is the same as the input of the ResBlock, including three tensors: V (values), K (keys), and Q (queries). The Scaled

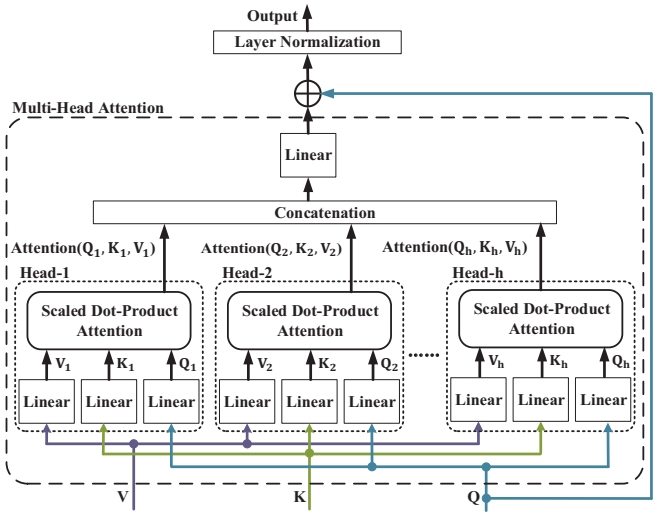


Fig. 2. The structure of the MHA ResBlock.

Dot-Product Attention function in the MHA can be expressed as follows:

$$Attention(Q_i, K_i, V_i) = softmax(Mask(\frac{Q_i K_i^T}{\sqrt{d_k}}))V_i. \quad (1)$$

The Mask operation is used to mask out all values in the input of the softmax corresponding to illegal connections, and the parameter d_k, which is equal to 64 in both the Transformer base model and the Transformer big model. The parameter h is equal to 8 in the base model, or equal to 16 in the big model.

The FFN ResBlock contains a fully connected feed-forward network, consisting of two linear sublayers and a ReLU activation between them:

$$\begin{aligned} FFN(x) &= ReLU(xW_1 + b_1)W_2 + b_2, \\ FFN\_ResBlock(x) &= LayerNorm(x + FFN(x)). \end{aligned} \quad (2)$$

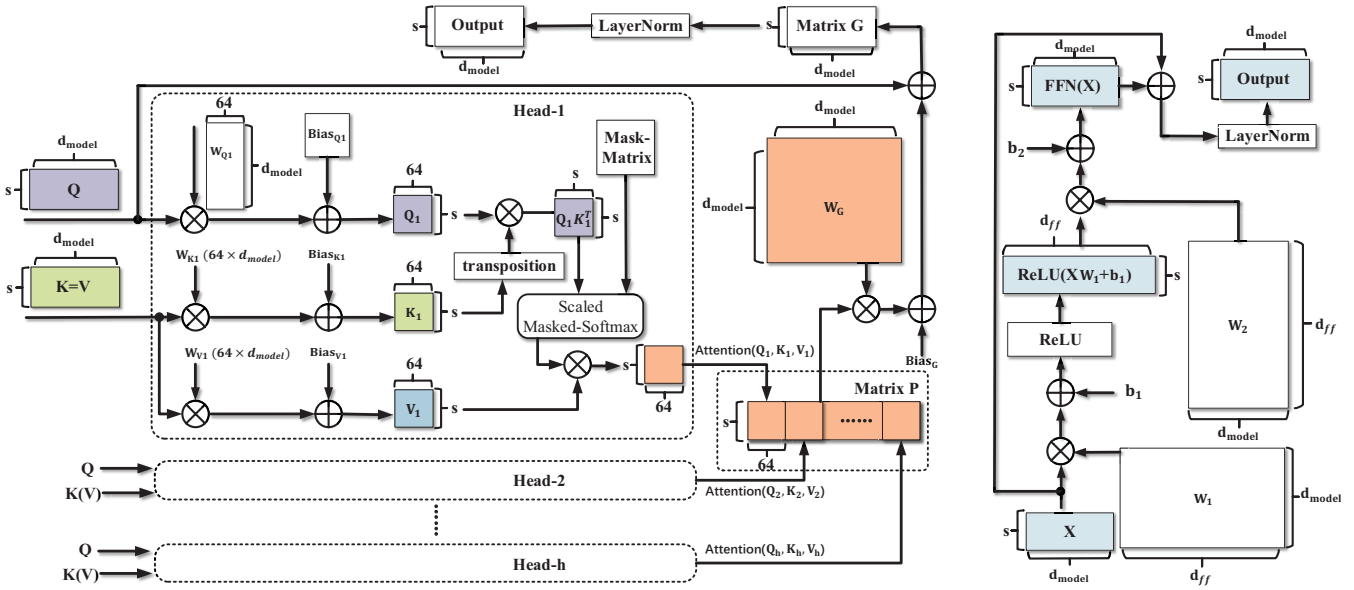
B. Transformer-Based Pre-Trained Models

An important pre-trained model is Bidirectional Encoder Representations from Transformers (BERT). Analyses in [5] also point out that, the MHA and the FFN ResBlocks still occupy most of the storage space and have the highest numbers of FLOPs.

The General Language Understanding Evaluation (GLUE) benchmark [12] is a collection of diverse natural language understanding tasks. Recently, many Transformer-based pre-trained models have obtained top placements on the GLUE score list. Most of these models, such as T5 [9], ERINE [10], and structBERT [14], have very similar structure to the BERT. These facts all prove the necessity of designing efficient hardware accelerators for the MHA and the FFN ResBlocks, which are two commonly used structures in these models.

III. Partitioning Matrices in the FFN and the MHA

Considering the characteristics of the Transformer architecture, we believe that the proposed hardware accelerator should be able to accelerate not only the MHA ResBlock, but also the FFN ResBlock. To make sure that the MHA ResBlock and the FFN ResBlock can reuse the hardware resources, we first



(a) The MHA ResBlock. (b) The FFN ResBlock.

Fig. 3. Matrix Operations in the MHA and the FFN ResBlocks. Note that all the multiply operations marked in this figure are dealing with cross products.

analyze these two ResBlocks from the perspective of matrix operations, and then give a method to partition the matrices so that all the general matrix-matrix multiplications (GEMMs) can be done with one and the same systolic array (SA), the size of which is limited to $s \times 64$.

Assuming that the input of the FFN is called X, the shape of the tensor X is the same as Q (one of the input tensors of the MHA), which is $[batch\_size, seq\_len\_q, d_{model}]$. Additionally, Fig. 1 shows that the tensor K is always equal to the tensor V, the shape of which is $[batch\_size, seq\_len\_v, d_{model}]$. In normal circumstances, seq_len_q is equal to seq_len_v, so the shape of all these four tensors can be expressed as $[batch\_size, s, d_{model}]$. Supposing that the batch size is equal to 1, the computations of these two ResBlocks can be considered sets of matrix operations, which are represented in Fig. 3. Obviously, an $s \times 64$ SA can support all the matrix multiplications in the Linear sublayers of all the Heads. However, how to complete other multiplications between larger matrices, including $P \times W_G$, $X \times W_1$, and $ReLU(XW_1 + b_1) \times W_2$, is another important issue to be considered.

TABLE I
VARIATIONS ON THE TRANSFORMER AND THE BERT ARCHITECTURES.

	d_{model}	d_{ff}	h
Transformer-base	512	2048	8
Transformer-big	1024	4096	16
$BERT_{BASE}$	768	3072	12
$BERT_{LARGE}$	1024	4096	16

Table I shows that in these Transformer networks, we all have $d_{model} = 64h$, and $d_{ff} = 4d_{model} = 256h$. On the basis of this pattern, the three large weight matrices W_G, W_1, and W_2 can be partitioned as shown in Fig. 4. Thus, most of the GEMMs can be done with an $s \times 64$ SA.

The only one left is the operation of $Q_i \times K_i^T$ in each Head

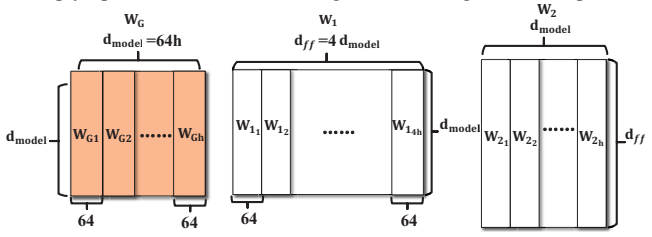


Fig. 4. Partition W_G, W_1, and W_2.

of the MHA. The ratio of the number of multiplications in this operation to the entire MHA ResBlock can be roughly calculated as follows:

$$\frac{s^2 64^2 h}{s^2 64^2 h + 3(64s(d_{model})^2)h + s(d_{model})^3 + (64s^3)h} = \frac{s}{s + 256h^2 + 64}. \quad (3)$$

Since $256h^2$ is no smaller than 16,384 and s is usually no bigger than 128, this ratio should be very small, which illustrates that the management of this single operation will not influence the overall hardware utilization much. If s is smaller than 64, it can be done with the $s \times 64$ SA through zero padding to the K_i. Otherwise by partitioning the Q_i, the $s \times 64$ SA can still support this operation with little impact on the utilization of the SA.

IV. HARDWARE ARCHITECTURE DESIGN FOR THE PROPOSED ACCELERATOR

Using the proposed method of partitioning these weight matrices, the complete hardware accelerator is designed. The top-level architecture is illustrated in Fig. 5.

The $s \times 64$ SA is made up of a 2D array of processing elements (PE), with s rows and 64 columns. It is designed to output the product matrix column by column, so each column has s elements. Connected to the SA output, s adders are required to add the bias to the product matrix, and another

Algorithm 1: The Overall Computation Flow

1 **if** *Calculating MHA ResBlock* **then**
2 **for** $i = 1; i \leq h; i++$ **do**
3 Temp1=$QW_{Qi} + Bias_{Qi}$;
4 Temp2=$KW_{Ki} + Bias_{Ki}$;
5 Softmax Input=$Temp1 \times Temp2^T$;
6 Temp1=Softmax output,
  Temp2=$VW_{Vi} + Bias_{Vi}$;
7 P_i=$Temp1 \times Temp2$;
8 **end**
9 **for** $i = 1; i \leq h; i++$ **do**
10 G_i=P$\times W_{Gi} + Bias_{Gi} + Q_i$;
11 **end**
12 Output=LayerNorm(G);
13 **end**
14 **if** *Calculating FFN ResBlock* **then**
15 **for** $i = 1; i \leq 4h; i++$ **do**
16 P_i=ReLU$(XW_{1_i} + b_{1_i})$;
17 **end**
18 **for** $i = 1; i \leq h; i++$ **do**
19 G_i=$PW_{2_i} + b_{2_i} + X_i$;
20 **end**
21 Output=LayerNorm(G);
22 **end**
23 **return** *Output*

s adders are required to add the residual before calculating the layer normalization function. Overall, the SA Module has the highest computational complexity, containing at least $64s$ multipliers and $64s$ adders. To increase the hardware utilization, we make the calculations of the Softmax Module running parallel to $V \times W_{v_i} + Bias_{V1}$ (line 6 in Algorithm 1). Owing to carefully designing the computation flow of the entire system, the SA Module will hardly stop running until the LayerNorm Module starts. As long as the Softmax module can give the output no later than the SA module finishing calculating "$VW_{Vi}+Bias_{Vi}$", the latency of the entire system will be determined by the SA module and the LayerNorm module. The architectures of these two nonlinear modules are introduced in detail as follows.

A. Scaled Masked-Softmax

The Softmax module in the proposed architecture is used to calculate the scaled masked-softmax function. For the convenience of discussion, we named the input matrix $Q_i \times K_i^T$ (refer to line 5 in Algorithm 1) as D, the shape of which is $s \times s$. The output matrix is defined as Y, and the mask matrix is defined as M. Therefore, the scaled masked-softmax function can be expressed as:

$$Y(i,j) = \begin{cases} exp(\frac{D(i,j)}{8}) / \sum_{j=1,M(i,j)=0}^{s}(exp(\frac{D(i,j)}{8})) & M(i,j) = 0, \\ 0 & M(i,j) = 1. \end{cases} \tag{4}$$

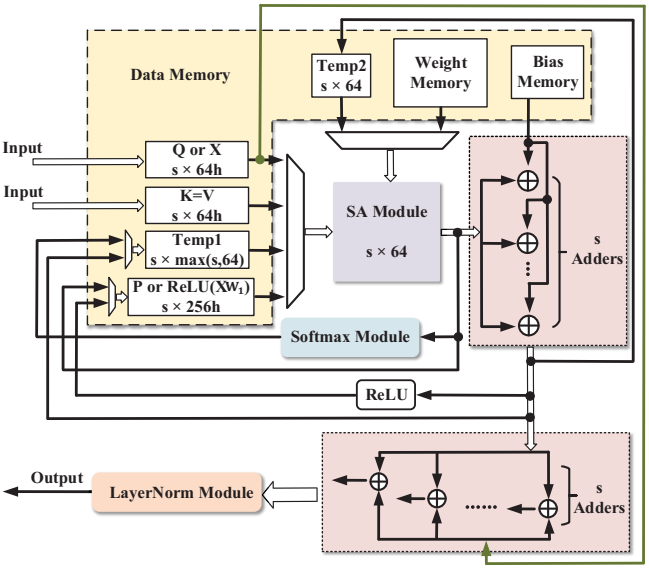


Fig. 5. The top-level architecture of our design.

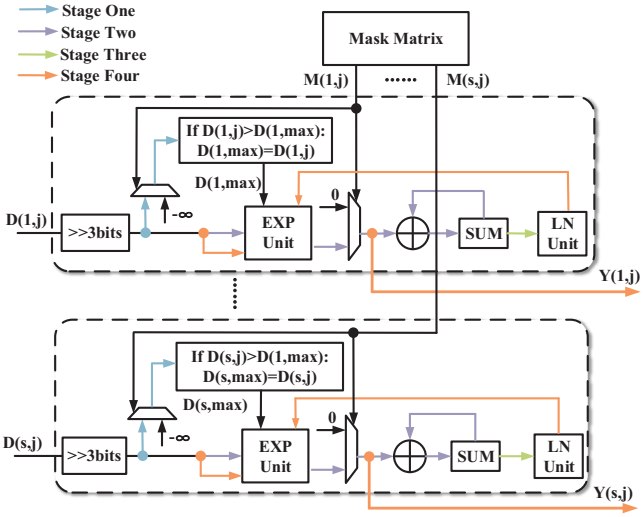


Fig. 6. The architecture of Softmax module. The "$>>$" denotes right shift operation.

Although the computational complexity of this Softmax Module is lower than the SA module, the exponentiation and division calculations are still quite expensive. In [13], by making good use of the log sum-exp trick [15] and designing algorithmic reduction strategies for exponential function and logarithmic function, a high-speed and low-complexity hardware architecture for softmax function was proposed. These tricks and strategies are also used in this work to build an efficient architecture for scaled masked-softmax. The division calculation and numerical underflow can be avoided by using the log-sum-exp trick ($\forall i \in 1, 2, ..., d_k, \chi_{max} \geq \chi_i$):

$$\begin{aligned} Softmax(\chi_i) &= \frac{exp(\chi_i - \chi_{max})}{\sum_{j=1}^{d_k} exp(\chi_j - \chi_{max})} \\ &= exp(\chi_i - \chi_{max} - ln(\sum_{j=1}^{d_k} exp(\chi_j - \chi_{max}))) \end{aligned} \tag{5}$$

According to Equation (5), the computation of this module can be broken into four different phases, which is described in Fig. 6. The transformations of exponential function and logarithmic function allow us to build the Softmax module without using any regular multipliers and lookup tables. The detailed architectures of the EXP Unit and the LN Unit are the same as [13].

B. Layer Normalization

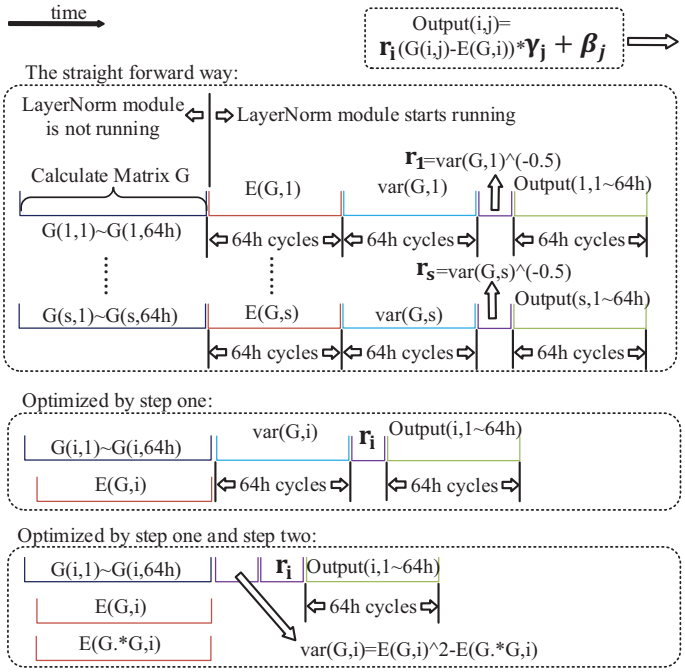


Fig. 7. The method to minimize the latency of the LayerNorm module.

As discussed in Section II, both of these two ResBlock have to calculate the layer normalization function before the output starts. This means that the LayerNorm module is always on the critical path of the system latency. In this subsection, we propose a method to minimize its latency.

Unlike the batch normalization, the layer normalization does not impose any restriction on the size of a mini-batch. So it is able to be used in the pure online regime with the batch size equal to 1. [1]. The layer normalization function used in these two ResBlocks is:

$$Output(i,j) = \frac{G(i,j) - E(G,i)}{\sqrt{var(G,i) + \varepsilon}}\gamma_j + \beta_j, \tag{6}$$

where the constant ε is equal to 10^{-8}, which is used to avoid the denominator from being zero. The variable $E(G,i)$ is the mean value of all the elements in the i-th row of matrix G ($s \times d_{model}$):

$$E(G,i) = \frac{1}{d_{model}} \sum_{k=1}^{d_{model}} G(i,k). \tag{7}$$

The variance of these elements is defined as:

$$var(G,i) = \frac{1}{d_{model}} \sum_{k=1}^{d_{model}} [(G(i,k) - E(G,i))^2]. \tag{8}$$

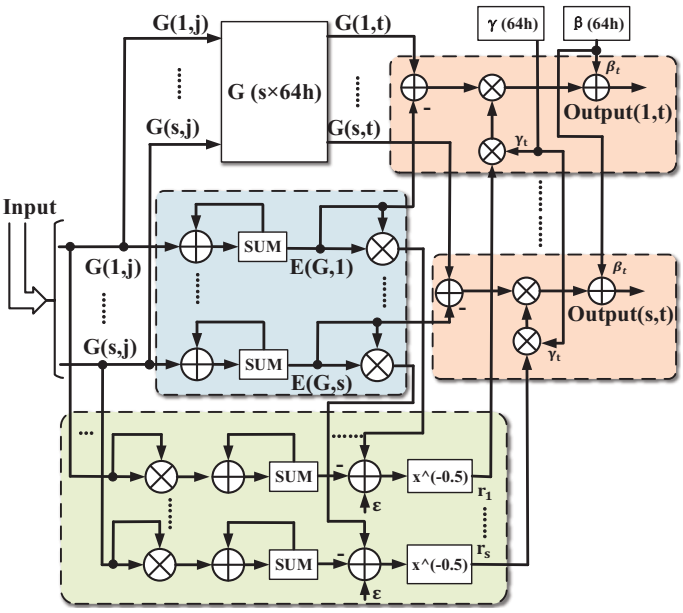


Fig. 8. The architecture of LayerNorm module.

According to these above equations, the straightforward way to calculate the layer normalization is described in Fig. 7. To calculate $E(G)$ and $var(G)$, at least $128h$ cycles are added to the whole system latency.

As is shown in Fig. 7, there are two steps in our method of minimizing the delay of this module, and the key is to make the LayerNorm module start running in advance. The first step is using s accumulators to calculate $\sum_{k=1}^{d_{model}} G(i,k)$, and keeping them connected directly to the input of this module. The second step is choosing another way to calculate the variance:

$$var(G,i) = E(G,i)^2 - \frac{1}{d_{model}} \sum_{k=1}^{d_{model}} G(i,k)^2. \tag{9}$$

At last, very few cycles are required between the system finishing calculating all the elements of matrix G and starting the output, which also means the latency of the entire system is further reduced. The architecture of the LayerNorm module is described in Fig. 8. The "x^(-0.5)" unit is implemented with a lookup table in our experiment.

V. Experimental Results

A. Quantization of Transformer Base Model

Before evaluating our complete design with FGPA, we quantize a Transformer base model for a machine translation task [1]. This model has been trained and tested with IWSLT 2016 German-English parallel corpus, and the test BLEU score is 23.88 on "tst2014". Learning from [2], replacing FP32 with INT8 in the Transformer can greatly reduce the computational complexity with limited accuracy loss.

Since linear approximation is used in the exponential function and the logarithmic function of the Softmax module, the

[1]https://github.com/Kyubyong/transformer

process of the quantization is divided into two steps. First, all the trainable variable matrices and activation matrices in Fig. 3 are all quantized with INT8, but the internal calculations in the Scaled Masked-Softmax operation are still implemented with FP32. After that the BLEU score drops to 23.48, proving that quantizing with INT8 in this network is acceptable. Second, the Softmax module is quantized based on the fixed-point model built in the first step. The previously mentioned log-sum-exp trick and the transformations of exponential function and logarithmic function are used. The final BLEU score of the quantized Transformer base model is 23.57, which is even a little higher than 23.48. These results also show that using the simplified architecture for softmax designed in [13] has little impact on the accuracy of this translation task.

B. Hardware Implementation Results

By setting the batch size to 1 and the max sequence length to 64, the proposed architecture is evaluated on Xilinx xcvu13p-fhga2104-3-e FPGA by using the Vivado 2018.2. The simulation results show that it takes 21,344 cycles and 42,099 cycles to finish the calculation of MHA ResBlock and FFN ResBlock, respectively. The Vivado implementation results show that our design can run up to 200MHz, and the total on-chip power is 16.7W (13.3W dynamic power and 3.4W device static power). The utilization report is presented in TABLE II.

TABLE II
UTILIZATION REPORT FOR THE PROPOSED HARDWARE ACCELERATOR AND ITS PRIMARY MODULES

	LUT	CLB Registers	BRAM	DSP
Available	1728000	3456000	2688	12288
Top	471563	217859	498	129
64×64 SA	420867	173110	0	0
Softmax	21190	32623	0	0
LayerNorm	10551	5325	27.5	129
Weight Memory	3379	80	456	0

Using the same hyper parameters (batch size equal to 1 and max sequence length equal to 64), we also measure the latency of these two layers in a GPU implementation of the Transformer base model [2] on an NVIDIA V100. The comparison results are shown in TABLE III, proving that our design is able to accelerate the inference for the Transformer on FPGA platform.

TABLE III
COMPARISONS BETWEEN FPGA AND GPU LATENCY RESULTS

	FPGA Latency	GPU Latency	Speed-Up
MHA ResBlock	106.7us	1557.8us	14.6×
FFN ResBlock	210.5us	713.4us	3.4×

VI. CONCLUSION AND FUTURE WORK

In this work, we present the first hardware accelerator for the MHA ResBlock and the FFN ResBlock in the Transformer. The FPGA implementation shows promising results in terms of both speed and power, which demonstrates that this design can contribute to operating the Transformer network in mobile device or embedded systems. In the future, we will build a FPGA or ASIC accelerator for the complete Transformer inference.

[2]https://github.com/jadore801120/attention-is-all-you-need-pytorch

REFERENCES

[1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffery E. Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[2] Aishwarya Bhandare, Vamsi Sripathi, Deepthi Karkada, Vivek Menon, Sun Choi, Kushal Datta, and Vikram Saletore. Efficient 8-bit quantization of transformer neural machine language translation model. *arXiv preprint arXiv:1906.00532*, 2019.

[3] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NIPS 2014 Workshop on Deep Learning, December 2014*, 2014.

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

[5] Prakhar Ganesh, Yao Chen, Xin Lou, Mohammad Ali Khan, Yin Yang, Deming Chen, Marianne Winslett, Hassan Sajjad, and Preslav Nakov. Compressing large-scale transformer-based models: A case study on bert. *arXiv preprint arXiv:2002.11985*, 2020.

[6] Tae Jun Ham, Sung Jun Jung, Seonghak Kim, Young H Oh, Yeonhong Park, Yoonho Song, Jung-Hun Park, Sanghee Lee, Kyoung Park, Jae W Lee, et al. a^3: Accelerating attention mechanisms in neural networks with approximation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 328–341. IEEE, 2020.

[7] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[8] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. In *International Conference on Learning Representations*, 2019.

[9] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*, 2019.

[10] Yu Sun, Shuohuan Wang, Yukun Li, Shikun Feng, Xuyi Chen, Han Zhang, Xin Tian, Danxiang Zhu, Hao Tian, and Hua Wu. Ernie: Enhanced representation through knowledge integration. *arXiv preprint arXiv:1904.09223*, 2019.

[11] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 6000–6010, 2017.

[12] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 353–355, 2018.

[13] Meiqi Wang, Siyuan Lu, Danyang Zhu, Jun Lin, and Zhongfeng Wang. A high-speed and low-complexity architecture for softmax function in deep learning. In *2018 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pages 223–226. IEEE, 2018.

[14] Wei Wang, Bin Bi, Ming Yan, Chen Wu, Zuyi Bao, Liwei Peng, and Luo Si. Structbert: Incorporating language structures into pre-training for deep language understanding. *arXiv preprint arXiv:1908.04577*, 2019.

[15] Bo Yuan. Efficient hardware architecture of softmax layer in deep neural network. *2016 29th IEEE International System-on-Chip Conference (SOCC)*, pages 323–326, 2016.