SOLE: Hardware-Software Co-design of Softmax and LayerNorm for Efficient Transformer Inference

Wenxun Wang*, Shuchang Zhou†, Wenyu Sun*, Peiqin Sun† and Yongpan Liu*
* Department of Electronic Engineering, Tsinghua University, Beijing, China
† MEGVII Technology, Beijing, China
wx-wang23@mails.tsinghua.edu.cn, wy-sun@sz.tsinghua.edu.cn, ypliu@tsinghua.edu.cn, †{zsc,sunpeiqin}@megvii.com

2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD) | 979-8-3503-2225-5/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICCAD57390.2023.10323725

***Abstract*—Transformers have shown remarkable performance in both natural language processing (NLP) and computer vision (CV) tasks. However, their real-time inference speed and efficiency are limited due to the inefficiency in Softmax and Layer Normalization (LayerNorm). Previous works based on function approximation suffer from inefficient implementation as they place emphasis on computation while disregarding memory overhead concerns. Moreover, such methods rely on retraining to compensate for approximation error which can be costly and inconvenient.**

In this paper, we present SOLE, a hardware-software co-design for Softmax and LayerNorm which is composed of E2Softmax and AILayerNorm. E2Softmax utilizes log2 quantization of exponent function and log-based division to approximate Softmax while AILayerNorm adopts low-precision statistic calculation. Compared with state-of-the-art designs, we achieve both low-precision calculation and low bit-width storage on Softmax and LayerNorm. Experiments show that SOLE maintains inference accuracy without retraining while offering orders of magnitude speedup and energy savings over GPU, achieving 3.04×, 3.86× energy-efficiency improvements and 2.82×, 3.32× area-efficiency improvements over prior state-of-the-art custom hardware for Softmax and LayerNorm, respectively.

***Index Terms*—Transformers, neural networks, hardware-software co-design, softmax, layer normalization**

I. Introduction

In recent years, transformer-based networks [1] have achieved success in enhancing the performance of computer vision (CV) tasks [2]–[4] as well as natural language processing (NLP) tasks [5]–[7]. Despite their impressive performance, their computational characteristics have become a non-negligible defect. Compared to other networks, like CNNs [8] and RNNs [9], transformers have a large number of parameters and massive computation overhead, i.e. ViT-22B [10], GPT-3 [11], PaLM [12]. Efforts have been made to mitigate the problem. Algorithms like quantization [13]–[15] are developed to diminish memory footprint and reduce computation overhead while many accelerators have also been proposed to expedite inference [16]–[19]. These endeavors primarily focused on accelerating the matrix multiplication in transformers, which is widely considered the primary bottleneck of conventional network, such as CNNs and MLPs. However, little attention has been paid to the non-linear computations that exist extensively in self-attention mechanism, including Softmax and Layer Normalization (LayerNorm). Implemented by 32-bit floating-point (FP32) arithmetic units, these operations severely restrict the enhancement of end-to-end transformer inference as reported in [20]. As shown in Fig. 1(a), this issue becomes worse when matrix multiplications are calculated using 8-bit integer (INT8) arithmetic after quantization, emphasizing the urgent need for a solution.

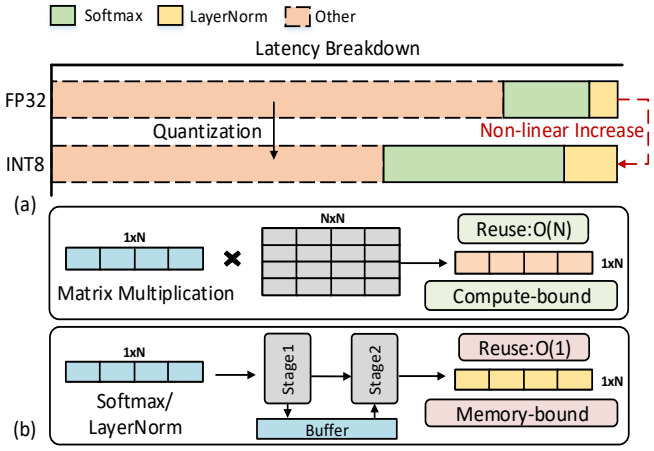

Fig. 1. (a) Latency Breakdown of Deit-Tiny with the image size of 448 on a 2080Ti GPU. (b) Difference between MatMul operations with Softmax/LayerNorm operations.

Previous works have proposed methods based on function approximation to optimize these operations. However, the effectiveness of these works is limited to computation, and storage concerns are often overlooked. Fig. 1(b) illustrates two distinct differences between Softmax/LayerNorm operations and traditional matrix multiplication. Firstly, Softmax/LayerNorm operations employ a two-stage dataflow, which requires buffering intermediate data since the output cannot be generated directly. Secondly, due to the rarity of input data reuse in these operations, it is challenging to amortize memory costs through computation. Collectively, these two factors contribute to the memory-bound issue of Softmax/LayerNorm operations. Prior works including Softermax [20] and I-BERT [21] fail to address this problem as they still need to buffer 16-bit and 32-bit data in the process, respectively. Moreover, retraining or fine-tuning is usually required for such works as compensation of approximation errors, incurring additional training overhead

Yongpan Liu and Peiqin Sun are the corresponding authors.

that becomes increasingly expensive as transformers grow larger.

In this paper, we propose SOLE, a hardware/software co-design method to solve these problems. In SOLE, we design E2Softmax and AILayerNorm for Softmax and LayerNorm, respectively. For Softmax, log2 quantization is appiled on the output of exponent function, which compresses intermediate data to 4-bit and significantly mitigates the memory-bound issue. With log2 quantization, the exponent function is implemented through a specialized Log2Exp Unit that is composed solely of shifters and adders. Besides, floating-point division is substituted with the proposed Approximate Log-based Division to fully utilize the log2-quantized output. In this way, E2Softmax can be implemented multiplication-free and LUT-free. For LayerNorm, we find that the statistic calculation is resilient to errors introduced by small variation in inputs. Hence, we adopt dynamic compression and combine it with Power-of-Two Factor (PTF) [22], uniformly optimizing the dataflow to achieve low-precision statistic calculation. Consequently, AILayerNorm requires only 8-bit data buffering and 4-bit multiplication for statistic calculation.

In summary, our contributions can be concluded as follows:

- We propose Efficient log2 quantized Softmax (E2Softmax), a hardware-friendly softmax algorithm based on log2 quantization of exponent function.
- We propose Approximate Integer Layer Normalization (AILayerNorm), an efficient layernorm algorithm with low-precision statistic calculation.
- Exclusive software experiments have been conducted on CV and NLP tasks with various transformer-based network. Results show that SOLE incurs negligible accuracy drops without additional training.
- Hardware experiments demonstrate that SOLE delivers 36.2x and 61.3x average speedup, as well as orders-of-magnitude energy-efficiency improvements over GPU for Softmax and LayerNorm, respectively. In comparison to state-of-the-art custom hardware, SOLE provides 2.82x and 3.32x area-efficiency improvements and 3.04x and 3.86x energy-efficiency improvements for Softmax and LayerNorm, respectively

II. Related Work

A. Prominent Non-Linear Operations in Transformers

A typical transformer model consists of stacked encoder and decoder blocks, each of which is composed of multiple layers of two sub-modules: a multi-head self-attention block and a feed-forward network [1]. Non-linear operations occupy significant parts in the transformer computation [20], especially Softmax and LayerNorm. The Softmax function is used to compute the attention weights for the multi-head self-attention mechanism, while LayerNorm is used to normalize the hidden states at each layer of the model. Softmax and LayerNorm can be defined as follows:

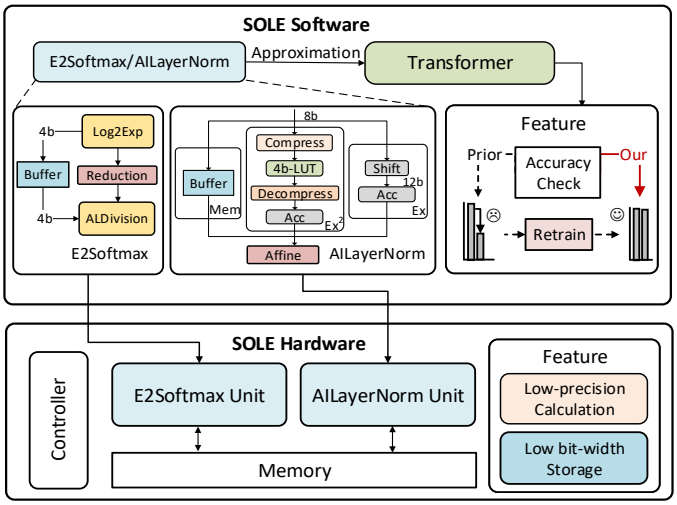

Fig. 2. Overview of SOLE

$$\begin{aligned} \mathrm{Softmax}(X_i) &= \frac{\exp(X_i - X_{max})}{\sum_{j=1}^{n} \exp(X_j - X_{max})} \\ \mathrm{LayerNorm}(X_i) &= \frac{X_i - \mu}{\sigma} \cdot \gamma + \beta \end{aligned} \tag{1}$$

where $\mu = \frac{1}{C}\sum_{i=1}^{C} X_i, \sigma = \sqrt{\frac{1}{C}\sum_{i=1}^{C}(X_i - \mu)^2}$. These operations contribute a large fraction of run-time in transformer inference when implemented with costly FP32 arithmetic computation [20]. For hardware targeting low-end device, it is considered as significant overhead [23]. Therefore, several methods have been proposed to address this issue.

B. Accelerating Non-Linear Operations

A conventional method for accelearating non-linear operations is function approximation [24], [25]. For instance, Softermax [20] proposed low-precision computation for Softmax. I-BERT [21] used 32-bit integer arithmetic as an approximation of Softmax, LayerNorm and other non-linear operations. NN-LUT [26] adpoted hardware-friendly Look-up table (LUT) and linear piece-wise approximation of non-linear operations based on I-BERT code base. The LUT contents were obtained through one-hidden-layer ReLU network. Desipte the benefits, there are two notable drawbacks associated with these approaches. Firstly, fine-tuning or retraining is required for prior works [20], [21] to recover the performance, incurring increasing costs as transformers grow larger. Secondly, these works fail to make efficient implementation of Softmax and LayerNorm since both high-precision multiplication and high bit-width data storage are still indispensable.

Fig. 2 shows the overview of SOLE. Unlike prior works, SOLE develops E2Softmax and AILayerNorm algorithms specifically designed for Softmax and LayerNorm. These algorithms incur minimal accuracy loss without the requirement for retraining. Additionally, SOLE introduces custom

hardware units to support these algorithm, enabling efficient implementation through low-precision calculation and low bit-width storage.

III. SOLE Software

In SOLE, we design E2Softmax and AILayerNorm algorithms for efficient Softmax and LayerNorm implementation. In this section, we first introduce the preliminaries of our algorithms. Then, we explain the details of E2Softmax and AILayerNorm algorithms.

A. Preliminary

We explain the techniques relevant to the proposed algorithms in this section.

Log2 Quantization. Generally speaking, it converts continues value in $(0, 1)$ into a set of discrete numbers like $2^0, 2^{-1}, 2^{-2}$ [27]. Assuming the quantization bit-width b, the log2 quantization process can be defined as:

$$\text{Log2Q}(X) = \text{Clip}(\lfloor -\log_2(X) \rceil, 0, 2^b - 1), X \in (0, 1) \quad (2)$$

Log-based Division. Proposed by Mitchell [28], it uses linear approximation to compute the logarithm of binary numbers and performs division by subtract and shift operations. Suppose an unsigned N-bit integer X, it can be defined as:

$$X = \sum_{i=0}^{N-1} 2^i b_i = 2^{k_x} + \sum_{i=0}^{k_x - 1} 2^i b_i = 2^{k_x}(1 + x) \quad (3)$$

where k_x represents the leading-one bit of X, b_i stands for the $i-th$ bit and $x = (\sum_{i=0}^{k_x-1} 2^{i-k_x} b_i) \in (0, 1)$. The approximation of the logarithm is described as $\log_2(X) = k_x + x$, where k_x is the characteristic part and x is the decimal part. When it comes to the division, for instance, $Q = \frac{X_1}{X_2}$, we can approximate the log of quotient as:

$$\log_2(Q) = k_1 + x_1 - k_2 - x_2 \quad (4)$$

The approximate quotient is calculated by applying the inverse of the approximate logarithm:

$$Q^{'} = \begin{cases} 2^{k_1 - k_2 - 1}(2 + x_1 - x_2), & x_1 - x_2 < 0 \\ 2^{k_1 - k_2}(1 + x_1 - x_2), & x_1 - x_2 \geq 0 \end{cases} \quad (5)$$

Since the inverse of logarithm requires the fractional part is in $(0, 1)$, the expression is divided into two parts in case a borrow is taken from the characteristic part when $x_1 - x_2 < 0$.

Power-of-Two Factor (PTF). Proposed by [22] for LayerNorm quantization, PTF equips different channels with different factors to address the inter-channel variation in the inputs of LayerNorm layers. Assuming the input activation $\mathrm{X} \in \mathbb{R}^{\mathrm{B} \times \mathrm{L} \times \mathrm{C}}$, the quantization bit-width b, the layer-wise quantization parameters $\mathrm{s}, \mathrm{zp} \in \mathbb{R}^1$,and the PTF $\alpha \in \mathbb{N}^{\mathrm{C}}$, the quantized activation $\mathrm{X_Q}$ can be defined as:

$$X_Q = \text{Clip}(\lfloor \frac{X}{2^\alpha s} \rceil + \mathrm{zp}, 0, 2^b - 1) \quad (6)$$

where s and zp stand for the quantization scale and zero point, respectively. Generally speaking, channels with large range of activation will have larger α such that their scaling factors appropriately match their range.

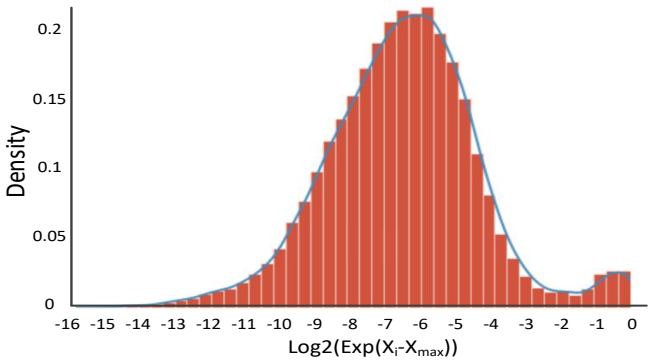

Fig. 3. Distribution of $\mathbf{Exp}(X_i - X_{max})$ in logarithm.

B. E2Softmax

E2Softmax algorithm adopts several techniques to improve efficiency. Firstly, we apply log2 quantization on the output of exponent function. Secondly, we substitute floating-point division and exponent function with approximate log-based division and Log2Exp function, as shown in Algorithm 1. All computations are carried out in fixed-point precision with 8-bit quantized inputs. An online normalization scheme [29] is added to further reduce latency.

Log2 Quantization of Exponent Output. The exponent function is a crucial component of Softmax, but its non-linearity makes it expensive to implement. Typically, multiplication and large LUTs are required to perform linear piece-wise approximation, inducing power and area burden. To alleviate the issue, we apply log2 quantization on the output of exponent operation so that it can be implemented hardware-friendly, which can be defined as follows:

$$\text{Log2Exp}(x) = -\lfloor \log2(e^x) \rceil = -\lfloor (x) \times \frac{1}{ln2} \rceil \quad (7)$$

$\frac{1}{ln2}$ can be approximated as 1.4375. Hence, the function is simply:

$$\text{Log2Exp}(x) = -\lfloor x + x >> 1 - x >> 4 \rceil, x \in (-\infty, 0] \quad (8)$$

We adopt this transformation mainly for three reasons. Firstly, the inputs of exponent function need to perform subtraction with the maximum of the related vector to avoid overflow, naturally satisfying the input range of Log2Exp function. Secondly, the distribution of the exponent output is similar to the normalization distribution when plotted on a log2 scale (as shown in Fig. 3), making the log2 method an ideal quantization choice. Thirdly, the Softmax function is concerned with the relative value of the exponent output instead of the absolute value. Therefore, the error introduced through log2 quantization of the exponent output will be decreased after division [17]. In this way, we can also implement exponent function in a more efficient way. Experiments in Table I and Table II show that the exponent output can be quantized to 4-bit with minimal accuracy drop through log2 quantization, resulting in further downsizing of memory footprints.

Approximate Log-based Division. Another non-linear operation in Softmax is division, which also relies large LUTs and high-precision multipliers. Since 4-bit logarithm of the exponent output is obtained through log2 quantization, it is reasonable to leverage log-based division to further reduce computational complexity. Assuming the log2 quantized exponent output $k_y = \text{Log2Exp}(Q_y)$ and the reduced sum $S = 2^{k_s} \cdot (1+s), s \in (0,1)$, we have:

$$\frac{Q_y}{S} = 2^{-(k_y+k_s)} \cdot \frac{1}{1+s} \tag{9}$$

k_s represents the position of the leading-one of S and s stands for the rest bits. We adopt linear approximation and quantize the s to 1-bit $q(s)$:

$$\begin{aligned} q(s) &= \frac{\lfloor 2 \cdot s \rfloor}{2},\ q(s) \in \{0, 0.5\} \\ \frac{Q_y}{S} &\approx 2^{-(k_y+k_s+1)} \cdot (1 - q(s)) \end{aligned} \tag{10}$$

Therefore, the error between approximate division and full-precision version can be formulated as:

$$\delta = 2^{-(k_y+k_s)} \cdot \frac{s - 1 - q(s) \cdot (s+1)}{2(1+s)} \tag{11}$$

Considering s as an uniform random variable that takes value in $[0, 1]$, we can get the expectation of the error:

$$\begin{aligned} E(\delta) &= 2^{-(k_y+k_s)} \left(\int_0^{0.5} \frac{s-1}{2(s+1)} ds + \int_{0.5}^{1} \frac{s-3}{4(1+s)} ds \right) \\ &= 2^{-(k_y+k_s+1)} \times (\text{-}0.636) \end{aligned} \tag{12}$$

To make our approximation unbiased, we modify the Approximate Log-based Division (ALDivision) as:

$$\text{ALDivision}(k_y, S) = 2^{-(k_y+k_s+1)} \cdot (1.636 - q(s)) \tag{13}$$

In this way, we can perform division through hardware-friendly shift and subtraction operation. In fact, the subtraction can be replaced with a two-way multiplexer since $q(s)$ is a 1-bit number. Implementation details will be discussed in the section 4.

Algorithm 1 Efficient Log2 Quantized Softmax

Input: $X \in \mathbb{R}^L$: Input Activation
Ouput: $Y \in \mathbb{R}^L$: Softmax Output
$m_0 \leftarrow -\infty$
$Sum \leftarrow 0$
for $i \leftarrow 1$ to L **do** ▷ Stage1
    $m_i \leftarrow \text{Max}(X_i, m_{i-1})$
    $Y_i \leftarrow \text{Log2Exp}(X_i - m_i)$
    $Sub \leftarrow \text{Log2Exp}(m_{i-1} - m_i)$
    $Sum \leftarrow Sum >> Sub + 2^{-Y_i}$
end for
for $i \leftarrow 1$ to L **do** ▷ Stage2
    $Sub \leftarrow \text{Log2Exp}(m_i - m_L)$
    $Y_i \leftarrow \text{ALDivision}(Sub + Y_i, Sum)$
end for
return Y

C. AILayerNorm

AILayerNorm algorithm utilizes dynamic compression to achieve low-precision statistic calculation based on the PTF quantization. The procedure of AILayerNorm is presented in Algorithm 2.

Dynamic Compression. LayerNorm utilizes mean and variance to normalize inputs. Therefore statistic calculation is a crucial part of the algorithm. To obtain the mean and variance, the expected value of the input and squared input ($E(x)$ and $E(x^2)$) are typically computed [30]. While $E(x)$ can be computed using only addition, $E(x^2)$ requires a large number of multiplications. Therefore, we propose dynamic compression for low-precision statistic calculations, driven by the fact that small values are less important in the reduction of x^2 than in x:

$$\frac{x_1^2}{x_1^2 + x_2^2} < \frac{x_1}{x_1 + x_2}, \quad x_1 < x_2 \text{ and } x_1, x_2 > 0 \tag{14}$$

To reduce computational complexity and minimize performance loss, we propose a dynamic compression method for 8-bit unsigned integer $x[7:0]$ to generate a 4-bit approximation $y[3:0]$, which dynamically compresses inputs based on their value. As shown in Fig. 5, we filter some less significant bits for compression. The length of bit-filtered is dynamically changed based on the value of input, which can be defined as $y, s = \text{Dynamic Compress}(x)$:

$$\text{Dynamic Compress}(x) = \begin{cases} \text{Clip}(\lfloor x/2^4 \rceil, 0, 15),\ 1,\ x[7:6] \neq 0 \\ \text{Clip}(\lfloor x/2^2 \rceil, 0, 15),\ 0,\ \text{else} \end{cases} \tag{15}$$

With our compression method, the calculation of x^2 can be done using 4-bit integer arithmetic and shift operation. A 1-bit signal s is also generated to determine the length of bit shift (2 or 4) when recovering correct inputs. Experiments show that our method only induces errors of 0.2% over $E(x^2)$ and 0.4% over standard deviation with uniformly distributed input data.

Low-precision statistic calculation Our method is based on the PTF for LayerNorm quantization which addresses inter-channel variation of LayerNorm and compresses input acvtivation to 8-bit integer. However, channel-wise shift operations are required before statistic calculation to obtain accurate value of inputs, leading to high-precision calculation of variance. Specifically, 12-bit multiplication must be performed to calculate X^2 after bit-shifting of 8-bit integer X. Hence, We optimize the dataflow to tackle this problem based on an equivalent mathematical transformation.

$$\begin{aligned} \text{Square}(X, \alpha) &= (X << \alpha) \cdot (X << \alpha) \\ &= (X \cdot X) << (2\alpha) \end{aligned} \tag{16}$$

It is arranged in the Decompress phase along with the decoding of dynamic compression. In conjunction with dynamic compression mentioned before, this optimization avoids 12-bit multiplication in statistic calculation and uses 4-bit integer arithmetic instead, contributing to low-precision statistic calculation.

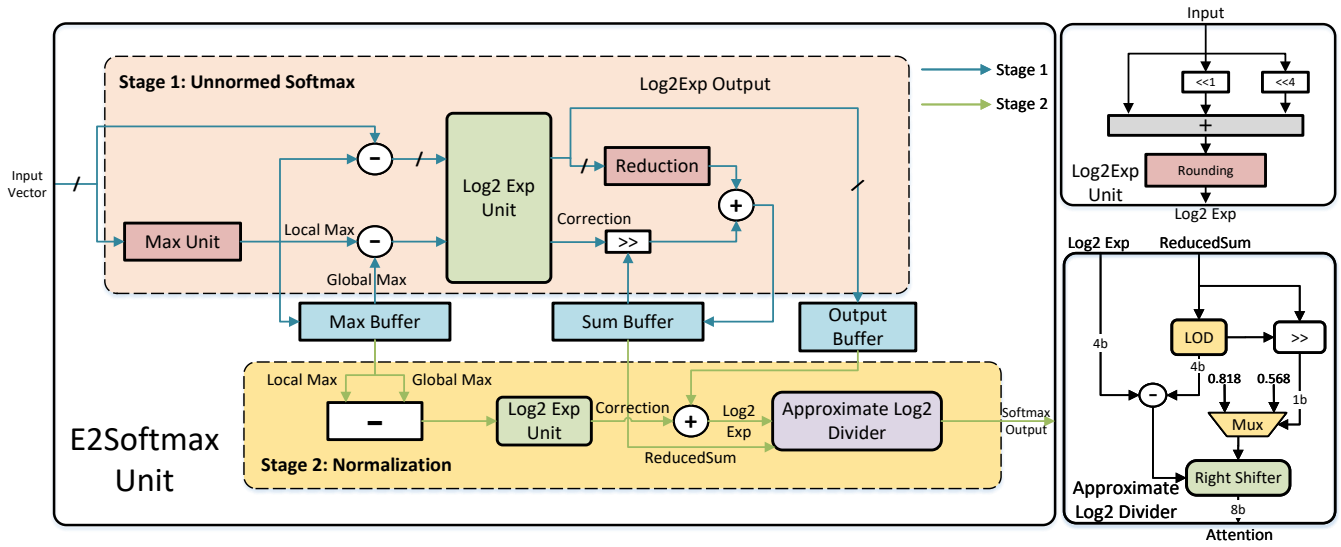

Fig. 4. Hardware design of E2Softmax Unit

Algorithm 2 Approximate Integer Layer Normalization

Input: X, α : quantized input activation, power of two factor
γ, β, zp : affine weight, affine bias, zero point
Output: Y : quantized output activation
for $i \leftarrow 0$ to C **do** ▷ Stage1
  $X_i \leftarrow X_i - zp$
  $X_c, s \leftarrow$ Dynamic Compress(X_i) ▷ Compress
  $X_c \leftarrow X_c^2 << (4s)$ ▷ Square & Decompress
  $E_x \leftarrow E_x + X_i << \alpha_i$ ▷ PTF Shift
  $E_{x^2} \leftarrow E_{x^2} + X_c << (2\alpha_i)$
end for
$E_x, E_{x^2} \leftarrow E_x \cdot \frac{1}{C}, (E_{x^2} << 4) \cdot \frac{1}{C}$
$\mu, std_{inv} \leftarrow E_x, (E_{x^2} - (E_x)^2)^{-\frac{1}{2}}$
for $i \leftarrow 0$ to C **do** ▷ Stage2
  $A, B \leftarrow \gamma_i \cdot std_{inv}$, β_i
  $X_i \leftarrow (X_i << \alpha_i - \mu)$
  $Y_i \leftarrow A \cdot X_i + B$
end for
return Y

IV. SOLE Hardware

In SOLE, we design custom hardware units as commodity hardware platforms do not support the special function units involved in algorithms. With custom hardware design, it can be integrated into other dedicated accelerators or tensor processing hardware like GPU tensor cores to accelerate transformer inference.

A. E2Softmax Unit

We propose E2Softmax Unit to implement our algorithm. Its key features include the Log2Exp Unit and Approximate Log-based Divider, which are implemented in a LUT-free and multiplication-free manner to achieve efficiency. As shown in Fig. 4, the computation process is divided into two stages to organize the dataflow. Buffers are designed ping-pong to support pipeline.

Unnormed Softmax. Stage 1 consists of three subunits, namely Max Unit, Log2Exp Unit and Reduction Unit. It receives a slice of the input vector since the length of vector can be as large as 1024. The Max Unit determines the local maximum of input vector by using a comparison tree. Then it is subtracted from the input vector and $GlobalMax$ read from buffer in case $LocalMax$ is larger than $GlobalMax$. The Log2Exp Unit receives them as inputs to generate outputs by shifts and additions, while modern hardware usually requires LUTs and multiplier. Rounding operations are applied at the end to get 4-bit integer outputs, namely $Log2Exp\ Output$ and $Correction$. The $Log2Exp\ Output$ are stored in the Output Buffer for Stage 2 and sent to the Reduction Unit at the same time, while the $Correction$ modifies the data from Sum Buffer for online normalization. Since $Log2Exp\ Output$ are quantized to 4-bit, little memory is required to store the intermediate result.

Normalization. Once the computation of reduced sum is completed, Stage 2 starts to perform division and generate softmax output. While division is generally approximated by linear piece-wise functions that rely LUTs and high-precision multipliers in prior works [20], [26], we introduce the Approximate Log-based Divider in our design. $Correction$ is firstly computed and added to generate $Log2Exp$ as modification, then $Log2Exp$ and $ReducedSum$ are sent to divider to obtain outputs. The Approximate Log-based Divider is simple and light-weight, consisting of a Leading-one detector (LOD), a subtractor, a two-way multiplexer and two shifters. As mentioned in algorithm 1, the characteristic part of $ReducedSum$ is produced by LOD and subtracts $Log2Exp$. Then it Selects the bit next to the leading-one bit by shifting and generate a 1-bit result. It is functioned as select signal s of the Multiplexer

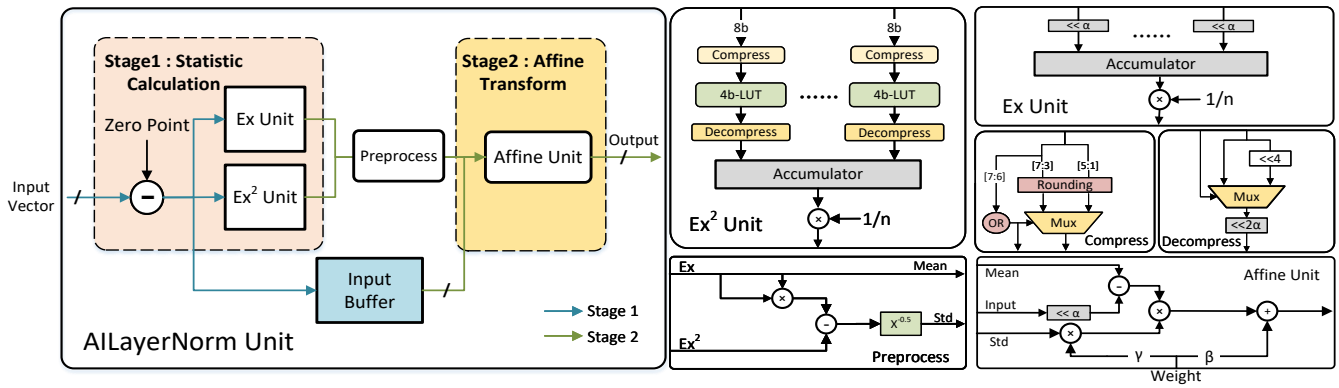

Fig. 5. Hardware design of AILayerNorm Unit

to decide the output, which equals with the formula that:

$$O = \frac{1.636 - 0.5s}{2} = \begin{cases} 0.818, & s = 0 \\ 0.568, & s = 1 \end{cases} \tag{17}$$

The result is then scaled using right shifter with respect to the subtraction result, thus obtaining the final output of E2Softmax. Compared with previous work, our divider utilizes the logarithm input produced by Log2Exp Unit, involving only a subtraction, a LOD, a two-way multiplex and two shift operations.

B. AILayerNorm Unit

We design AILayerNorm Unit characterized by low-precision statistic calculation to implement AILayerNorm algorithm. It is also divided into two stages to handle statistic calculation and affine transform. Ping-pong buffers are utilized to design a pipelined unit. 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 [31].

Statistic Calculation. Layer normalization computes mean and variance for normalization of inputs. Stage 1 is in charge of the statistic calculation. It consists of two subunits, namely Ex Unit and Ex^2 Unit. While prior works require high-precision calculation for mean and variance which introduce power and area inefficiency, for example INT32 for I-BERT [21], our design is capable of doing so in a low-precision manner. The 8-bit quantized inputs firstly subtract zero point and the results are sent to Ex, Ex^2 Unit and Input Buffer. In Ex Unit, inputs are scaled with power of two factors and then gathered to perform 12-bit reduction. In Ex^2 Unit, dynamic compression are firstly applied to obtain 4-bit inputs. Since square function with 4-bit inputs only have 16 possible outcomes, we implement square function with 16-entry Look-up table instead of multiplier to achieve better efficiency. Scaling operations are performed in the Decompress Unit as compensation for PTF and Dynamic Compression. Accumulator receives the outputs to perform reduction. After $\sum_{i=1}^{n} X_i$ and $\sum_{i=1}^{n} X_i^2$ are obtained, other operations such as power of -0.5 ($x^{-0.5}$) function are used in Preprocess Unit to generate parameters that Stage 2 requires. The $x^{-0.5}$ Unit is implemented using a LUT in our design which takes up little area and power consumption due to its small operation density.

Affine Transform. After Stage 1 finishes the calculation of statistic, Stage 2 starts to normalize inputs and rescale them with affine parameters γ, β. Catering to Algorithm 2, the Affine Unit fuses normalization and rescale operation into two multiplication and two addition. The first multiplier computes A with weights and Std. Inputs read from Input Buffer are scaled with PTF and subtracted with μ from Ex Unit. At last, multiplication and accumulation are performed to obtain outputs $Y = A \cdot X + B$. It should be noted that weights are also quantized to 8-bit integers.

V. Evaluation

A. Experiment Setting

Software setup. To validate our algorithm, we conducted exclusive experiments on CV and NLP tasks. For CV tasks, we selected ImageNet-1K [32] as our benchmark with different vision transformers, i.e., DeiT [33], Swin Transformer [34]. For NLP tasks, we conducted experiments on the GLUE benchmark [35] and SQuAD v1.1 dataset [36] with BERT-Base model [37]. FP32 and INT8 models were selected as our baseline. We chose INT8 model as baseline to demonstrate that SOLE can serve as a plugin for other compression method like quantization. We further applied SOLE algorithms as a replacement for the floating-point version to evaluate the impact of SOLE on accuracy, which was noted as FP32+SOLE and INT8+SOLE. INT8 models in Table I are obtained through post-training quantization following the settings in [22] while in Table II we performed quantization-aware fine-tuning with 8-bit weights and activation to get INT8 models [38].

Hardware setup. To evaluate the impact of SOLE on speedup and hardware efficency, we implemented SOLE hardware in RTL-Verilog and synthesized RTL using Synopsys Design Compiler on 28nm TSMC library with target 1ns clock period (1GHz). Power consumption was estimated at the typical

TABLE I
ALGORITHM EXPERIMENT ON IMAGENET-1K BENCHMARK.

Model	DeiT-T	DeiT-S	DeiT-B	Swin-T	Swin-S	Swin-B
FP32	72.21	79.85	81.85	81.38	83.23	83.60
FP32 + SOLE	**71.33**	**79.27**	**81.60**	**80.58**	**82.75**	**83.05**
INT8	71.72	79.25	81.42	80.42	82.84	83.14
INT8 + SOLE	**71.07**	**78.89**	**81.12**	**80.14**	**82.60**	**82.79**

TABLE II
ALGORITHM EXPERIMENT ON SQUAD AND GLUE TASKS WITH BERT-BASE MODEL.

Benchmark	CoLA	MRPC	SST-2	QQP	MNLI	QNLI	RTE	SQuAD
FP32	83.49	85.66	92.19	91.35	84.06	91.63	62.81	88.17
FP32 + SOLE	**83.56**	**86.38**	**91.52**	**91.00**	**84.09**	**90.91**	**62.62**	**87.57**
INT8	82.25	86.62	91.96	90.71	83.13	89.11	63.49	87.16
INT8 + SOLE	**82.72**	**85.74**	**91.41**	**90.39**	**82.89**	**88.70**	**65.39**	**86.45**

corner by PrimeTimePX, with the switching activity from VCD simulation traces. Vector size of SOLE hardware was set as 32 to match the MAC throughput of traditional accelerators.

For speedup evaluation, we chose NVIDIA 2080Ti GPU as our baseline. We scaled up our E2Softmax Unit and AILayerNorm unit resource by 32 times to make relatively fair comparisons. For area and power evaluation, apart from GPU, we compare SOLE with state-of-the-art custom hardware. Softermax [20] was selected as the baseline for Softmax while NN-LUT [26] was chosen as the baseline for LayerNorm [1]. We re-implemented these designs under the same setting with SOLE to extract power and area for fair comparison.

B. Algorithm Performance

Table I and Table II summarize the accuracy comparison on different transformer models, across CV and NLP tasks. Firstly, we compare the accuracy of FP32 and FP32+SOLE. The results show that SOLE incurs negligible accuracy drops. The worst accuracy drop is under 0.9% while the average accuracy drop is nearly 0.38% on different datasets and models. Secondly, we compare the performance of INT8 and INT8+SOLE. The results reveal that SOLE functions well in cooperation with INT8 quantization with the worst accuracy drop at 0.8% and the average drop at 0.2%, demonstrating that SOLE can be integrated with other model compression methods. Note that only matrix multiplication is done under INT8 format and other operations including Softmax and LayerNorm are still under FP32 format in INT8 models. Through integration with SOLE, Softmax and LayerNorm can be calculated with the input and output in 8-bit format. Another notable advantage of SOLE is that SOLE maintains inference accuracy without additional training or fine-tuning, therefore avoiding expensive training overhead and making it convenient to use.

[1]In NN-LUT, the implement of non-linear operations adopted NN-based LUT design while the other followed the method of I-BERT.

C. Speedup

Fig. 6 reveals the speedup of SOLE over 2080Ti GPU. We evaluate both stand along Softmax/LayerNorm operations on DeiT-Tiny, across different batchsize from 1 to 16. Token length is set as 785 corresponding to 448×448 image size. We also test end-to-end speedup of INT8 model over FP32 model with and without our HW/SW optimization.

As illustrated in Fig. 6(a), compared with GPU, SOLE achieves 29.3×-57.5× and 38.4×-86.8× speedup on Softmax and LayerNorm with average speedup at 36.2× and 61.3×, respectively. The speedup mainly comes from that we design highly specialized and pipelined datapath to reduce latency. In SOLE, ping-pong buffers and online normalization are utilized to maximize throughput. In Fig. 6(b), we can observe that INT8 model only achieve 1.10× to 1.28× speedup over FP32 model in GPU. It is due to the fact that INT8 quantization only alleviates overhead in matrix multiplication whereas non-linear operations like Softmax and LayerNorm occupy a significant portion of the inference. However, with the optimization of SOLE in non-linear operations, the burden can be significantly alleviated, resulting in 1.50× to 2.09× end-to-end speedup over FP32 model.

It has been demonstrated in section 5.2 that algorithms of SOLE can be integrated with INT8 quantization with negligible accuracy drop. The hardware of SOLE can also be easily integrated into modern accelerators since its 8-bit input and output are compatible with existing 8-bit integer vector MAC datapaths in accelerators and modern GPU tensor cores. Thus, comprehensive optimization can be achieved to get higher speedup. It is worth noticing that we simply use 32 E2Softmax Units and AILayerNorm Units to compare with GPU whose hardware resource still dominates ours. Whereas, SOLE offers scalable acceleration over Softmax, LayerNorm and end-to-end inference for transformer.

D. Hardware efficiency

Energy-efficiency. We first compare energy-efficiency of SOLE with Softermax, NN-LUT and 2080Ti GPU in Table III. The setting is the same as mentioned in section 5.3 and we report the average energy-efficiency in Table III.For in-depth evaluation, we compare the power consumption of the subunits, i.e. *Normalization Unit* and *Statistic Unit* mentioned in Section 4, which reveal computation power overhead. We also examine the complete hardware units, i.e. *Softmax Unit* and *LayerNorm Unit*, to measure both computation and memory overhead, thereby providing a comprehensive measure of power consumption. We find that SOLE offers 2.46× and 11.3× energy-efficiency improvements in *Normalization Unit* and *Statistic Unit*, resulting in hardware that are 3.04× and 3.86× more energy efficient for Softmax and LayerNorm respectively. The difference in energy-efficiency between subunits and complete hardware stems from the fact that power consumption in complete hardware is mainly determined by memory overhead, as discussed in Section 1. Overall , our design averagely deliver 3.04×, 4925× energy-efficiency improvements over Softermax and GPU in Softmax and 3.86×,

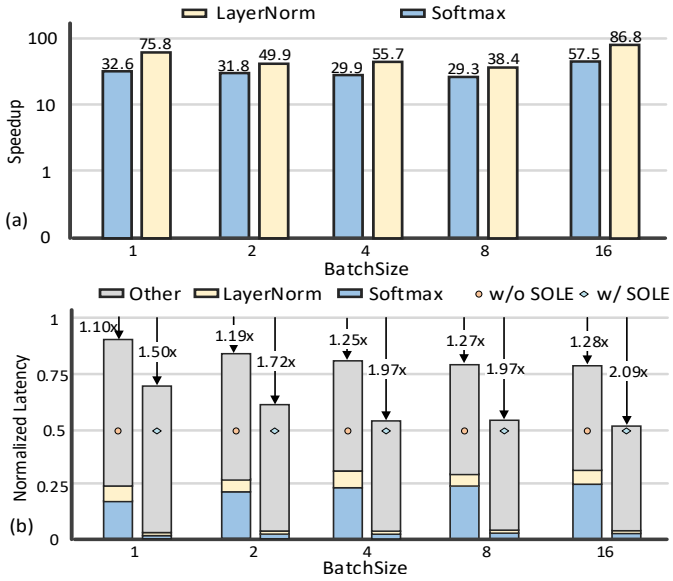

Fig. 6. (a) Speedup over GPU on Softmax and LayerNorm. (b) End-to-end speedup and latency breakdown, the latency is normalized with respect to FP32 implementation.

4259× energy-efficiency improvements over NN-LUT and GPU in LayerNorm.

Results have shown prominent energy-efficiency improvements in both computation and memory. For Softmax, the computational advantage is due to the use of the Log2Exp Unit and Approximate Log-based Divider in E2Softmax Unit. These units utilize shift and addition to implement non-linear operations while prior works need LUTs and multiplication. Additionally, log2 quantization on the exponent output allows intermediate results to be temporarily stored in 4-bits, greatly reducing the cost of memory access compared to the 16-bits required by Softermax. For LayerNorm, we adopt dynamic compression and low-precision statistic calculation in AILayerNorm to avoid INT32 multiplication in NN-LUT entirely, instead using only 16-entry LUTs and shift operation, resulting in significant energy savings for computation. Futhermore, intermediate memory access is saved to a large extent as SOLE quantizes the input data to 8-bit while prior works need to store 32-bit data. In general, SOLE significantly outperforms prior works in energy-efficiency.

Area-efficiency As shown in Table III, SOLE also exhibits enhancements in terms of area compared with the state-of-the-art designs. Specifically, SOLE achieves 2.89× and 3.79× area-efficiency improvements in *Normalization Unit* and *Statistic Unit* while offer 2.82× and 3.32× area-efficiency improvements for complete *Softmax Unit* and *LayerNorm Unit* at the same time. The area saving in computation subunits comes from the simplified implementation. In Softmax, SOLE only needs shifters and adders while Softermax relies on multipliers and LUTs. Similarly, SOLE utilizes 16-entry LUTs and shifters to replace 32-bit multipliers in LayerNorm. The entire calculation process is based on 8-bit input data so that the computational area burden is alleviated. Another aspect of the area-efficiency directly comes from the decreased size of buffer since SOLE reduces the bit-width of data stored in buffer from 16/32-bit to 4/8-bit, for Softmax and LayeNorm respectively.

TABLE III
SOLE COMPARISON TO SOFTERMAX, NN-LUT AND GPU IN ENERGY AND AREA.

Baseline		Energy-Efficiency	Area-Efficiency
Softermax [20]	Normalization Unit	2.46×	2.89×
	Softmax Unit	3.04×	2.82×
NN-LUT [26]	Statistic Unit	11.3×	3.79×
	LayerNorm Unit	3.86×	3.32×
2080Ti GPU	Softmax Unit	4925×	-
	LayerNorm Unit	4259×	-

VI. CONCLUSION

We propose SOLE, a hardware/software co-design method to enable efficient Softmax and LayerNorm inference in transformer. SOLE consists of E2Softmax and AILayerNorm which implement Softmax and LayerNorm with low-precision calculation and low bit-width data storage to achieve better efficiency. Experiments show that SOLE incurs negligible accuracy drop without additional training and can be integrated with orthogonal compression method like quantization. SOLE achieves orders of magnitude speedup and energy savings over GPU while offering 3.04×, 3.86× energy-efficiency improvements and 2.82×, 3.32× area-efficiency improvements over Softermax and NN-LUT, respectively.

REFERENCES

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

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

[3] X. Zhu, W. Su, L. Lu, B. Li, X. Wang, and J. Dai, "Deformable detr: Deformable transformers for end-to-end object detection," *arXiv preprint arXiv:2010.04159*, 2020.

[4] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever, "Zero-shot text-to-image generation," in *International Conference on Machine Learning*, pp. 8821–8831, PMLR, 2021.

[5] S. Zheng, J. Lu, H. Zhao, X. Zhu, Z. Luo, Y. Wang, Y. Fu, J. Feng, T. Xiang, P. H. Torr, *et al.*, "Rethinking semantic segmentation from a sequence-to-sequence perspective with transformers," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 6881–6890, 2021.

[6] B. Wang, K. Liu, and J. Zhao, "Inner attention based recurrent neural networks for answer selection," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1288–1297, 2016.

[7] S. Sukhbaatar, J. Weston, R. Fergus, *et al.*, "End-to-end memory networks," *Advances in neural information processing systems*, vol. 28, 2015.

[8] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015.

[9] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[10] M. Dehghani, J. Djolonga, B. Mustafa, P. Padlewski, J. Heek, J. Gilmer, A. Steiner, M. Caron, R. Geirhos, I. Alabdulmohsin, *et al.*, "Scaling vision transformers to 22 billion parameters," *arXiv preprint arXiv:2302.05442*, 2023.

[11] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[12] D. Driess, F. Xia, M. S. Sajjadi, C. Lynch, A. Chowdhery, B. Ichter, A. Wahid, J. Tompson, Q. Vuong, T. Yu, *et al.*, "Palm-e: An embodied multimodal language model," *arXiv preprint arXiv:2303.03378*, 2023.

[13] Z. Liu, Y. Wang, K. Han, W. Zhang, S. Ma, and W. Gao, "Post-training quantization for vision transformer," *Advances in Neural Information Processing Systems*, vol. 34, pp. 28092–28103, 2021.

[14] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.

[15] S. Shen, Z. Dong, J. Ye, L. Ma, Z. Yao, A. Gholami, M. W. Mahoney, and K. Keutzer, "Q-bert: Hessian based ultra low precision quantization of bert," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, pp. 8815–8821, 2020.

[16] Z. Qu, L. Liu, F. Tu, Z. Chen, Y. Ding, and Y. Xie, "Dota: detect and omit weak attentions for scalable transformer acceleration," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 14–26, 2022.

[17] H. Wang, Z. Zhang, and S. Han, "Spatten: Efficient sparse attention architecture with cascade token and head pruning," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 97–110, IEEE, 2021.

[18] L. Lu, Y. Jin, H. Bi, Z. Luo, P. Li, T. Wang, and Y. Liang, "Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 977–991, 2021.

[19] T. J. Ham, S. J. Jung, S. Kim, Y. H. Oh, Y. Park, Y. Song, J.-H. Park, S. Lee, K. Park, J. W. Lee, *et al.*, "Aˆ 3: Accelerating attention mechanisms in neural networks with approximation," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 328–341, IEEE, 2020.

[20] J. R. Stevens, R. Venkatesan, S. Dai, B. Khailany, and A. Raghunathan, "Softermax: Hardware/software co-design of an efficient softmax for transformers," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 469–474, 2021.

[21] S. Kim, A. Gholami, Z. Yao, M. W. Mahoney, and K. Keutzer, "I-bert: Integer-only bert quantization," in *International conference on machine learning*, pp. 5506–5518, PMLR, 2021.

[22] Y. Lin, T. Zhang, P. Sun, Z. Li, and S. Zhou, "Fq-vit: Post-training quantization for fully quantized vision transformer," in *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pp. 1173–1179, 2022.

[23] S. Kim, C. Hooper, T. Wattanawong, M. Kang, R. Yan, H. Genc, G. Dinh, Q. Huang, K. Keutzer, M. W. Mahoney, *et al.*, "Full stack optimization of transformer inference: a survey," *arXiv preprint arXiv:2302.14017*, 2023.

[24] M. Wang, S. Lu, D. Zhu, J. Lin, and 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)*, pp. 223–226, IEEE, 2018.

[25] D. Wu, J. Li, S. Behroozi, Y. Kim, and J. San Miguel, "Uno: Virtualizing and unifying nonlinear operations for emerging neural networks," in *2021 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 1–6, IEEE, 2021.

[26] J. Yu, J. Park, S. Park, M. Kim, S. Lee, D. H. Lee, and J. Choi, "Nn-lut: neural approximation of non-linear operations for efficient transformer inference," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pp. 577–582, 2022.

[27] J. Cai, M. Takemoto, and H. Nakajo, "A deep look into logarithmic quantization of model parameters in neural networks," in *Proceedings of the 10th International Conference on Advances in Information Technology*, pp. 1–8, 2018.

[28] J. N. Mitchell, "Computer multiplication and division using binary logarithms," *IRE Transactions on Electronic Computers*, no. 4, pp. 512–517, 1962.

[29] M. Milakov and N. Gimelshein, "Online normalizer calculation for softmax," *arXiv preprint arXiv:1805.02867*, 2018.

[30] S. Lu, M. Wang, S. Liang, J. Lin, and Z. Wang, "Hardware accelerator for multi-head attention and position-wise feed-forward in the transformer," in *2020 IEEE 33rd International System-on-Chip Conference (SOCC)*, pp. 84–89, IEEE, 2020.

[31] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv preprint arXiv:1607.06450*, 2016.

[32] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009.

[33] H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles, and H. Jégou, "Training data-efficient image transformers distillation through attention," 2021.

[34] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," 2021.

[35] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "Glue: A multi-task benchmark and analysis platform for natural language understanding," *arXiv preprint arXiv:1804.07461*, 2018.

[36] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100,000+ questions for machine comprehension of text," *arXiv preprint arXiv:1606.05250*, 2016.

[37] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019.

[38] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, "Integer quantization for deep learning inference: Principles and empirical evaluation," *arXiv preprint arXiv:2004.09602*, 2020.