Electronic Research Archive

http://www.aimspress.com/journal/ERA

ERA, 30(10): 3755–3785.
DOI: 10.3934/era.2022192
Received: 18 June 2022
Revised: 18 July 2022
Accepted: 27 July 2022
Published: 16 August 2022

Review

Hardware-friendly compression and hardware acceleration for transformer: A survey

Shizhen Huang[1], Enhao Tang[1], Shun Li[1], Xiangzhan Ping[2] and Ruiqi Chen[3,*]

[1] College of Physics and Information Engineering, Fuzhou University, Fuzhou 350116, China
[2] Department of Optoelectronic Information Engineering, Chongqing University of Posts and Telecommunications, Chongqing 400065, China
[3] Zhangjiang Fudan International Innovation Center, Fudan University, Shanghai 200433, China

* **Correspondence:** Email: ruiqichen@ieee.org.

Abstract: The transformer model has recently been a milestone in artificial intelligence. The algorithm has enhanced the performance of tasks such as Machine Translation and Computer Vision to a level previously unattainable. However, the transformer model has a strong performance but also requires a high amount of memory overhead and enormous computing power. This significantly hinders the deployment of an energy-efficient transformer system. Due to the high parallelism, low latency, and low power consumption of field-programmable gate arrays (FPGAs) and application specific integrated circuits (ASICs), they demonstrate higher energy efficiency than Graphics Processing Units (GPUs) and Central Processing Units (CPUs). Therefore, FPGA and ASIC are widely used to accelerate deep learning algorithms. Several papers have addressed the issue of deploying the Transformer on dedicated hardware for acceleration, but there is a lack of comprehensive studies in this area. Therefore, we summarize the transformer model compression algorithm based on the hardware accelerator and its implementation to provide a comprehensive overview of this research domain. This paper first introduces the transformer model framework and computation process. Secondly, a discussion of hardware-friendly compression algorithms based on self-attention and Transformer is provided, along with a review of a state-of-the-art hardware accelerator framework. Finally, we considered some promising topics in transformer hardware acceleration, such as a high-level design framework and selecting the optimum device using reinforcement learning.

Keywords: transformer; hardware accelerators; self-attention; compression; FPGA

1. Introduction

The Transformer [1] has demonstrated impressive performance gains in Natural Language Processing (NLP) tasks, including Machine Translation, Text Categorization and Language Modeling [2–4]. Because the Transformer can process data in any order, it is possible to train on large volumes of data that would not have been possible. Likewise, this has led to the creation of pretrained models, such as Bert [5] and RoBERTa [6], which have achieved breakthroughs in several natural language understanding tasks, including Sentiment Analysis [7] and Semantic Role Labeling [8]. The Transforms and its variant models have become the backbone of many NLP tasks in modern times.

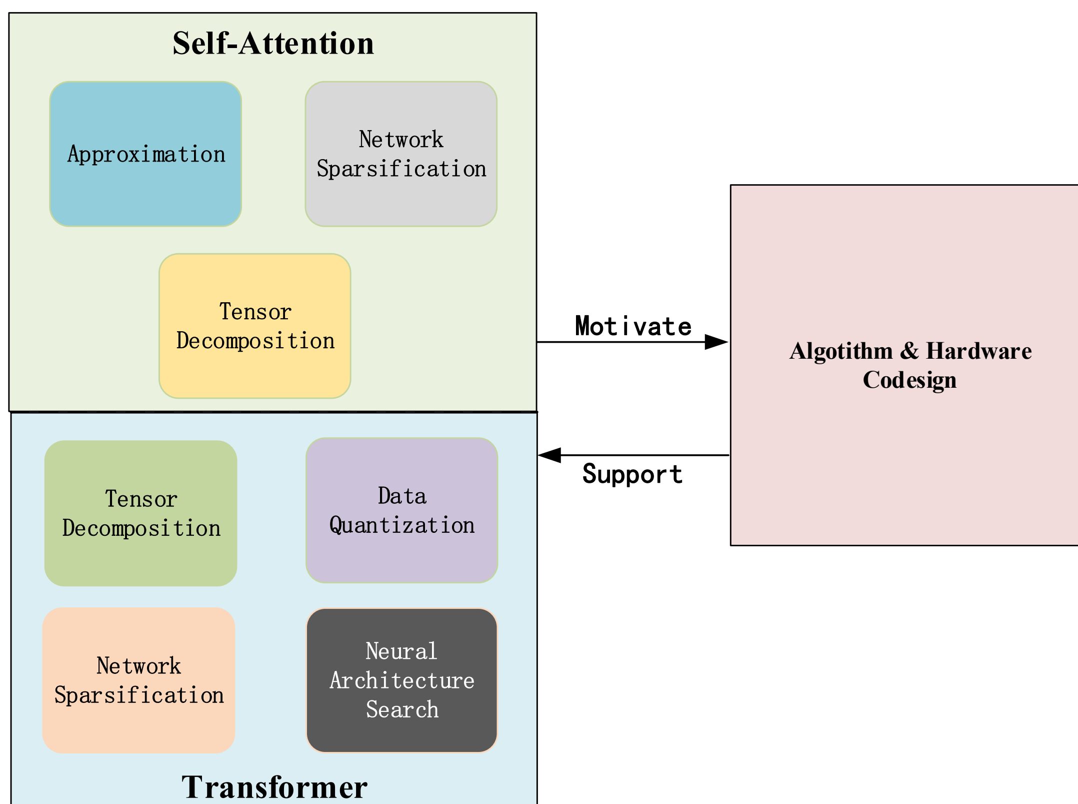

Figure 1. Overview of transformer compression and acceleration.

However, the impressive performance of the Transformer is not only due to the innovation of the model but also to the improvement in conventional processing, that is, the advent of Graphics Processing Units (GPU). As the size of the transformer model continues to increase, the traditional Central Processing Unit (CPU) can no longer withstand the high time delay brought by the deployment of the Transformer, and high latency is also a consequence of the dramatic increase in memory bandwidth and computational complexity. For example, the Transformer usually has millions of parameters, such as the BERT [5] model has 340M parameters, and the BERT model after distill has 67M parameters [9]. Therefore, GPU with high parallelism and memory bandwidth has become the main platform for transformer model training and inference in cloud computing. On the one hand, with the development of high-performance computing, corresponding requirements are put forward for the system's power consumption [9]. On the other hand, due to the excellent performance of transformers

in the field of NLP, the trend of applying this model on mobile terminals is becoming more and more obvious, such as mobile phones, tablets, etc. Such platforms also have great requirements for power consumption. FPGA takes into account the characteristics of low power consumption and high performance. It is very suitable as an acceleration platform for Transformer. FPGA is also widely used to accelerate deep learning algorithms because of its high parallelism, low latency, and low power consumption. At the same time, for acceleration platforms like Transformer and Deep Neural Network (DNN), FPGAs exhibit higher energy efficiency than GPUs and CPUs. From this, it can be seen that the need for Domain-specific hardware accelerators with specialized and customized deployment operations and memory hierarchies is becoming more and more obvious. In addition, because the development tools of GPU are very mature now, it is difficult to optimize the hardware architecture in the development process of using the C language to complete the entire function. Therefore, the hardware accelerators in this article refer to ASIC and FPGA. However, deploying huge models like transformers on these accelerated platforms is challenging because they often have limited on-chip memory, off-chip bandwidth, and resources. Therefore, compressing transformer model parameters and reducing computational cost has become an urgent topic. The compression model enables effective reduce memory and computational cost [10,11]. The compression techniques are driven by usage [12,13], data [14–16], transmission [17], and model [18,19]. However, it’s important to consider system performance (e.g., latency, energy cost, storage, and processing capability). In terms of algorithms, various algorithms for compressing transformers have been proposed in recent years, such as Knowledge Distillation [9,20–23], Network Sparsification [24–27], Data Quantization [28–31], Neural Architecture Search [32–34]. However, the model parameters processed by the compression algorithm are irregular in the memory, leading to irregular memory access, greatly consuming hardware resources, and reducing the operation speed, which is extremely unfriendly to hardware. And a survey on the compression algorithm of the transformer model has also been organized [35].

However, a comprehensive survey of the corresponding hardware-deployed accelerated Transformer has not emerged by adopting hardware-friendly compression transformer algorithms and corresponding joint algorithms. Therefore, this paper’s purpose is to review the past hardware-friendly transformer compression algorithms and to give the hardware architecture for the application of the algorithm. Because self-attention is a very important part of the Transformer, most of the Transformer’s running time is mainly on the self-attention mechanism. Therefore, we also reviewed the hardware-friendly compression algorithm of the self-attention mechanism and the hardware architecture of deploying the self-attention mechanism on hardware in conjunction with the corresponding compression algorithm. Figure 1 shows our classification of the hardware-friendly compression algorithms of self-attention and introduces the algorithm and hardware co-design of the above algorithms, that is, the related hardware architecture of self-attention and Transformer combined with the above algorithms for hardware acceleration is introduced. Finally, we conclude and discuss several interesting and promising topics in this field.

The overall architecture of this paper is shown in Table 1. Section 2 briefly introduces the basics of the transformer model. The third section details the hardware architecture of the self-attention mechanism combined with the compression algorithm in hardware and the corresponding compression algorithm. The fourth section mainly introduces the hardware-friendly compression algorithm for the Transformer and the hardware architecture that combines this algorithm. The fifth section discusses the future trend and summary.

Table 1. Content guidance of this article.

Preliminaries	Brief Preliminaries of Transformer		Section 2
Self-attention	Algorithm	Approximation	Section 3.1.1
		Network Sparsification	Section 3.1.2
		Tensor Decomposition	Section 3.1.3
	Hardware	Accelerators with Approximation	Section 3.2.1
		Accelerators with Network Sparsification	Section 3.2.2
		Accelerators with Tensor Decomposition	Section 3.2.3
Transformers	Algorithm	Tensor Decomposition	Section 4.1.1
		Data Quantization	Section 4.1.2
		Network Sparsification	Section 4.1.3
		Neural Architecture Search	Section 4.1.4
	Hardware	Accelerator with Tensor Decomposition	Section 4.2.1
		Accelerator with Data Quantization	Section 4.2.2
		Accelerator with Network Sparsification	Section 4.2.3
		Accelerator with Neural Architecture Search	Section 4.2.4
Conclusions and Discussion	Conclusions		Section 5.1
	Discussion		Section 5.2

2. Brief preliminaries of transformer

In this section, we briefly introduce the architecture and background of the transformer and variant model BERT. In the past, the state-of-the-art methods for Language Modeling and Machine Translation were long-short term memory (LSTM) [36], Gated Recurrent Unit (GRU) [37], and Self-Attentional mechanism [1,38]. However, they are performed in a loop, and the running time is linear with the sequence length, making it difficult to parallelize. The Transformer completely abandons this repetitive idea and uses a Self-Attentional mechanism to describe the dependency between input and output.

Figure 2 shows the overall architecture of the Transformer. Generally speaking, the Transformer is composed of Encoder and Decoder stacked, and the Encoder and Decoder are also stacked with Self-attention and a fully connected (FC) layer. Figure 3(b) shows the calculation process of Self-attention. The role of the Encoder is to use the Self-Attentional mechanism to convert the input sequence into a digital code, and the role of the Decoder is to convert the digital code output by the Encoder into the corresponding output format. First, the input sequence passes through the Inputs Embedding layer to map each word into a vector. That is, the input sequence is adapted into a matrix form by the inputs embedding layer to express, and the Positional Encoding layer is used to show the word's position. Because the Transformer completely abandons RNN and CNN, the Transformer has no word order information. In order to introduce word order information, the Transformer introduces Positional encoding as the relative or absolute position information of the input sequence and then adds Positional encoding and Input Embedding so that the complete information of the input sequence can be expressed. At the same time, the output part of the Output Embedding and Positional encoding is added, and then they are input into the Decoder.

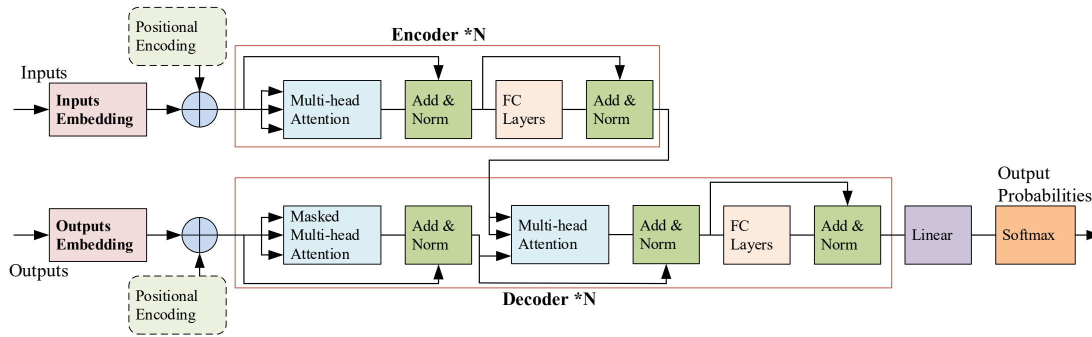

Figure 2. Model structure of transformer [39].

Encoder: The Encoder consists of N identical layers. Each layer is composed of Multi-head Attention, Add & Norm, and FC Layers. Multi-head Attention and FC Layers will be connected to Add & Norm, and Add & Norm means Norm after Add operation. Add here can help the model explore deeper because there is no risk of gradient vanishing.

Decoder: Decoder is also composed of N identical layers. Each layer consists of three sublayers: Masked Multi-head attention Layer, Multi-head attention Layer, and FC Layer. Each sublayer is also connected to an Add & Norm block, which is similar to Encoder. The Masked Multi-head attention layer masks the incoming output sequence and masks the words after the currently processed word to ensure that the prediction of the current position is only related to the word in its previous position. The Multi-head attention layer is different from the Encoder layer. The Multi-head attention of the Decoder plays a role in interacting with the output of the Encoder. It accepts the Encoder's key matrix and value matrix output and combines it with the Decoder's query matrix to achieve interaction.

Multi-head Attention: As shown in Figure 3, it is obvious that the Multi-head Attention mechanism is a combination of a single Self-Attention Mechanism, and its function is that the model can obtain information from multiple spaces, thereby capturing more feature information. The expressions are shown in Eqs (2) and (3). Therefore, we mainly introduce Self-Attention Mechanism. From Eq (1), it can be seen that the Self-Attention function is not complicated. The input is only three matrices of Q, K, and V, and both Self-Attention Mechanism and the entire Transformer are a matrix operation. QK^T represents the inner product operation to obtain the similarity score, and then the corresponding weight can be obtained through the softmax function, and d_k is a scale factor to prevent the gradient vanishing due to too small gradient when backpropagation is performed during training.

$$O_{attention} = Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \quad (1)\ [39]$$

$$O_{multi-head} = concat(O_{attention}^i) \quad (2)$$

$$O_{attention}^i = Attention(Q_i, K_i, V_i) = softmax\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right)V_i \quad (3)$$

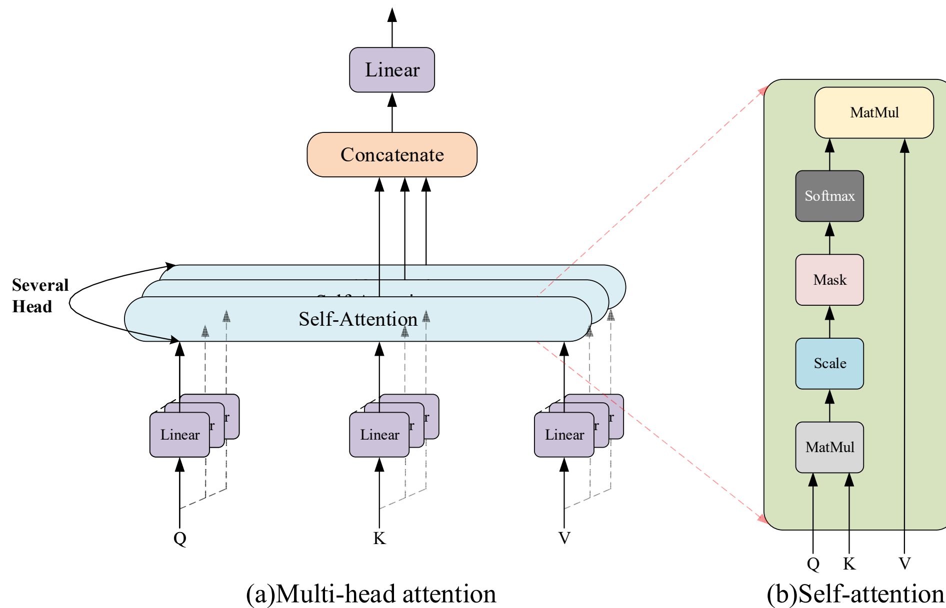

Figure 3. (a) Multi-head attention (b) Self-attention.

Linear and Softmax: Linear and Softmax are connected after Decoder. Linear can map the vector output by the Decoder to a log-odds probability vector. For example, the transformer model learns 10,000 different English words from the training set, so the log-odds vector is a vector of length of 10,000 cells, each cell corresponding to the score of a word. Next, the Softmax function converts this score into a probability, the highest probability is selected, and its corresponding word is the time step output.

3. Self-attention mechanism

In the past, the state-of-the-art methods for Language Modeling and Machine Translation were LSTM, GRU, and Self-Attentional mechanism. However, they are performed in a loop, and the running time is linear with the sequence length, making it difficult to parallelize. The Transformer completely abandons this repetitive idea and instead uses a self-attention mechanism to describe the dependency between input and output. So Self-attention is very important for Transformer. In this section, we introduce the hardware-friendly compression algorithm used by the Self-Attention Mechanism on hardware in detail and also introduce the hardware architecture of combining corresponding compression algorithms to accelerate the Self-Attention Mechanism.

As shown in Table 2, we have classified the compression algorithms of Self-attention and compared the performance of each algorithm. In terms of algorithm classification, I divided the compression algorithms proposed in related papers into Approximation, Network Sparsification, and Tensor Decomposition. First, determine whether the compression algorithm selects the key matrix part of the vector to filter out parameters that have little effect on the output result. If so, it is called approximation. If not, judge whether the parameters or operands of each layer are reduced a lot, which is called Network Sparsification. Then judge again, observe whether the tensor of the Transformer is divided into many smaller sub-tensors or use one tensor to replace many Tensor calculations, which is called Tensor Decomposition.

Table 2. Comparison of the hardware-friendly algorithm for self-attention.

Reference	Algorithm	Dataset	Accuracy Loss	Compression Ratio	BLEU	Year
A^3 [40]	Approximation	SQuADv1.1	1.3%	/	/	2020
ELSA [41]	Approximation	SQuADv1.1	< 1%/2.5%	/	/	2021
Zhang. et al. [42]	Network Sparsification	Multi30K	0%	95%	25.8	2021
Lu et al. [43]	Tensor Decomposition	IWSLT2016	/	/	23.57	2020

3.1. Algorithm

3.1.1. Approximation

In essence, the Self-Attentional mechanism can be viewed as an approximate search of the input sequence. In the past calculation of the Self-Attentional mechanism, the similarity score is usually obtained by calculating the dot product of the key matrix and the query vector. Then the similarity score is normalized to weight by the softmax function, and the weight is weighted and processed with each row in the value matrix. Therefore, it can be seen that the computational complexity of the multiplication between the key matrix and the query matrix varies according to the input sequence length. Longer the input sequence and the time information, the larger the computational complexity and the resources required by the model. And for some other self-attention [1,44,45], models, the situation is even less optimistic because their self-attention calculation changes with the square of the input sequence.

However, the softmax function usually converts the value with a small similarity score is converted to weight by the softmax function, and the weight value is usually approximately close to 0. And these weights close to 0 often do not affect the accuracy of the model and the inference process, but they are important in the model training process, but the hardware accelerators we generally introduce only perform the inference process. So for these weights that are approximately 0, we make it equal to 0 in operation, which can reduce a large part of the calculation amount because they do not have to perform softmax calculation, value matrix weighted, and weighted sum calculation. The approximation algorithm proposed by A^3 [40] can avoid selecting the corresponding vectors in the key matrix and query vertor that do not need the dot product operation. Since the approximation algorithm will anticipate that the weight of these vectors will be close to 0 after the dot product operation is carried out by softmax. Therefore, it can avoid the dot product operation of all rows of the key matrix and the query vertor, and accurately select the most important vector of the key matrix, and the corresponding weight of other unselected vectors is 0. It also preprocesses the key matrix's data without destroying the critical path to reduce the calculation and speed up.

Specifically, the algorithm is divided into two parts, the first part is Greedy Candidate Search, and the second part is Post-scoring Approximation. The first part is more complicated and can be further divided into two small parts, Preprocessing and Iterative Candidate Selection.

Figure 4 shows the Preprocessing stage. In the Preprocessing stage, first sort each column of the key matrix, and store it in the sortedkey register after sorting. After that, when the query vector is ready, the Candidate Search starts. In the first step, if the query[j] of the corresponding column is positive,

max_ptr is assigned to the row index of the data with the largest value in the sortedkey matrix. Instead, max_ptr is set to the row index of the data with the smallest value in the sortedkey matrix. Min_ptr is also set in the same but opposite way. The second step is to set maxQ or minQ. Indexed by max_ptr, the corresponding data in sortedkey is first multiplied by the corresponding query data, and then the result is inserted into maxQ, and its rowID and colID are given together. After Preprocessing, the Iterative Candidate Selection phase begins. First, in the preprocessing stage, we know that max_ptr is the largest in the corresponding entry (min_ptr is the smallest in minQ). If the similarity score for the data corresponding to max_ptr is positive (negative for the min_ptr case), it will be selected as a candidate to add to the greedy_score array. After that max_ptr (and min_ptr) will be updated as it will point to the next largest (post-min) entry. Likewise, the new maxQ (and minQ) corresponding to max_ptr (and min_ptr) will be updated accordingly. This step will be repeated M times (M is a user-defined parameter), and the row with a positive greedy_score will be selected as a candidate row for softmax operation.

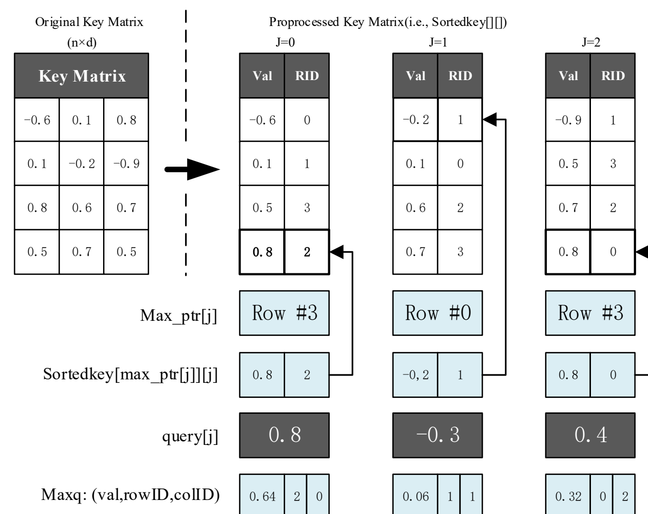

Figure 4. Illustration of the data structures for the efficient greedy search algorithm. minQ operations are omitted for conciseness. Adapted from [40].

After Greedy Candidate Search is executed, Post-scoring Approximation is performed, which directly calculates the dot product of the selected row of the key matrix and the corresponding query vector, and then inputs the result to the softmax function, and the result is the final weight value. After obtaining their dot product results, it is also necessary to take an approximation, that is, to sort their similarity scores, and then output the row with the highest score.

The similar ELSA [41] also proposed an approximation method, and the principle is also to filter out parameters that have little effect on the output result, which can greatly reduce the amount of calculation. First, the design is combined with Sign random projection (SRP) [46], and Efficient Hash Computation proposed by Kronecker Product features [47,48] to estimate the angle between query and key vectors. Secondly, because the dot product is proportional to the cosine value of the angle between the two vectors, use the angle just estimated to approximate the dot product of the query and key vectors. Finally, the dot product result is compared with a threshold to determine whether the selected key is relevant to the query.

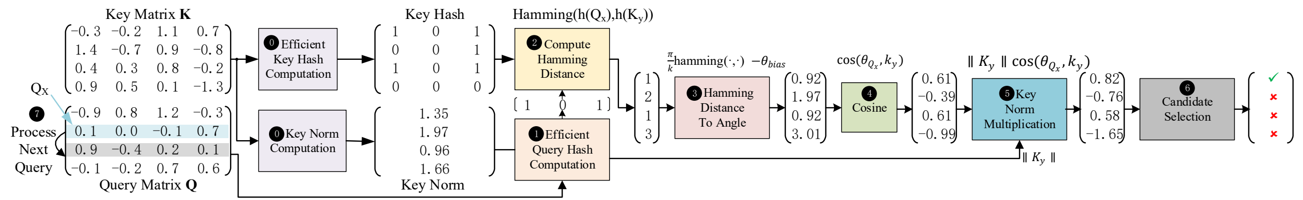

Figure 5. Approximate self-attention algorithm of ELSA. Adapted from [41].

Figure 5 is an approximation algorithm for self-attention. Next, the algorithm steps are described in detail. In preprocessing (step 0), the key is calculated using Efficient Hash Computation, and the norm of the key is calculated accordingly. After that, the approximate dot product of the query and all keys need to be calculated. Specifically, the first step is to calculate Efficient Query Hash Computation to get h(QX) and then calculate the hamming distance of query hash and all keys (i.e., hamming(h(x), h) (y))), hamming distance can be expressed as an unbiased estimator of the angular distance between them. The third step is to convert the hammering distance into the corresponding angle by Eq (4), and it is added θ_{bias} for correction. The angle is then applied to the cosine function and multiplied by the corresponding key norm to estimate the dot product between the normalized query and the key. Finally, it is necessary to check whether these values are relevant to the query by comparing them with thresholds. Because the thresholds between different self-attention layers differ, models like BERT-large have too many layers. Therefore, the design performs inference of the target neural network model on the training set and uses Self-Attention to check each layer's features so that the layer's threshold corresponding to the user-specified degree of approximation can be automatically found.

$$\theta_{x,y} \approx \frac{\pi}{k} \cdot hamming\big(h(x), h(y)\big) \tag{4}$$

It can be seen from Table 2 that ELSA has a smaller accuracy loss than A^3 under the same dataset. This is because the self-attention module of A^3 occupies a larger area, which reduces the parallelism and greatly limits the ability of A^3 to reach the target accuracy on time.

3.1.2. Network sparsification

A smaller model of the Self-Attentional mechanism is usually used to deploy the mechanism on embedded devices with limited memory resources, or a standard large model is compressed to fit the device. Since they can allocate their memory and computational resources flexibly, compressed model weights can be efficiently deployed on CPU/GPU. However, memory allocation and computing kernel cannot be flexibly allocated for FPGAs at runtime. After compressing the weight model, the weight size, height, and width are different, which will result in very low FPGA on-chip memory utilization and computing kernel efficiency. Therefore, Zhang et al. [42] proposed a new structurally pruning method incorporating memory footprint-aware compression.

This pruning algorithm performs two-stage pruning to ensure hardware efficiency after pruning compression, first coarse-grained pruning and then fine-tuning. Coarse-grained pruning is to prune each weight under the same proportion uniformly. The method used is that under all pruning rates, Getindex will get the smallest γ in the LayerNorm layer, where γ can be used as a scale factor to scale up or down a column of the input and γ can reflect the importance of the corresponding column of weight. At the same time, it is assigned to the weight column index, and the pruning rate is gradually

increased from 0%. The mask will shield the weight column in this column index, thus ensuring zero gradient vanishing during training. Finally, when the sparse model accuracy is lower than baseline (obtained at the beginning), the pruning rate reaches an upper limit. The fine-tuning is to prune the remaining γ cross-layer after coarse-grained prunes the masked column and prune the Encoder and Decoder, respectively. The first step is to group the γ of the cross-layer model norm and then store the γ of the corresponding weight after collecting the cross-layer γ type.

3.1.2. Tensor decomposition

The algorithm proposed by Lu et al. [43]divides the weight matrix into the same size matrices and sends them to the systolic array (SA) for operation so that all general matrix-matrix multiplications (GEMMS) can be completed by the SA module, whose size is limited to s × 64. The Tensor shape of the Self-Attention mechanism can be expressed as [Batch size, seq_len, dmodel], while in general, the Self-Attention mechanism Tensor K is always equal to V, and seq_len_q is equal to seq_len_v, so these three Tensor shapes can be represented uniformly by [Batch size, s, dmodel]. Assuming that when the Batch size is 1, the operation between each Tensor can be regarded as a matrix operation. As shown in Table 3, dmodel can be divided into 64 h, so most general matrix-matrix multiplications (GEMMS) can be done with s × 64SA.

Table 3. Variations on transformer and the BERT architecture [43].

Model	d_{model}	h
Transformer-base	512	8
Transformer-big	1024	16
$BERT_{BASE}$	768	12
$BERT_{LARGE}$	1024	16

3.2. Hardware

3.2.1. Accelerators with approximation

First, we will introduce the hardware architecture of the basic Self-Attentional mechanism in A^3 [40] and then introduce the hardware accelerator modules specially designed for the approximation method, namely candidate selection and post-scoring approximate. Figure 6 shows the overall A^3 architecture combined with A^3 Base Design, candidate selection, and post-scoring approximate modules.

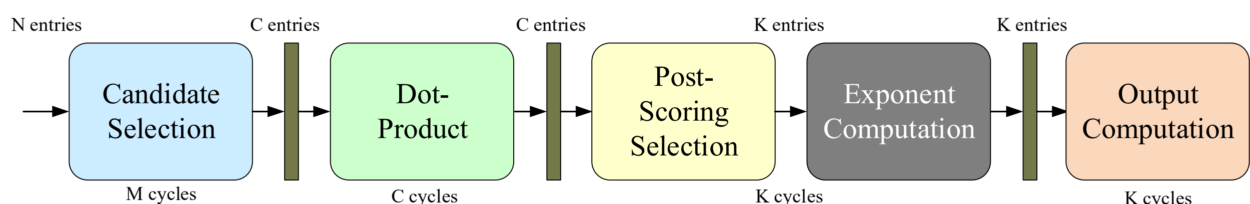

Figure 6. High-level block diagram of the A^3 design with approximation. Adapted from [40].

The A^3 Base Design module can be divided into three sub-modules: Dot-Product, Exponent Computation, and Output Computation. The Dot-Product module will loop out each row of the key matrix and then multiply and accumulate it with the query vector and store the result in the register. In addition, the maximum value in the result vector can be obtained. For the Exponent Computation module, this design does not sacrifice too many hardware resources but adopts the look-up table (LUT) method to prevent the overflow caused by the input fixed-point number being too large. The module first subtracts the maximum value in the input vector from the incoming dot product value so that all inputs will be less than or equal to 0, and the exponent of the corresponding input must also be less than or equal to 1. This does not cause errors because the softmax function simultaneously adds (or subtracts) the same number to the input, and the output is unchanged. In order to reduce the size of the LUT, this module decomposes an exponent operation into the multiplication of two exponents. The Output Computation module divides each element through the exponential outputting module by the sum of all output elements to complete the normalization operation. Then multiply the result by the value matrix to get the final output.

Figure 7 is a detailed diagram of the Candidate Selection Module. The key matrix is pre-stored in sram after sorting, and the row index of the corresponding data is also stored in sram. This module includes two d registers for max_ptr and min_ptr, two multipliers, two sets of multiplication buffers, two comparison trees, and a greedy score register. First, use max_ptr (and min_ptr) as an index to find the corresponding value from the sorted key matrix and Query vector in sram, send it to the multiplier for multiplication, and then enter the multiplication buffer to wait for a column of data to be full, and send all the data in the multiplication buffer to the comparison tree for comparison. Thus, the max (and min) value is obtained, and then its row index is sent back to max_ptr (and min_ptr) for updating. At the same time, the dot product result is also input to the greedy score register, and the Candidate Vector is output after the sorted key matrix is all calculated.

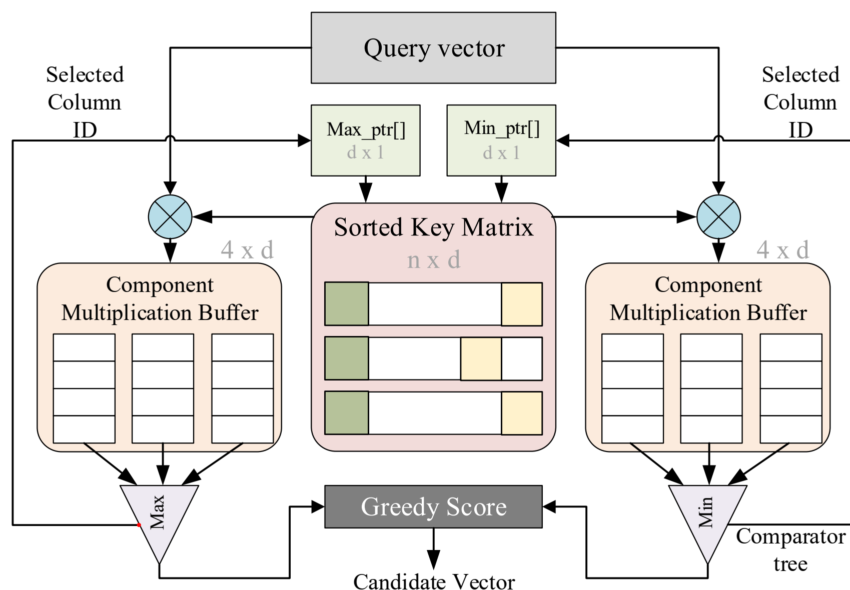

Figure 7. Simplified block diagram of the A^3 candidate selection module. Adapted from [40].

As shown in Figure 6, the Post-scoring Approximate Module hardware module is placed in front of the index calculation module. The main function of this module is to select a row with the largest

dot product value among all the candidate rows after the candidate selection module. Therefore, it needs to calculate the difference between the maximum dot product value and the remaining rows. Specifically, when the difference between a data being compared and the selected maximum dot product value is greater than a preset threshold, the compared data can pass through the module to reach the index calculation module.

Figure 8 shows the ELSA accelerator's data flow and pipeline situation [41]. The input of the ELSA accelerator only includes the query matrix, key matrix, and value matrix. After the query and key matrix are fetched from their respective memory and ready, the preprocessing stage starts first. In this stage, the Hash computation module calculates the corresponding hash values of all rows of the key matrix and calculates and stores it in Key hash memory after completion. Then, the key norm is calculated by the norm computation module and stored in the key norm memory. Then calculate the hash value of each row of the query matrix. Candidate Selection will always receive query hash, key hash and key norm as input and then calculate the hamming distance between key hash value and query hash value by XOR and adder. Then access the filled LUT with the obtained hamming distance and the LUT stores $\cos(\pi/k \cdot d_{Hamming} - \theta_{bias})$. After the corresponding value is retrieved, it is multiplied by the key norm to get the approximate similarity score, and then the approximate value is compared with the threshold. If it is greater than the threshold, it is selected, and the key index is passed to the queue. Multiple candidate selection modules are executed in parallel, and their outputs are sent to the arbit module and then passed to the self-attention computation for computation after arbitration. After the Attention computation module receives the input key index of the arbit module, it uses the multiplier and adder tree to calculate the dot product of the corresponding key and query. For softmax, the design uses the LUT method to calculate the exponent of the value, and the sumexp register accumulates a low of the exponent components, and the sum is sent to the output div module as the output of the self-attention computation module. After calculating the index of the corresponding dot product value, multiply and accumulate it with the corresponding value. When the calculation of all the corresponding selected keys of the current query is completed, the obtained output vector and sumexp are passed to the output div module. The output div module divides all components output by the self-attention computation module corresponding softmax function completed by sumexp. Attention computation and output div modules are fully pipelined, and output div modules can be parallelized with other modules.

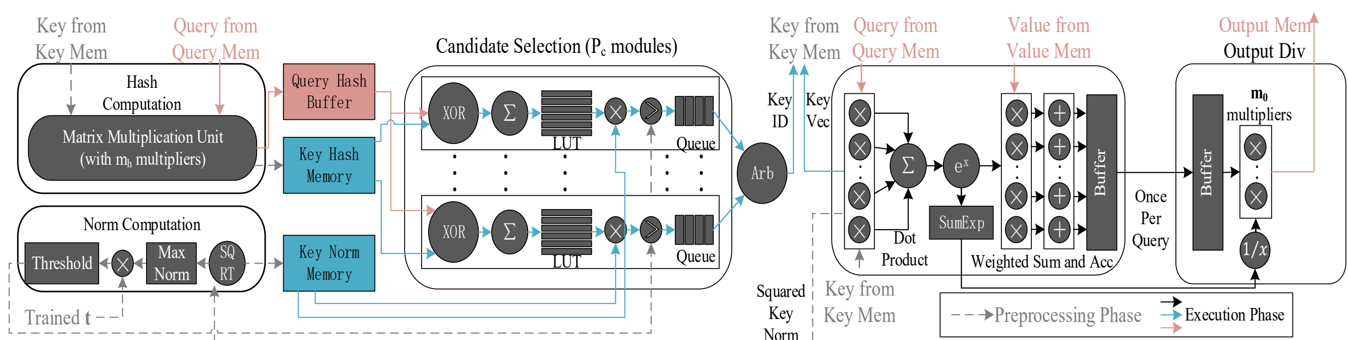

Figure 8. ELSA pipeline block diagram [41].

Table 4 compares the chip area and power of A^3 and ELSA deployed on ASIC. ELSA is better than A^3 on Area, which indirectly proves that the approximation scheme proposed in ELSA better

reflects hardware friendliness.

Table 4. Comparison of accelerators with approximation.

Reference	Algorithm	Area (mm^2)	Dynamic Power (mW)	Static Power (mW)	Target	Year
A^3 [40]	Approximation	2.082	98.92	11.502	ASIC	2020
ELSA [41]	Approximation	1.255	956.05	13.31	ASIC	2021

3.2.2. Accelerators with network sparsification

The Network Sparsification of the hardware architecture joint algorithm proposed by Zhang et al. [42] is shown in Figure 9 for its system hardware architecture diagram. This design considers that the memory resources of the FPGA still cannot bear compressed model size. Therefore, it is necessary to cooperate in design modules such as data calculation, exchange, and scheduling.

The input, weight, and output data in Figure 9 are the corresponding buffer areas. In front of the DDR interface, there will be a data bus with a ping-pong buffer function to support continuous data processing. The processing element (PE) schedule register is used to store data acquisition information and execution times so that one output column and the entire output column can be obtained. Among them, PE has two types, multiplication and addition. The two multiplications are packaged into the DSP and then accumulated through the addition PE, and the multiplication PE is used for simultaneous processing, and the addition PE accumulates the tree structure of the two vectors for operation. Read the corresponding elements from the PE schedule register, input buffer, and weight buffer and perform the calculation, thereby outputting all column elements. And the accelerator can be called recursively to perform the matrix multiplication operation in the Transformer.

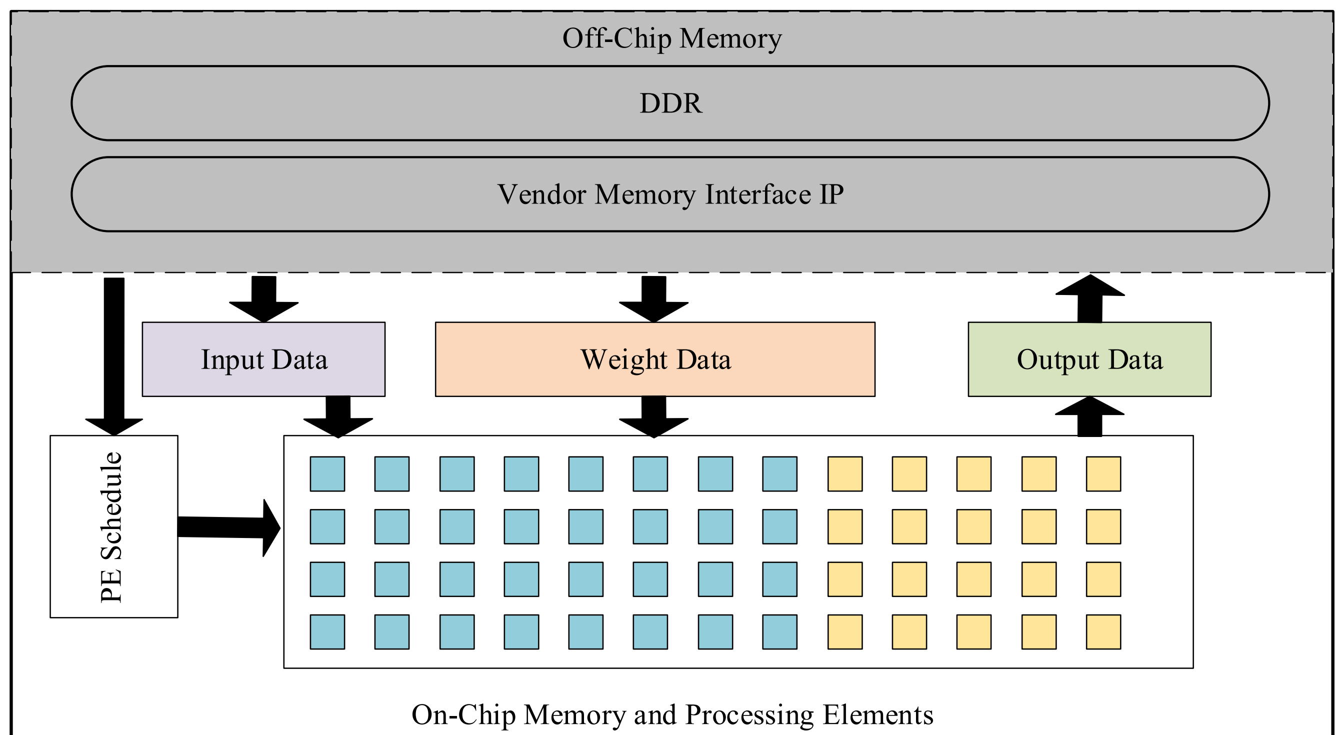

Figure 9. Attention mechanism accelerator with Network Sparsification architecture overview [42].

3.2.3. Accelerators with tensor decomposition

The hardware design proposed by Lu et al. [43] combined the weight matrix into several sub-matrices adapted to the SA to accelerate the Self-Attentional mechanism, as shown in Figure 10, this hardware also designed layer-norm, but this section only introduces the hardware architecture of self-attention. The s ×64 SA module outputs column by column, Temp1 and Temp2 are intermediate registers, and P is a self-attention mechanism output structure accumulator. Because the weight matrix is divided into h parts, it needs to be executed in a loop for h times. Figure 10 is the overall architecture diagram of the hardware. Next, introduce its data flow. The sub-matrix multiplication is performed in the SA module. First, there will be a multiplexer after Q, K, V, Temp1, and P to control whether they can pass through and then input to the SA module for multiplication. The S adder adds bias from the Bias Memory, and then the results of Q and K will be sent to Temp1 and Temp2 in turn. Each of them is then input to the SA module again through the multiplexer for operation. The result is used as the input of the Softmax module and sent to the Temp1 register. After V is calculated by the SA and S adders, it is stored in Temp2, and then Temp1 and Temp2 enter the SA module together for operation and output to the P register, and the final output is obtained after h times of accumulation. From this point of view, the SA module has the highest complexity and does not stop running until the Layer-Norm module starts. The complexity second to SA in this design should be the softmax module. Similarly, softmax's exponentiation and division operations are very critical to the hardware. The article by Wang et al. [49] is referenced, log sum-exp is used to express the technique [50], and an algorithmic strategy for the reduction index and logarithmic function is designed. This module also uses the log-sum-exp trick to avoid calculations such as division. As seen from Eq (5), the softmax module is constructed by transforming exponential and logarithmic functions without using regular multipliers and LUT. Table 5 shows the performance comparison of accelerators with Structurally Pruning and Tensor Decomposition.

$$Softmax(X_i) = \frac{exp(X_i - X_{max})}{\sum_{j=1}^{d_k} exp(X_j - X_{max})}$$

$$= exp\ (X_i - X_{max} - ln\ (\textstyle\sum_{j=1}^{d_k} exp\ (X_j - X_{max}))) \qquad (5)$$

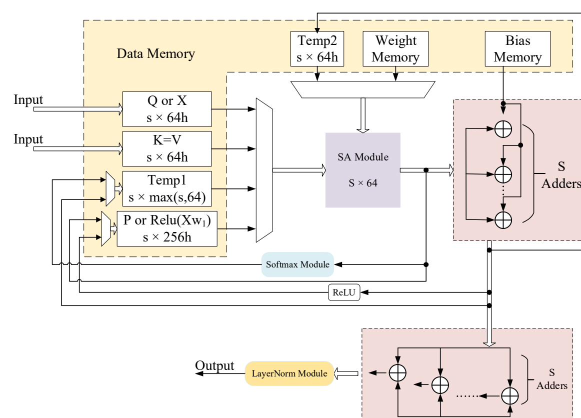

Figure 10. The top-level architecture of accelerator with partitioning matrices. Adapted from [43].

Table 5. Comparison of accelerators with structurally pruning and tensor decomposition.

Reference	Algorithm	Latency	Throughput	Target	Year
Zhang et al. [42]	Structurally Pruning	8.4 ms	2.04Gop	FPGA	2021
Lu et al. [38]	Tensor Decomposition	8.9 ms	/	FPGA	2020

4. Transformer

In this section, we introduce the classification method of the transformer compression algorithm, the hardware-friendly algorithm used on the transformer hardware, and the hardware architecture that combines the above corresponding compression algorithms to accelerate the Transformer.

For the classification method of the transformer compression algorithm, determine whether the Tensor of the Transformer is divided into many smaller sub-Tensors first, or one Tensor can replace many Tensor calculations, called Tensor Decomposition. If not, judge whether the model parameter bit width is reduced, which is represented as Data Quantization. Then judge again to see if the parameters or operands of each layer are reduced a lot, which is called Network Sparsification. Finally, an efficient and suitable model method Neural Architecture Search (NAS) is searched through a large search space [51].

4.1. Algorithm

4.1.1. Tensor decomposition

The algorithm proposed by Li et al. [39] references the Block-circulant Matrix (BCM) compression algorithm and proposes the Enhanced Block-Circulant Matrix model compression algorithm. While CirCNN [52] and C-LSTM [53] adopt the BCM algorithm for image classification and language recognition, respectively, both significantly improve performance. According to CirCNN, C-LSTM did not research large-scale language representation and wanted to maintain the prediction accuracy further. Based on this incentive, an Enhanced Block-Circulant Matrix model compression algorithm was proposed with a larger compression ratio and less accuracy loss. Specifically, the original weight matrix is replaced by one or more circulant matrix blocks to reduce storage, and the input is divided accordingly. For previous compression using BCM, they only indexed the first row/column as an index vector, i.e., only the first row/column was stored and computed. And Zhao et al. [54] also derived the theoretical basis, demonstrating their effectiveness, but they lacked efficient representations for other rows/columns. Based on the model compression of the designed Enhanced BCM, the formula of the exponential vector is modified to Equation 6. Use b to denote the row/column size of each circulant matrix. In terms of matrix-vector multiplication, the algorithm adopts the fast Fourier transformer (FFT), which is based on the Cyclic convolution theorem [55,56]. BCM-based matrix-vector multiplication $W_{ij}X_j = P_{ij} \circledast X_j = IFFT(FFT(P_{ij}) \circ FFT(X_j))$, where $\circledast$ represents circular convolution and $\circ$ is element-level multiplication.

$$P_{ij} = \begin{bmatrix} \frac{1}{b}\sum_{j=1}^{b} W_{1j} \\ \frac{1}{b}\sum_{j=1}^{b} W_{2j} \\ \cdots \\ \frac{1}{b}\sum_{j=1}^{b} W_{bj} \end{bmatrix} \tag{6}$$

4.1.2. Data quantization

Quantization is one of the common and important methods for model compression. Quantization can reduce the bit width of the parameter data of the transformer model and retain the original structurally system, thereby reducing its huge computational complexity and memory consumption. The Quantization bit width and precision loss have become the distinguishing criteria of the Quantization algorithm. The Quantization method proposed by Liu et al. [57] fully quantizes BERT, including weight, activation, Softmax, layer normalization, and all intermediate results, so that computational complexity and memory issues can be better optimized. This algorithm is also friendly to hardware, and it quantizes all data to integer or fixed-point data, 8/4 bit and 8/8 bit multiplication when used by the hardware. The quantization of weight and activation adopts a hardware-friendly symmetric linear strategy. However, for the quantization of bias, the bias is quantized into a 32-bit integer using the scale factor of quantization weight and activation, which can facilitate its deployment on hardware. The quantization of other parameters such as softmax and layer normalization is 8 bits. However, under what circumstances can the Quantization accuracy be the best, and the previous work was to compare it by yourself. The algorithm of quantized Vision Transformers (ViTs) proposed by Sun et al. [58] realizes the automation framework and only needs to give the model structure and the required frame rate to automatically output the required quantization accuracy of activation. Figure 11 illustrates VAQF that builds an FPGA-based ViT inference accelerator. The ViT structure and desired frame rate (target FPS) are provided as input information. A compilation step is conducted to decide the required precision for activations with the accelerator settings to satisfy the FPS target, when the weights are binary. Specifically, it introduces a binarization method, using binary precision for weight and low precision for activation to achieve a tradeoff between efficiency and loss of precision. This binarization method is different from Binary-BERT [30], which directly applies the 1-bit convolution [59,60] method for binary weight quantization. The binary weight quantization according to the definition of 1-bit convolution (see Eq (7)), the two-mechanism weight W_b matrix is obtained from the given real number W_r, where ω_b and ω_r are a specific element of the matrices W_b and W_r, respectively, and $\frac{\|W_r\|_{l1}}{n}$ is the scale factor that minimizes the difference between the binary value and the actual weight value.

$$\omega_b = \frac{\|W_r\|_{l1}}{n} Sign(\omega_r) = \begin{cases} +\frac{\|W_r\|_{l1}}{n}, & if\ \omega_r > 0 \\ -\frac{\|W_r^n\|_{l1}}{n}, & if\ \omega_r \le 0 \end{cases} \tag{7}$$

Table 6 summarizes the data quantization methods introduced above, mainly compared in terms of data bit width, compression ratio, and precision loss. In terms of Accuarcy Loss, the Fully-Quantization proposed by Liu et al. [57] is smaller. It may be because, in the weight Quantization part, Sun et al. [58] chose 1 bit for the Quantization bit width.

Table 6. Comparison of the algorithm of data quantization.

Reference	Dataset	Weight Bits	Activation Bits	Accuracy Loss	Compression Ratio	Year
Liu et al. [57]	SST-2	4 bit	8 bit	0.81%	7.94 ×	2021
Sun et al. [58]	ImageNet-1K	1 bit	8 bit	4%	/	2022

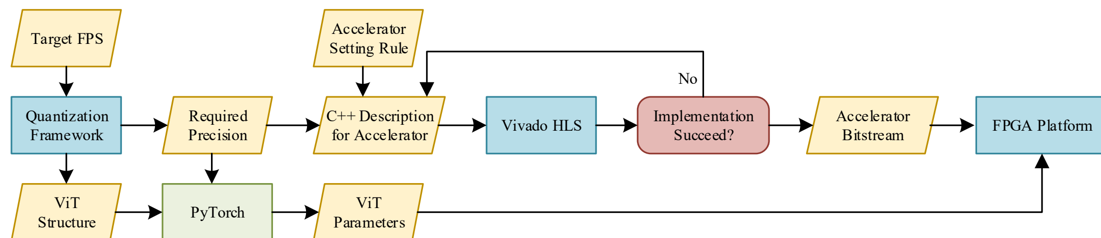

Figure 11. Overall flow of VAQF.

4.1.3. Network sparsification

The difference between Network Sparsification and Quantization is that Quantization reduces the bit width of the corresponding parameters of each layer to simplify the algorithm itself, while sparsification reduces the number of operands, which can greatly reduce the amount of memory and calculation necessary to achieve the acceleration. Deep learning models often employ methods such as irregular pruning [61], structured pruning [62], and pattern pruning [63]. Naturally, sparse operations also entail irregular memory accesses and non-zero element index overheads.

Qi et al. [63] proposed a hardware-friendly Hierarchical Pruning (HP) algorithm, which combines block structured pruning (BP) [64] and vector-wise pruning (VW) [65] methods to propose this hierarchical pruning Hierarchical Pruning algorithm. Specifically, the BP model is used as the backbone model of HP, so the first coarse-grained pruning is performed with the BP algorithm, which mainly prunes some unimportant columns of the weight matrix divided into blocks. A second fine-grained pruning is then performed using the VW algorithm to prune the unimportant parameters from the first unpruned columns, maintaining balance by pruning the same number of parameters in each block.

The part of the pruning also affects the accuracy of the model, and weight pruning reduces the number of weights and speeds up the Transformer. Peng et al. [66] proposed column-balanced block pruning for transformers. Similarly, column balance is achieved by trimming the weight matrix by an equal number of elements across each column.

Firstly, divide the weight matrix into sub-matrices of the same size, calculate the L2 norm of each sub-matrix block, and discard the sub-matrix with the smallest L2 norm. After all pruning, the obtained sub-matrices are horizontally connected to form the trimmed matrix.

Qi et al. [66] proposed a block balanced pruning using structurally pruning because of the strong regularity of structural pruning [64,67,68], which is friendly to hardware. Ma et al. [63] compared the accuracy of block-balanced pruning (BBP) [69] and block-wise pruning (BW) [68] under different sparsity ratios and finally chose BBP as the compression algorithm. Because the pruning method of BBP is more fine-grained, it will not prune the whole block like BW, which is likely to delete some important information from the model.

Our comparison of the pruning methods introduced above is shown in Table 7, comparing each method's compression ratio and accuracy. Generally, the tradeoff between sparsity ratio, structure, and accuracy establishes the essence of each pruning algorithm. Based on the above algorithm, block-level or structural pruning is performed according to the model structure, and a balance is achieved by pruning the same number of parameters to ensure a sufficient hardware compatibility level.

Table 7. Comparison of algorithm of network sparsification.

Reference	Dataset	Model	Compression Ratio	Accuracy	Year
Qi et al. [69]	WikiText-2	Transformer	89.85%	96.13%	2021
Peng et al. [66]	WikiText-2	Transformer	90%	95.12%	2021
Qi et al. [70]	WikiText-2	Transformer	80–90%	/	2021

4.1.4. Neural architecture search

In addition to the above compression methods, Neural Architecture Search is also an effective method for determining the most appropriate transformer model, such as Evolved Transformer [71]. However, the search process is prohibitively expensive, and the hardware will be more expensive. Wang et al. [72] applied a weight-sharing supernet to identify efficient models. The Hardware-Aware Transformer (HAT) framework is proposed and utilizes Neural Architecture Search (NAS) to discover suitable and efficient methods for deployment on target hardware. Its hardware-aware neural architecture search framework is illustrated in Figure 12. As a beginning, it is necessary to construct a large design space to identify as many encoders, decoders, and heterogeneous layers as possible. The randomly selected sub-transformers are then iteratively optimized to train a supernet-supertransformer that can be shared with weights. The supernet-supertransformer can provide a performance proxy to the subtransformer. In this step, the hardware platform is brought into play, which will collect a dataset with the subtransformers and measure delays, and train the delay predictor to obtain fast and accurate delay times. Finally, a search is carried out based on time constraints to identify an efficient input model for the hardware platform. As soon as the model is obtained, it is trained from scratch until the final performance is obtained.

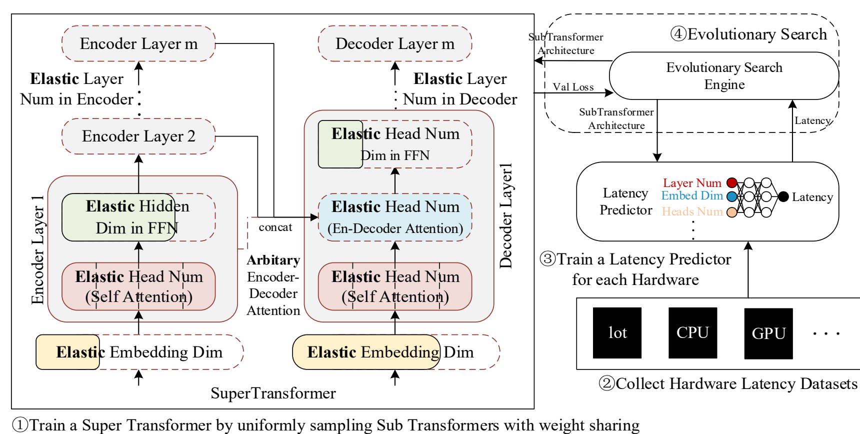

Figure 12. Hardware-Aware Transformer(HAT) overview [72].

4.2. Hardware

4.2.1. Accelerators with tensor decomposition

The hardware architecture of Li et al. [39] combined the Enhanced BCM compression algorithm to accelerate the Transformer. Although the transformer model has been model compressed, the entire model size is still too large for the memory resources of the FPGA, and the transformer parameters cannot be satisfied by the on-chip resources of the FPGA. Therefore, in FTRANS, the model is divided into Embedded layer and Encoder/Decoder stack, where Embedded layer is designed as a LUT for converting input sentence discrete tokens into continuous space. In order to avoid the consumption caused by frequent access to off-chip weight, the Embedded layer is offloaded to off-chip memory, and the FPGA on-chip resources are exclusively used for Encoder and Decoder stack calculations. Then, FTRANS performed inter-layer coarse-grained pipelines, intra-layer fine-grained pipelines, and computational scheduling to alleviate I/O constraints.

4.2.2. Accelerators with data quantization

In general, when the Quantization data bit width is larger than 8 bits, the loss of precision is negligible. However, obtaining Quantization with a lower bit width is possible in pursuit of higher performance with an acceptable loss of precision in the Quantization field. In the deep learning model, there are usually two types of Quantization, fixed bit width Quantization and variable bit width Quantization.

Liu et al. [57] proposed fully quantizing the BERT (FQ-BERT) to introduce variable bit width. Since different layers of BERT have different bit widths, a reconfigurable module design with variable bit width is adopted. A bit-level reconfigurable multiplication accelerator is proposed, which can support 8/4-bit and 8/8-bit multiplication. As shown in Figure 13, adopting a method similar to Bit-fusion [73], a Bit-split Inner-product Module (BIM) is designed for demultiplexing the same module to support operations of different bit widths. Each BIM contains multipliers, adder trees, and corresponding shift adders, and each multiplier has a signed signal to represent the difference between signed and unsigned numbers. The function of the shift-add logic is to shift the partial sum when the input bit width is greater than 4 bits. There are two ways to add the position of the shift logic. Using TypeA can save more resources, but the input data needs to be arranged in order. In a Processing Elements (PE) of FQ-BERT, the output of the BIM is sent to an accumulator to be added to the previous data, and then the result is input into PsumBuf. Then PsumBuf is input to the Quantization module, so that the final value including the bias and scale factor is obtained.

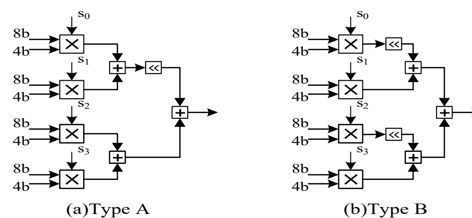

Figure 13. Two types of BIM. Adapted from [57].

For the introduction of fixed bit width, Sun et al. [58] quantized the weight into two bits, activated low-precision Quantization and accelerated ViT based on this algorithm. It uses loop-tile technology to divide the input, weight, and output of each layer of ViT into tiles to reduce the memory and computing resources of FPGA. Figure 14(a) shows the tiles of the unquantized layer. At the module computing level, the multi-layer perceptron (MLP), the FC layer of the multi-head attention module, and the self-attention module in ViT are computationally complex and intensive. For the multi-head attention module, it must be used multiple times to achieve parallel processing, but the FC layer calculation only performs one matrix multiplication. In order to be compatible with the FC layer at the same time, the input of the FC layer is divided into a parallel number of multi-head attention modules. The output of each segment is then added together when it is output. Figure 14(b) shows the calculation flow of these two layers. The computation is made more efficient by unrolling and piping the L2, L3, and L4 loops under L1. Figure 14(c) shows a flow of whether it is a vit layer of Quantization, when no Quantization is performed the DSP resources on the FPGA are occupied, and the Quantization is replaced by the corresponding addition and subtraction of the LUT. Because the weight is binarized, the weight value is +1 or -1. In addition to the above contributions of this design, it is worth noting that it implements automated operations and only needs to give the model structure and the required frame rate to automatically output the Quantization accuracy required for activation.

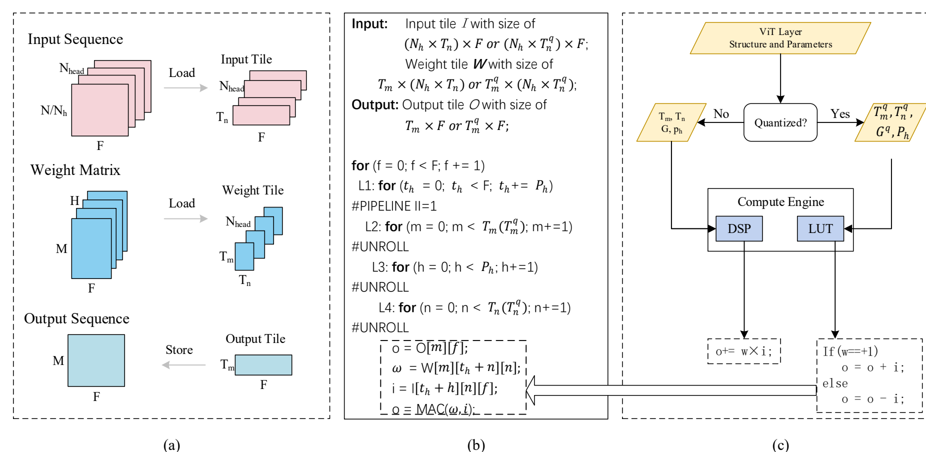

Figure 14. Detailed implementation of ViT accelerator. (a) Loop tiling of input, weight, and output for one model layer; (b) Computation flow in compute engine with loop tiling, pipelining, and unrolling; (c) Processing flow of one model layer based on whether it is quantized or not. The superscript q represents the parameter after Quantization. Adapted from [58].

Table 8 compares the Throughout, Latency, and FPS/W of each accelerator under the corresponding bit width. The balance between model accuracy and performance should also be considered. INT2 or INT4 can achieve a good balance in small-sized datasets, and quantized INT4 is the best choice for large datasets.

Table 8. Comparison of accelerators with data quantization.

Reference	Weight Bits	Activation Bits	Latency	Throughout	FPS/W	Target	Year
Liu et al. [57]	4 bit	8 bit	43.89 ms	/	2.32	FPGA	2021
Sun et al. [58]	1 bit	8 bit	/	861.2GOPS	2.85	FPGA	2022

4.2.3. Accelerators with network sparsification

After the pruning algorithm processes the matrix parameters, they will become sparse matrices. That is, the number of non-zero elements (NZ) is much smaller than the number of zero elements. In order to skip the computation of a large number of 0 elements in sparse matrices, various sparse matrix storage formats have been proposed to compress sparse matrices to reduce computational and storage resources, and the typical formats are such as COO [74], CSR [75], BCSR [75], and MBR [76], etc.

Qi et al. [69] proposed a bitmap format similar to MBR to cooperate with the HP algorithm, which can further reduce memory usage. The corresponding format is selected according to whether the sparse ratio is 0, consisting of the number of blocks, the column index of the block, and three sets of arrays with non-zero elements in the column. In contrast, the other format lacks the column index array. Its accelerator design is different from the previously proposed accelerators [65,77–79], and their methods cannot be applied to transformers. In order to solve the challenge of random reading and writing, the design changes the memory access mode. By multiplying the multi-line of the input matrix by one column of the weight matrix to achieve sequential writing, the matrix problem of random writing results can be solved so that the parallelism between rows can be used to calculate the dot product and corresponding column of the weight matrix of the input matrix at the same time. An input matrix row buffer IRB is allocated to buffer each row of the multiplied input matrix, which is implemented by a randomly accessible register.

Peng et al. [66] also proposed the compressed sparse column block (CSCB) compressed sparse matrix format based on their Column Balanced Block Pruning algorithm. This compression format is formed by combining the CSB format and the BCSR format, which ensures that only one index address is needed for each sub-matrix block, so it has lower memory consumption than CSB and BCSR formats. Figure 15 shows the overall architecture of the design. Because the Embedded layer is usually designed as a LUT, it is used in external memory to save on-chip memory. The data flow is: first, the PC sends the generated tokenized sentence to the FPGA through PCIe, and then the DDR controller obtains the embedding information of each word in the DRAM, and transmits the word input sequence to the on-chip memory of the FPGA for use by Encoder and Decoder.

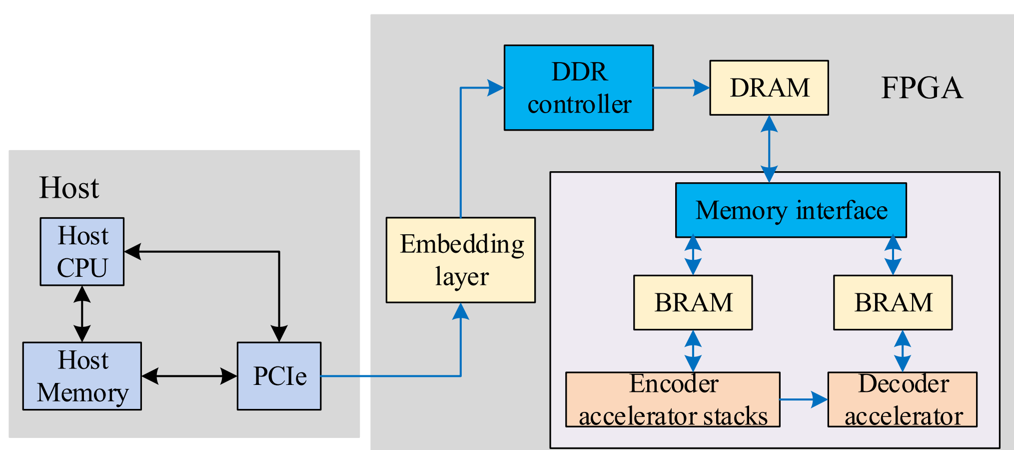

Figure 15. Overall structure of the [66].

The sparse matrix multiplication operation occupies most of the computation during the transformer inference process, and this accelerator delves into the sparse matrix multiplication

accelerator. This sparse matrix multiplication accelerator is combined with the Column Balanced Block Pruning algorithm and CSCB compression method, and a dedicated pipeline is designed by exploiting the parallelism between PEs and within PEs. Store the compressed sparse weight matrix and its index matrix in BRAM, and extract an element in the vector and a data block in the compressed sparse matrix from the index value of the index matrix, respectively. And perform general dense matrix multiplication within a PE, after which the result is sent to the accumulation module for accumulation, which will traverse all block columns of the sparse matrix to obtain the final dot product output.

Qi et al. [70]proposed Compressed Block Row(CBR) for Block balance pruning that proposed by themselves. It consists of a three-dimensional array to store the non-zero elements of the sparse weight matrix and a two-dimensional array to store the internal block index of the non-zero sub-rows, so it can significantly reduce memory usage and take advantage of the block balanced nature to save decoding overhead. Figure 16 shows the detailed data flow of the transformer accelerator proposed by this accelerator. It consists of FPGA on-chip memory to establish weight buffers, intermediate result buffers, and five calculation modules. Among them, the five calculation modules are Multi-head attention, add_norm, feed-forward connected, and decoder. These five modules run in the order in the figure due to the dependency between the data. Similarly, the dot product operation of the Multi-head attention module needs to be run in parallel to improve computational efficiency. The pipeline method is used for each layer of operations to provide transportation efficiency. In terms of saving memory, it reuses intermediate result buffers multiple times per layer. Also, because sine and cosine operations are used in embedding and positional encoding, it is very computationally expensive for FPGA and cannot take advantage of its parallelism. Therefore, operations are performed on the CPU side, and then input to the FPGA side through the PCIe interface.

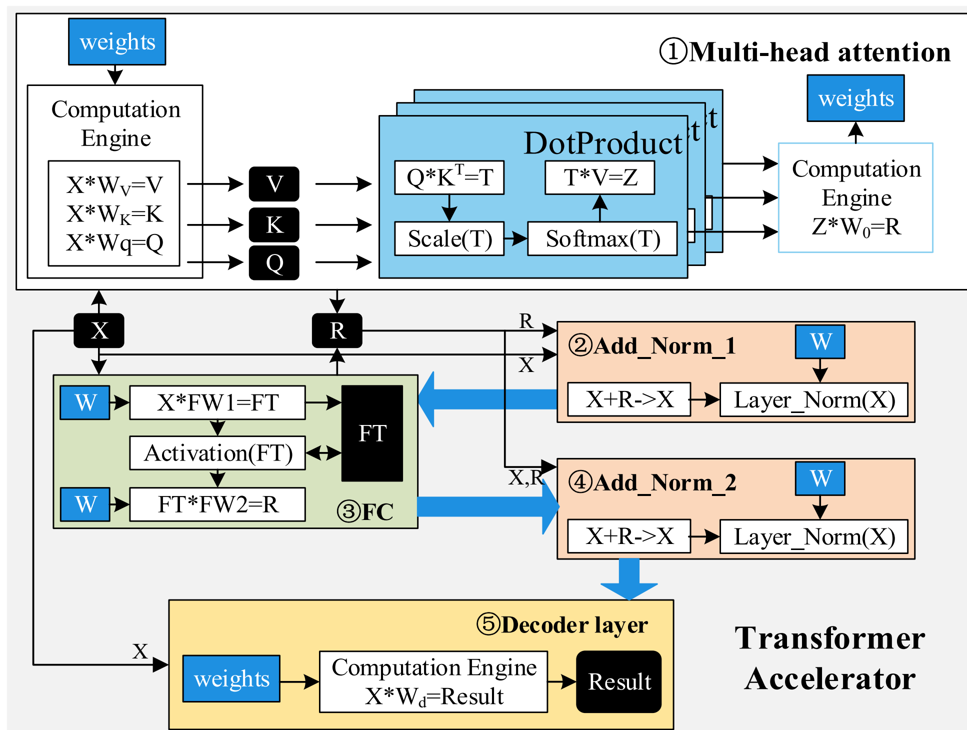

Figure 16. The detailed computation dataflow of [70].

Table 9. Comparison of compression format.

Reference	Compression Format	Contribution	Target	Year
Qi et al. [52]	MBR	Change Memory access	FPGA	2021
Peng et al. [55]	CSCB	Improvement of CSB and BCSR	FPGA	2021
Qi et al. [56]	CBR	Take advantage of the balanced properties of the block	FPGA	2021

Table 9 summarizes the compression format of each accelerator with Network Sparsification and innovation of their compression format. Table 10 summarizes and compares each accelerator with Network Sparsification, mainly to collect and compare the Latency and Throughout parameters of each accelerator. It can be seen that the accelerator proposed by Qi et al. [69] yields better performance. In addition, it can be seen that applying hardware-friendly compression algorithms can eliminate the overhead caused by irregular memory access on the hardware.

Table 10. Comparison of accelerators with network sparsification.

Reference	Model	Latency	Throughout	Target	Year
Qi et al. [69]	Transformer	6.45 ms	14.14GFLOPS	FPGA	2021
Peng et al. [66]	Transformer	10.35 ms	3091.8FPS	FPGA	2021
Qi et al. [70]	Transformer	7.85 ms	0.1136GFLOPS	FPGA	2021

4.2.4. Accelerators with neural architecture search

In terms of hardware, Wang et al. [72] designed an accelerator named SpAtten to co-process the self-attention layer in computational NLP. The accelerator can support novel token pruning to reduce the memory access and computation of the self-attention layer. The entire accelerator module is completely pipelined, and each module corresponds to each operation, which will not cause a lot of data movement and greatly reduce the data movement overhead. The attention layer input is stored in High Bandwidth Memory (HBM). Because there will be random access problems after Token pruning, it uses a crossbar to deal with address conflicts so that HBM channels are kept busy, and bandwidth utilization is also increased accordingly. Then, the top-k engine has a high degree of parallelism and can support dynamic pruning of tokens with certain time complexity. For the dynamic low-precision implementation, an on-chip bit-width converter is used to deal with the segmentation of the obtained bits and the connection of msb and lsb.

The detailed diagram of the SpAtten accelerator architecture is shown in Figure 17. For the attention calculation of each query, the top-k module first handles the accumulation of important scores, and the obtained K indices are input into the fetcher. Fetcher calculates the address of K and then inputs it into the crossbar, and then the crossbar transmits the relevant data and performs dot product operation and softmax calculation to obtain the attention probability. Then, the probability is provided to the dynamic precision determination module and the importance score accumulator, which play the roles of whether lsb is required and the execution of accumulation, respectively. After that, the local token is calculated by top-k for local token pruning and the k index of the corresponding V is sent to the fetcher. Finally, the attention output of the residual probability and the corresponding V multiplied is obtained. From Table 11, VAQF gets the biggest accuracy loss, [42] gets the smallest accuracy loss, and ELSA gets the smallest average accuracy loss.

Table 11. Comparison of accuracy loss of accelerators.

	A^3 [40]	ELSA [41]	Zhang. et al. [42]	Lu et. al. [43]	FQ-BERT [57]	VAQF [58]	Qi et al. [69]	Peng et al. [66]	Qi et al. [70]
Max	0.826%	0.964%	0%	/	3.08%	4%	1.37%	0.31%	/
Min	0.826%	0.819%	0%	/	0.81%	4%	1.37%	0.31%	/
Average	0.826%	0.898%	0%	/	1.94%	4%	1.37%	0.31%	/

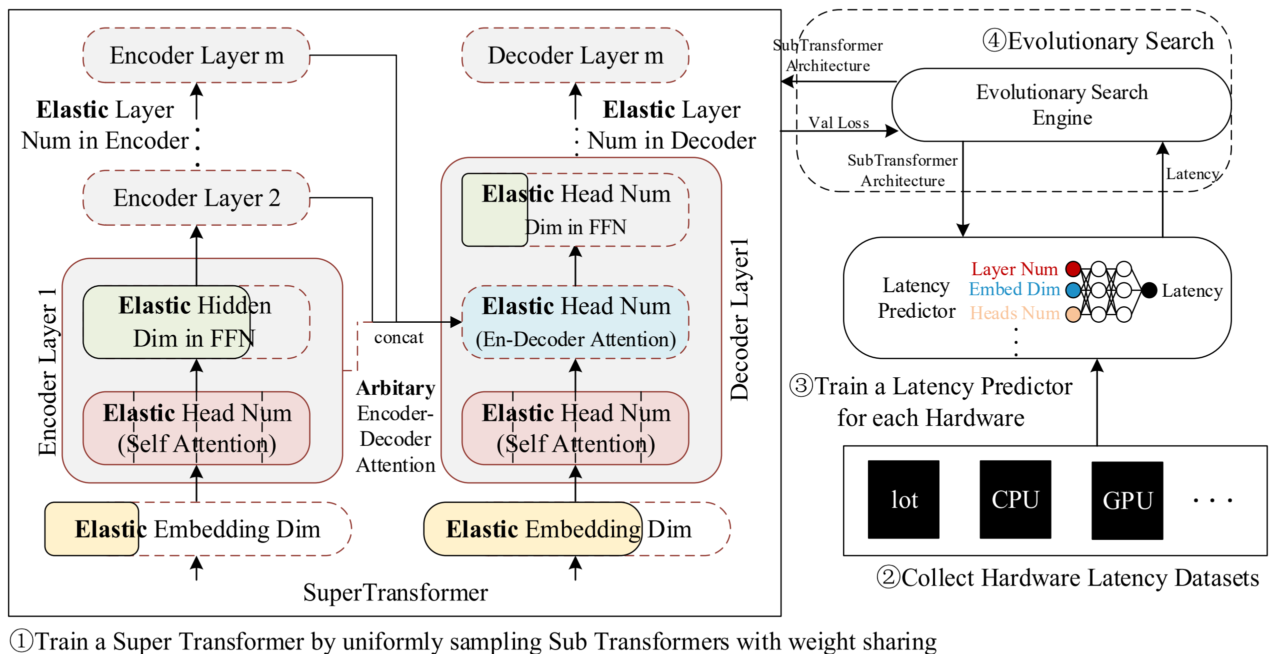

Figure 17. SpAtten architecture overview. All of the modules are fully pipelined to achieve high performance [72].

5. Conclusions and discussion

5.1. Conclusions

Transformators play an increasingly important role in NLP, and their performance is becoming increasingly powerful. The powerful performance of the Transformer is a consequence of its size and the surge in computing power. At the same time, an efficient and low-power consumption hardware platform is needed to support the trend of electronic devices being deployed to mobile and embedded devices, as well as the limitation of resources and computing capabilities of mobile devices. This paper provides a comprehensive and detailed review of the compression and hardware acceleration of Self-Attention and Transformer from the perspective of hardware-friendly algorithms and hardware deployment. As far as Self-Attention is concerned, we have discussed all the hardware architectures in recent years, and we have discussed its data flow analysis and architecture analysis in detail. The following are the four compression algorithms we propose for the Transformer: Tensor Decomposition, Data Quantization, Network Sparsification, and Neural Architecture Search, as well as the respective hardware-friendly compression algorithm and the Transformer's hardware architecture when combined

with the above algorithms.

5.2. Discussion

The application scenarios of Transformer appear more in mobile devices with demanding edge computing requirements, such as real-time translation using smart devices, text processing, etc. Existing research is more on the deployment and implementation of high performance computing devices, and few types of research directly compare and optimize with mobile devices such as ARM and Jetson. We believe that the iteration of edge devices such as Zynq, Pynq, and MPSoC provides the possibility of high-performance edge realization for related research. The related challenges lie in three points: 1) Further compression and Quantization of the model to adapt to the limited resources of edge devices; 2) Efficient memory storage format and access strategy to improve resource utilization; 3) More efficient architecture to reduce the system's power consumption. Based on the above research, it can effectively reduce data network communication's time and resource consumption and bring users a better offline real-time experience.

Further research at the terminal is also required. With the development of technology, more and more hardware devices can be deployed to run Transformer after further updating, such as FPGA (ZCU102, Alveo200, etc.). Models with different sizes or constraints can be deployed to many devices with different hardware resources. However, in the past work, the designer did not choose the most suitable hardware device for the model, which may lead to insufficient resource utilization. This puts forward higher requirements and challenges for designers' Algorithm and Hardware Co-design capabilities in the future, as well as higher requirements for the compatibility of development tools. In the future, designers may need to complete the optimization of the model, the Quantization of the software, and the hardware structure design simultaneously. We believe that the continued development of HLS as a tool can bridge this gap. HLS uses a high-level abstract programming language that provides the following benefits, 1) it conforms to the development habits of software developers, 2) it reduces the learning cost of hardware developers, and 3) it can be compatible with the development needs of both software and hardware.

Further, using GPU, FPGA, or ASIC for acceleration has exposed many shortcomings. For example, frequent data exchange leads to unacceptable communication overhead, and in the face of irregular data, DSP usage efficiency is low. While we can mitigate this problem by implementing a CPU kernel with an FPGA, it is not economical to sacrifice FPGA throughput for flexibility. We believe that deploying DSA on-chip for efficient data exchange and scheduling is highly important for optimizing model acceleration. Although we now have platforms like Zynq and MPSoC, their CPU performance still cannot meet higher demands. HARP and Versal ACAP may be candidates. They effectively solve the high frequency and high bandwidth CPU connection PL module and also have corresponding toolchain support. The benefits they bring are that 1) more hardware resources can be saved for processing irregular operations instead of routing; 2) simplified layout and routing can increase the operating frequency of the system.

Acknowledgments

This research was partly funded by Industry-University-Research Collaboration Foundation of Fuzhou University grant number 0101/01011919, and by the young scientist project of MOE innovation platform.

Conflict of interest

The authors declare there is no conflict of interest.

References

1 A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, et al., Attention is All you Need, in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017. https://doi.org/10.48550/arXiv.2206.09457

2. Q. Wang, B. Li, T. Xiao, J. Zhu, C. Li, D. F. Wong, et al., Learning deep transformer models for machine translation, preprint, arXiv:1906.01787.
3. S. A. Chowdhury, A. Abdelali, K. Darwish, J. Soon-Gyo, J. Salminen, B. J. Jansen, Improving arabic text categorization using transformer training diversification, in *Proceedings of the Fifth Arabic Natural Language Processing Workshop (COLING-WANLP)*, (2020), 226–236. https://aclanthology.org/2020.wanlp-1.21
4. X. Ma, P. Zhang, S. Zhang, N. Duan, Y. Hou, M. Zhou, et al., A tensorized transformer for language modeling, preprint, arXiv:1906.09777.
5. J. Devlin, M. W. Chang, K. Lee, K. Toutanova, BERT: Pre-training of deep bidirectional transformers for language understanding, preprint, arXiv:1810.04805.
6. Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, et al., RoBERTa: A robustly optimized BERT pretraining approach, preprint, arXiv:1907.11692.
7. H. Xu, B. Liu, L. Shu, P. S. Yu, BERT post-training for review reading comprehension and aspect-based sentiment analysis, preprint, arXiv:1904.02232.
8. P. Shi, J. Lin, Simple BERT models for relation extraction and semantic role labeling, preprint, arXiv:1904.05255.
9. V. Sanh, L. Debut, J. Chaumond, T. Wolf, DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter, preprint, arXiv:1910.01108.
10. Y. Cheng, D. Wang, P. Zhou, T. Zhang, Model compression and acceleration for deep neural networks: The principles, progress, and challenges, *IEEE Signal Process. Mag.*, **35** (2018), 126–136. https://doi.org/10.1109/MSP.2017.2765695
11. S. Cheng, D. Lucor, J. P. Argaud, Observation data compression for variational assimilation of dynamical systems, *J. Comput. Sci.*, **53** (2021), 101405. https://doi.org/10.1016/j.jocs.2021.101405
12. S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, J. Du, On-demand deep model compression for mobile devices: A usage-driven model selection framework, in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, (2018), 389–400. https://doi.org/10.1145/3210240.3210337
13. S. Liu, J. Du, K. Nan, Z. Zhou, H. Liu, Z. Wang, et al., AdaDeep: A usage-driven, automated deep model compression framework for enabling ubiquitous intelligent mobiles, *IEEE Trans. Mob. Comput.*, **20** (2021), 3282–3297. https://doi.org/10.1109/TMC.2020.2999956
14. V. L. Tran, S. E. Kim, Efficiency of three advanced data-driven models for predicting axial compression capacity of CFDST columns, *Thin-Walled Struct.*, **152** (2020), 106744. https://doi.org/10.1016/j.tws.2020.106744

15. Z. X. Hu, Y. Wang, M. F. Ge, J. Liu, Data-driven fault diagnosis method based on compressed sensing and improved multiscale network, *IEEE Trans. Ind. Electron.*, **67** (2020), 3216–3225. https://doi.org/10.1109/TIE.2019.2912763
16. S. Cheng, I. C. Prentice, Y. Huang, Y. Jin, Y. K. Guo, R. Arcucci, Data-driven surrogate model with latent data assimilation: Application to wildfire forecasting, *J. Comput. Phys.*, **464** (2022). https://doi.org/10.1016/j.jcp.2022.111302
17. S. Yang, Z. Zhang, C. Zhao, X. Song, S. Guo, H. Li, CNNPC: End-edge-cloud collaborative CNN inference with joint model partition and compression, *IEEE Trans. Parallel Distrib. Syst.*, (2022), 1–1. https://doi.org/10.1109/TPDS.2022.3177782
18. H. He, S. Jin, C. K. Wen, F. Gao, G. Y. Li, Z. Xu, Model-driven deep learning for physical layer communications, *IEEE Wireless Commun.*, **26** (2019), 77–83. https://doi.org/10.1109/MWC.2019.1800447
19. Z. Liu, M. del Rosario, Z. Ding, A markovian model-driven deep learning framework for massive MIMO CSI feedback, *IEEE Trans. Wireless Commun.*, **21** (2022), 1214–1228. https://doi.org/10.1109/TWC.2021.3103120
20. W. Wang, F. Wei, L. Dong, H. Bao, N. Yang, M. Zhou, MiniLM: Deep self-attention distillation for task-agnostic compression of pre-trained transformers, preprint, arXiv:2002.10957.
21. X. Jiao, Y. Yin, L. Shang, X. Jiang, X. Chen, L. Li, et al., TinyBERT: Distilling BERT for natural language understanding, preprint, arXiv:1909.10351.
22. S. Sun, Y. Cheng, Z. Gan, J. Liu, Patient knowledge distillation for BERT model compression, preprint, arXiv:1908.09355.
23. H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles, H. Jegou, Training data-efficient image transformers & distillation through attention, in *Proceedings of the 38th International Conference on Machine Learning (ICML)*, (2021), 10347–10357. https://doi.org/10.48550/arXiv.2012.12877
24. P. Michel, O. Levy, G. Neubig, Are sixteen heads really better than one?, *Adv. Neural Inf. Process. Syst.*, preprint, arXiv:1905.10650.
25. M. A. Gordon, K. Duh, N. Andrews, Compressing BERT: Studying the effects of weight pruning on transfer learning, preprint, arXiv:2002.08307.
26. T. Chen, Y. Cheng, Z. Gan, L. Yuan, L. Zhang, Z. Wang, Chasing sparsity in vision transformers: An end-to-end exploration, *Adv. Neural Inf. Process. Syst.*, (2021), 19974–19988. https://doi.org/10.48550/arXiv.2106.04533
27. T. Chen, J. Frankle, S. Chang, S. Liu, Y. Zhang, Z. Wang, et al., The lottery ticket hypothesis for pre-trained BERT networks, *Adv. Neural Inf. Process. Syst.*, (2020), 15834–15846. https://doi.org/10.48550/arXiv.2007.12223
28. S. Shen, Z. Dong, J. Ye, L. Ma, Z. Yao, A. Gholami, et al., Q-BERT: Hessian based ultra low precision quantization of BERT, preprint, arXiv:1909.05840.
29. Z. Liu, Y. Wang, K. Han, S. Ma, W. Gao, Post-training quantization for vision transformer, preprint, arXiv:2106.14156.
30. H. Bai, W. Zhang, L. Hou, L. Shang, J. Jin, X. Jiang, et al., BinaryBERT: Pushing the limit of BERT quantization, preprint, arXiv:2012.15701.
31. O. Zafrir, G. Boudoukh, P. Izsak, M. Wasserblat, Q8BERT: Quantized 8Bit BERT, in *the 5th Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS 2019*, (2019), 36–39. https://doi.org/10.1109/EMC2-NIPS53020.2019.00016

32. Z. Wu, Z. Liu, J. Lin, Y. Lin, S. Han, Lite transformer with long-short range attention, preprint, arXiv:2004.11886.
33. L. Hou, Z. Huang, L. Shang, X. Jiang, X. Chen, Q. Liu, DynaBERT: Dynamic BERT with adaptive width and depth, preprint, arXiv:2004.04037.
34. M. Chen, H. Peng, J. Fu, H. Ling, AutoFormer: Searching transformers for visual recognition, in *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, (2021), 12250–12260. https://doi.org/10.1109/ICCV48922.2021.01205
35. P. Ganesh, Y. Chen, X. Lou, M. A. Khan, Y. Yang, H. Sajjad, et al., Compressing large-scale transformer-based models: A case study on BERT, *Trans. Assoc. Comput. Linguist.*, **9** (2021), 1061–1080. https://doi.org/10.1162/tacl_a_00413
36. S. Hochreiter, J. Schmidhuber, Long short-term memory, *Neural Comput.*, **9** (1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735
37. J. Chung, C. Gulcehre, K. Cho, Y. Bengio, Empirical evaluation of gated recurrent neural networks on sequence modeling, preprint, arXiv:1412.3555.
38. D. Bahdanau, K. Cho, Y. Bengio, Neural machine translation by jointly learning to align and translate, preprint, arXiv:1409.0473.
39. B. Li, S. Pandey, H. Fang, Y. Lyv, J. Li, J. Chen, et al., FTRANS: energy-efficient acceleration of transformers using FPGA, in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, (2020), 175–180. https://doi.org/10.1145/3370748.3406567
40. T. J. Ham, S. J. Jung, S. Kim, Y. H. Oh, Y. Park, Y. Song, et al., A^3: Accelerating attention mechanisms in neural networks with approximation, in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, (2020), 328–341. https://doi.org/10.1109/HPCA47549.2020.00035
41. T. J. Ham, Y. Lee, S. H. Seo, S. Kim, H. Choi, S. J. Jung, et al., ELSA: Hardware-software co-design for efficient, lightweight self-attention mechanism in neural networks, in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, (2021), 692–705. https://doi.org/10.1109/ISCA52012.2021.00060
42. X. Zhang, Y. Wu, P. Zhou, X. Tang, J. Hu, Algorithm-hardware co-design of attention mechanism on FPGA devices, *ACM Trans. Embed. Comput. Syst.*, **20** (2021), 1–24. https://doi.org/10.1145/3477002
43. S. Lu, M. Wang, S. Liang, J. Lin, Z. Wang, Hardware accelerator for multi-head attention and position-wise feed-forward in the transformer, in *IEEE International SOC Conference*, (2020), 84–89. https://doi.org/10.1109/SOCC49529.2020.9524802
44. A. Parikh, O. Täckström, D. Das, J. Uszkoreit, A decomposable attention model for natural language inference, in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, (2016), 2249–2255. https://doi.org/10.48550/arXiv.1606.01933
45. Z. Lin, M. Feng, C. N. dos Santos, M. Yu, B. Xiang, B. Zhou, et al., A structured self-attentive sentence embedding, preprint, arXiv:1703.03130
46. M. S. Charikar, Similarity estimation techniques from rounding algorithms, in *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, (2002), 380–388. https://doi.org/10.1145/509907.509965
47. X. Zhang, F. X. Yu, R. Guo, S. Kumar, S. Wang, S. F. Chang, Fast orthogonal projection based on kronecker product, in *2015 IEEE International Conference on Computer Vision (ICCV)*, (2015), 2929–2937. https://doi.org/10.1109/ICCV.2015.335

48. Y. Gong, S. Kumar, H. A. Rowley, S. Lazebnik, Learning binary codes for high-dimensional data using bilinear projections, in *2013 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, (2013), 484–491. https://doi.org/10.1109/CVPR.2013.69
49. M. Wang, S. Lu, D. Zhu, J. Lin, Z. 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)*, (2018), 223–226. https://doi.org/10.1109/APCCAS.2018.8605654
50. R. Hu, B. Tian, S. Yin, S. Wei, Efficient hardware architecture of softmax layer in deep neural network, in *2018 IEEE 23rd International Conference on Digital Signal Processing (DSP)*, (2018), 1–5. https://doi.org/10.1109/ICDSP.2018.8631588
51. L. Deng, G. Li, S. Han, L. Shi, Y. Xie, Model compression and hardware acceleration for neural networks: A comprehensive survey, *Proc. IEEE*, **108** (2020), 485–532. https://doi.org/10.1109/JPROC.2020.2976475
52. C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, et al., C IR CNN: Accelerating and compressing deep neural networks using block-circulant weight matrices, in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, (2017), 395–408. https://doi.org/10.1145/3123939.3124552
53. S. Wang, Z. Li, C. Ding, B. Yuan, Q. Qiu, Y. Wang, et al., 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 (FPGA)*, (2018), 11–20. https://doi.org/10.1145/3174243.3174253
54. L. Zhao, S. Liao, Y. Wang, Z. Li, J. Tang, B. Yuan, Theoretical properties for neural networks with weight matrices of low displacement rank, in *Proceedings of the 34th International Conference on Machine Learning (ICML)*, (2017), 4082–4090. https://doi.org/10.48550/arXiv.1703.00144
55. V. Y. Pan, Structured matrices and displacement operators, in *Structured Matrices and Polynomials: Unified Superfast Algorithms*, Springer Science & Business Media, (2001), 117–153. https://doi.org/10.1007/978-1-4612-0129-8
56. J. O. Smith, Mathematics of the discrete fourier transform (DFT): with audio applications, in *Mathematics of the Discrete Fourier Transform (DFT): With Audio Applications*, Julius Smith, (2007), 115–164. https://ccrma.stanford.edu/~jos/st/
57. Z. Liu, G. Li, J. Cheng, Hardware acceleration of fully quantized BERT for efficient natural language processing, in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, (2021), 513–516. https://doi.org/10.23919/DATE51398.2021.9474043
58. M. Sun, H. Ma, G. Kang, Y. Jiang, T. Chen, X. Ma, et al., VAQF: Fully automatic software-hardware co-design framework for low-bit vision transformer, preprint, arXiv:2201.06618.
59. Z. Liu, Z. Shen, M. Savvides, K. T. Cheng, ReActNet: Towards precise binary neural network with generalized activation functions, in *Computer Vision–ECCV 2020 (ECCV)*, (eds. Vedaldi. A., Bischof. H., Brox. T., Frahm. J.-M.), Cham, Springer International Publishing, (2020), 143–159. https://doi.org/10.1007/978-3-030-58568-6_9
60. M. Rastegari, V. Ordonez, J. Redmon, A. Farhadi, XNOR-Net: ImageNet classification using binary convolutional neural networks, in *Computer Vision–ECCV 2016 (ECCV)*, (eds. Leibe. B., Matas. J., Sebe. N., Welling. M.), Cham, Springer International Publishing, (2016), 525–542. https://doi.org/10.1007/978-3-319-46493-0_32

61. S. Han, H. Mao, W. J. Dally, Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, preprint, arXiv:1510.00149.
62. W. Wen, C. Wu, Y. Wang, Y. Chen, H. Li, Learning structured sparsity in deep neural networks, in *Advances in Neural Information Processing Systems (NeurIPS)*, Curran Associates, (2016). https://doi.org/10.48550/arXiv.1608.03665
63. X. Ma, F. M. Guo, W. Niu, X. Lin, J. Tang, K. Ma, et al., PCONV: The missing but desirable sparsity in DNN weight pruning for real-time execution on mobile devices, in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, (2020), 5117–5124. https://doi.org/10.1609/aaai.v34i04.5954
64. B. Li, Z. Kong, T. Zhang, J. Li, Z. Li, H. Liu, et al., Efficient transformer-based large scale language representations using hardware-friendly block structured pruning, preprint, arXiv:2009.08065.
65. S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, et al., 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 (FPGA)*, (2019), 63–72. https://doi.org/10.1145/3289602.3293898
66. H. Peng, S. Huang, T. Geng, A. Li, W. Jiang, H. Liu, et al., Accelerating transformer-based deep learning models on FPGAs using column balanced block pruning, in *2021 22nd International Symposium on Quality Electronic Design (ISQED)*, (2021), 142–148. https://doi.org/10.1109/ISQED51717.2021.9424344
67. C. Ding, A. Ren, G. Yuan, X. Ma, J. Li, N. Liu, et al., Structured weight matrices-based hardware accelerators in deep neural networks: FPGAs and ASICs, in *Proceedings of the 2018 on Great Lakes Symposium on VLSI (GLSVLSI)*, Chicago, IL, USA, Association for Computing Machinery, (2018), 353–358. https://doi.org/10.1145/3194554.3194625
68. S. Narang, E. Undersander, G. Diamos, Block-sparse recurrent neural networks, preprint, arXiv:1711.02782.
69. P. Qi, E. H. M. Sha, Q. Zhuge, H. Peng, S. Huang, Z. Kong, et al., Accelerating framework of transformer by hardware design and model compression co-optimization, in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, (2021), 1–9. https://doi.org/10.1109/ICCAD51958.2021.9643586
70. P. Qi, Y. Song, H. Peng, S. Huang, Q. Zhuge, E. H. M. Sha, Accommodating transformer onto FPGA: Coupling the balanced model compression and FPGA-implementation optimization, in *Proceedings of the 2021 on Great Lakes Symposium on VLSI (GLSVLSI)*, Virtual Event, USA, Association for Computing Machinery, (2021), 163–168. https://doi.org/10.1145/3453688.3461739
71. D. So, Q. Le, C. Liang, The evolved transformer, in *Proceedings of the 36th International Conference on Machine Learning (ICML)*, PMLR, (2019), 5877–5886. https://doi.org/10.48550/arXiv.1901.11117
72. H. Wang, Efficient algorithms and hardware for natural language processing, Graduate Theses, Retrieved from the Massachusetts Institute of Technology, 2020. https://hdl.handle.net/1721.1/127440.
73. H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, et al., Bit fusion: Bit-Level dynamically composable architecture for accelerating deep neural network, in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, (2018), 764–775. https://doi.org/10.1109/ISCA.2018.00069

74. R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, et al., Templates for the solution of linear systems: Building blocks for iterative methods, in *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, Society for Industrial and Applied Mathematics, (1994), 39–55. https://doi.org/10.1137/1.9781611971538
75. W. Liu, B. Vinter, CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication, in *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS)*, Newport Beach, California, USA, Association for Computing Machinery, (2015), 339–350. https://doi.org/10.1145/2751205.2751209
76 R. Kannan, Efficient sparse matrix multiple-vector multiplication using a bitmapped format, in *20th Annual International Conference on High Performance Computing (HiPC)*, (2013), 286–294. https://doi.org/10.1109/HiPC.2013.6799135
77. W. Jiang, X. Zhang, E. H. M. Sha, L. Yang, Q. Zhuge, Y. Shi, et al., Accuracy vs. efficiency: achieving both through FPGA-implementation aware neural architecture search, in *Proceedings of the 56th Annual Design Automation Conference 2019 (DAC)*, Las Vegas NV USA, ACM, (2019), 1–6. https://doi.org/10.1145/3316781.3317757
78. W. Jiang, E. H. M. Sha, X. Zhang, L. Yang, Q. Zhuge, Y. Shi, et al., Achieving super-linear speedup across multi-FPGA for real-time DNN inference, preprint, arXiv:1907.08985.
79. W. Jiang, X. Zhang, E. H. M. Sha, Q. Zhuge, L. Yang, Y. Shi, et al., XFER: A novel design to achieve super-linear performance on multiple FPGAs for real-time AI, in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, Seaside, CA, USA, Association for Computing Machinery, (2019), 305. https://doi.org/10.1145/3289602.3293988

AIMS Press

©2022 the Author(s), licensee AIMS Press. This is an open access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0)