Softmax Acceleration with Adaptive Numeric Format for both Training and Inference

Tianhua Xia
Independent Researcher
San Diego, USA
bitxiatianhua@gmail.com

Sai Qian Zhang
New York University
New York, USA
sai.zhang@nyu.edu

ABSTRACT

The attention mechanism is a pivotal element within the Transformer architecture, making a substantial contribution to its exceptional performance. Within this attention mechanism, Softmax is an imperative component that enables the model to assess the degree of correlation between various segments of the input. Yet, prior research has shown that Softmax operations can significantly increase processing latency and energy consumption in the Transformer network due to their internal nonlinear operations and data dependencies. In this work, we proposed *Hyft*, a hardware efficient floating point Softmax accelerator for both training and inference. Hyft aims to reduce the implementation cost of different nonlinear arithmetic operations by adaptively converting intermediate results into the most suitable numeric format for each specific operation, leading to reconfigurable accelerator with hybrid numeric format. The evaluation results highlight that Hyft achieves a remarkable 15× reduction in hardware resource utilization and a 20× reduction in processing latency, all while maintaining a negligible impact on Transformer accuracy.

KEYWORDS

Hardware accelerator, Softmax, Transformer

1 INTRODUCTION

Ever since the debut of the Transformer [22], attention-based deep neural networks (DNNs) have achieved remarkable performance across various tasks in different fields [5, 6, 21]. The attention mechanism in the Transformer plays a crucial role in enabling the model to capture and model complex relationships and dependencies within input sequences, greatly enhancing the accuracy on different tasks.

Nonetheless, the superior accuracy Transformer is accompanied by increased computational and memory requirements. The model sizes have reached unprecedented magnitudes and continue to grow. On top of the model size, attention computation consists of a unique mix of linear matrix multiplication and non-linear operations such as Softmax, layer normalization, and GeLU. Unlike other DNN architectures like Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), which are primarily dominated by matrix multiplication, Transformers allocate a substantial portion of their runtime to attention operations. In particular, recent research has shown that the Softmax operation can consume a significant portion of runtime in the Transformer (>30% [20]), especially as the length of input sequences increases. This impact is not limited to Transformer model for NLP tasks. It also affects variants like Vision Transformers (ViT) [6], where Softmax computations become increasingly dominant, especially when computational resources are limited, as observed in a study by Dass et al. [4].

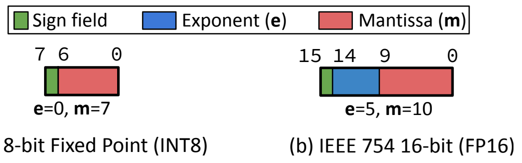

Figure 1: Fixed and floating-point number formats.

Moreover, previous research has revealed a significant distinction: unlike other DNNs that exhibit insensitivity to Softmax computation, the performance of Transformers is notably influenced by the Softmax output [13, 23]. This sensitivity implies that many current approximation algorithms for Softmax, such as those that reduce the precision of Softmax operands [19, 25], can result in a serious accuracy degradation on Transformers.

Accelerating the Softmax operation presents a significant challenge. Specifically, the complexity of hardware implementation for Softmax arises from two main factors. Firstly, the exponential and division operations involved in Softmax are computationally intensive and resource-demanding. Secondly, the data dependency inherent in Softmax makes it difficult to pipeline and parallelize the operations effectively, resulting in increased latency and energy consumption. Furthermore, to maintain compatibility with the other computations within attention-based models, it is preferable for the Softmax block to operate with floating-point input and output formats.

In addition to inference operations for Transformers, there is an increasing demand for fine-tuning Transformer models, particularly Transformer-based Large Language Models (LLMs), to dynamically adapt to unseen user datasets and tasks [9, 17]. This highlights the importance of efficiently performing Softmax training.

In this study, we introduce *Hyft*, a hardware-efficient Softmax accelerator designed for both training and inference. Hyft is designed to reduce the implementation cost of various nonlinear arithmetic operations by dynamically converting intermediate results into the most appropriate numeric format for each specific operation. Additionally, Hyft offers a high degree of reconfigurability, allowing users to adjust input and output precision to achieve an optimal balance between hardware performance and accuracy. Moreover, to facilitate the backward propagation operations for Softmax, Hyft strategically reuses the hardware blocks for forward propagation. This effectively increases the hardware utilization rate, thereby enhancing the efficiency of training operations.

- To the best of our knowledge, Hyft marks the first Softmax accelerator capable of efficiently supporting both Softmax inference and training operations in Transformers.
- Hyft leverages a hybrid approach, combining fixed and floating-point formats, to streamline the exponential, division, and multiplication operations involved in both the forward and backward propagation of Softmax. This innovative approach results in superior hardware performance, reducing processing latency and energy consumption.
- Hyft offers a high degree of configurability, allowing users to choose the optimal numeric format for representing intermediate results. This flexibility enables a superior trade-off between hardware performance and accuracy performance.
- The evaluation results highlight that Hyft achieves a remarkable 15× reduction in hardware resource utilization and a 20× reduction in processing latency, all while maintaining a negligible impact on Transformer accuracy.

2 BACKGROUND AND MOTIVATION

2.1 Numeric Format

Numerous numeric formats have been investigated to alleviate the computational demands in DNN inference and training, as documented in previous studies [1–3, 8, 10–12, 14–16, 26–28, 30]. These formats can broadly be classified into two main categories: fixed-point formats and floating-point formats. Fixed-point formats lack an exponent field, which results in a reduced dynamic range representation but simplifies the hardware implementation. Figure 1 (a) show an example on fixed-point representation with 8 bits.

On the contrary, floating-point formats offer a wider dynamic range when compared to fixed-point formats. While the conventional format for DNN training is IEEE 754 32-bit floating point or FP32, there has been exploration into other options such as 16-bit floating point (FP16) and various custom floating-point formats like bfloat16 and TensorFloat. Figure 1 (b) displays the IEEE 754 half-precision floating-point format (FP16). The value of a floating-point number can be calculated using Equation 1:

$$x = (-1)^{S_x} 2^{e_x} (1 + m_x) \tag{1}$$

where S_x represents the sign bit, $e_x = E_x - B$ denotes the value of the 5-bit exponent as B represents the bias, and $m_x = M_x/2^L$ stands for the value of the mantissa bits. In the IEEE 754 half-precision format, B is set to 15, and L is set to 10. Compared with fixed-point format, multiplication with floating-point formats is considerably more computationally intensive than with fixed-point formats. This is due to the fact that multiplying two floating-point numbers not only involves the multiplication of their high-precision mantissas but also requires addition operations between their exponent fields.

2.2 Computation in Transformer

A Transformer is constructed as a stack of Transformer encoders and decoders, with each encoder or decoder comprising two fundamental components: a Self-Attention (SA) block and a Multi-Layer Perceptron (MLP) block. During the inference process, the input H of the SA block is first multiplied with three weight matrices W_Q, W_K, and W_V, yielding the outputs referred to as query (A_Q), key (A_K), and value (A_V), respectively. The resulting A_Q and A_K, in combination with A_V, will then undergo multiplication, Softmax, and residual addition to generate the SA output H_{out} in equation 2, where d_{head} indicate the number of feature dimensions in multi-head attention mechanism [22].

$$H_{out} = W_o \times (A_v \times Softmax(\frac{A_Q \times A_K^T}{\sqrt{d_{head}}}) \tag{2}$$

The SA output will then be forwarded to the MLP blocks for further processing. The feed forward block consists of a stack of fully connected (FC) layers together with some intermediate activation function. The output of the last decoder layer will be sent to a linear layer, which then generates the final results.

2.3 Softmax Function

Softmax operation has been widely utilized in the Transformer architecture. Specifically, the Softmax function takes a vector $z = [z_1, z_2, \cdots, z_N]^T$ and generates an output $s = [s_1, s_2, \cdots, s_N]^T$, both have a length of N. It is defined as follows:

$$s_i = \frac{e^{z_i}}{\Sigma_{j=0}^{N-1} e^{z_j}} \quad For\ i = 1, 2, \cdots, N \tag{3}$$

direct calculations may encounter numeric issues due to limitations of the 32-bit single-precision floating-point format further leading to a NaN (Not a Number) outputs. To ensure numeric stability, the Softmax function's inputs are typically calibrated by subtracting their mean from each input, as illustrated below:

$$s_i = \frac{e^{z_i - z_{max}}}{\Sigma_{j=0}^{N-1} e^{z_j - z_{max}}} \tag{4}$$

This operation results in a stable input with a consistent range, without altering the final results. Additionally, the back propagation computation of Softmax is defined as follows:

$$\frac{\mathrm{d}s}{\mathrm{d}z} = diag(s) - ss^T \tag{5}$$

where $diag(s)$ represents a square matrix with s as its diagonal elements, while all other elements are set to zero. An example is shown below when $N = 4$:

$$\frac{\mathrm{d}s}{\mathrm{d}z} = \begin{bmatrix} s_1 - s_1^2 & -s_1 \cdot s_2 & -s_1 \cdot s_3 & -s_1 \cdot s_4 \\ -s_2 \cdot s_1 & s_2 - s_2^2 & -s_2 \cdot s_3 & -s_2 \cdot s_4 \\ -s_3 \cdot s_1 & -s_3 \cdot s_2 & s_3 - s_3^2 & -s_3 \cdot s_4 \\ -s_4 \cdot s_1 & -s_4 \cdot s_2 & -s_4 \cdot s_3 & s_4 - s_4^2 \end{bmatrix}$$

2.4 Related Work

Several hardware-efficient implementations of the Softmax have been proposed in the prior work. Most of these approaches have been tested in CNN or Multi-Layer Perceptrons (MLPs), where the Softmax operation is typically required only in the final layer of the DNN.

Numerous approximation algorithms have been proposed for the Softmax operation, such as Taylor expansion [7] and division approximation [25]. Additionally, in works like [20] and [29], authors have simplified Softmax implementation by replacing the base-e Softmax function with a base-2 version. However, it is important to note that all these previous solutions necessitate fine-tuning the DNN to account for the approximation error. Furthermore, all

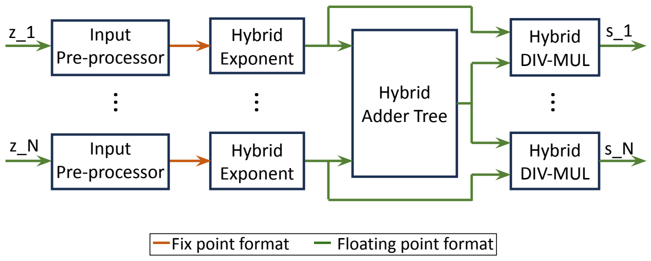

Figure 2: Forward propagation data path of Hyft.

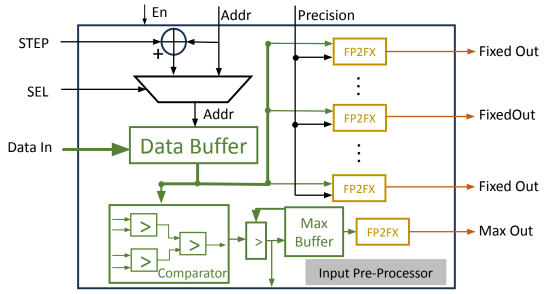

Figure 3: Hardware design for Input Pre-processor.

of these approaches employ fixed-point or low-precision floating-point formats for value storage, which deviates from the numeric format used in the remaining layers of the network. Consequently, an additional converter is needed to convert the data to ensure compatibility, resulting in extra implementation overhead. Additionally, it is important to note that none of the aforementioned methods support backpropagation operations for the Softmax layer in Transformer. While [29] does support training operations, it necessitates fine-tuning the resulting CNN to mitigate the significant approximation error introduced by the base-2 Softmax approximation. This makes it unsuitable for implementation in Transformers.

In contrast, Hyft operates using a half-precision mode to accommodate FP16 input and output, and it can be readily configured to function in full-precision mode, supporting input and output in FP32. Hyft can be seamlessly integrated into any existing DNNs without the need for format conversion.

3 HYFT ARCHITECTURE

The overall computation flow of Hyft during the forward propagation is illustrated in Figure 2. The Hyft system comprises four core components: input pre-processor, hybrid exponent unit, hybrid adder tree, and hybrid multiplication/division unit.

Hyft supports input and output data in either FP16 or FP32 and dynamically converts intermediate data to the most suitable number format for the current arithmetic operation. To achieve this, we utilize a fixed-point format for internal steps to streamline linear addition and subtraction operations. Additionally, we employ floating-point representation to facilitate exponential, multiplication, and division operations within the logarithmic space. The precision of the intermediate results can be fully configurable. In Figure 2, you can observe the floating-point data paths highlighted in green and the fixed-point data paths highlighted in red.

Given the input in floating-point format $\boldsymbol{z} = [z_1, z_2, \cdots, z_N]^T$, the input pre-processor operates by identifying the maximum value, denoted as z_{max}, within the input vector. To simplify the later input subtraction operation, the each element z_i of $\boldsymbol{z}$ and z_{max} will be converted to fixed-point format with configurable precision in the pre-processor. The hybrid exponent unit subtracts z_{max} from each element in the input vector, producing $\boldsymbol{z'} = [z'_1, z'_2, \cdots, z'_N]^T$, where $z'_i = z_i - z_{max}$. z'_i expressed in fixed point format will benefit the hybrid exponent unit by simplifying the integer fraction split step, which will be described in Section 3.2. The Hybrid exponent unit processes input data in fixed-point format and produces output data in floating-point format. This approach eliminates the need for a shift operation, which is typically required in traditional fixed-point exponential accelerators. The hybrid adder tree converts the floating-point inputs to fixed-point format in order to simplify the addition operations. After performing the additions, it then converts the results back to floating-point format. This approach helps streamline the addition process while maintaining compatibility with floating-point representations. The division unit performs floating point division and generates the Softmax results in floating point.

During our FPGA implementation, we observe that without the numeric format conversion among the blocks, the shift operations in the hybrid exponent units will become the critical path for Softmax computation. By strategically employing hybrid formats at various stages, we can significantly reduce the number of shift operations required, thus optimizing the overall performance. Next, we will introduce each component in detail.

3.1 Parameterized Input Pre-Processr

The parameterized input pre-processor serves two essential functions. Firstly, it searches the maximum value of the input vector to be used in hybrid exponential unit to prevent Softmax calculation overflow issues. Secondly, it converts floating-point inputs to fixed-point format to align with the input format of hybrid exponent unit (Section 3.2), simplifying hardware implementation as discussed later. The architecture of the input pre-processor in depicted Figure 3.

To accelerate the search for the maximum value among z_i, we leverage on the resilience of the Transformer architecture to computational noise. Specifically, a configurable parameter, *STEP*, is taken by input pre-processor as a input, which governs the number of data points that enter the comparator block for the max value search. When set to 1, the address of the next data to be compared is the current address incremented by 1, causing Hyft to utilize all the data for the max search. In contrast, when *STEP* is set to 2, the address of the next data is the current address incremented by 2, leading Hyft to use every other data point for the max search. We show in the evaluation section that most of the tasks can be executed with an accelerated maximum searching process without any accuracy degradation.

In parallel to max search block, there are floating point to fixed point converters (FP2FX) in the input pre-processor which converts the floating point inputs and their max value to fixed point format. The fixed point format outputs will benefit the hybrid exponent unit in Section 3.2. The input preprocessor in Hyft also accepts

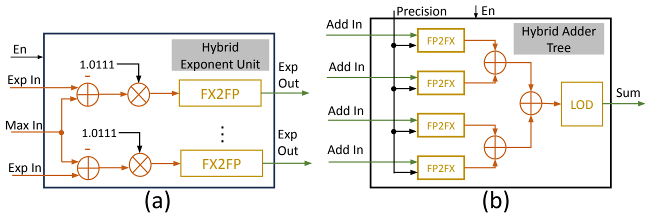

Figure 4: (a) Hybrid Exponent Unit and (b) Hybrid Adder Tree. The floating-point and fixed-point data paths are highlighted in green and red, respectively. The control signal are highlighted in black.

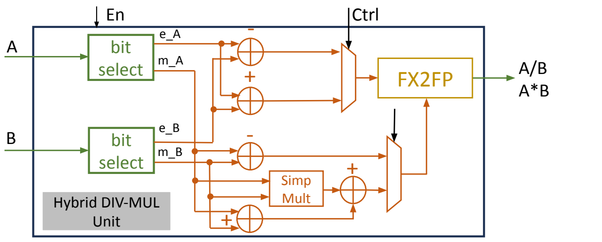

Figure 5: Hardware design of Hybrid division-multiplication unit.

an additional parameter, denoted as *Precision*, which determines the number of bits allocated for the decimal part in the converted fixed-point format.

3.2 Hybrid Exponent Unit

Our hybrid exponent unit is designed to accept fixed-point numbers z and z_{max} as input and produce results $e^{z-z_{max}}$ in floating-point format. Utilizing fixed-point input simplifies the input subtraction, the constant multiplication and the separation of integer and decimal components within our exponential unit. The use of floating-point output eliminates the delays associated with shift operations found in traditional fixed-point exponential accelerators. Importantly, this design also enhances the performance of the division and multiplication units within our Softmax implementation.

Exponent operations $e^{z'}$ involve numerous multiplications that consume a large amount of computational resources and lead to increased processing latency. To eliminate the computations associated with the Euler number, we transform the exponential operation as follows:

$$e^{z'} = 2^{z' log_2 e} = 2^{u+v} \tag{6}$$

where u and v represent the integer and fractional parts of $z' log_2 e$ value. We approximate $log_2 e$ as 1.0111_2, therefore $z' log_2 e$ can be approximated as $z' + (z' >> 2) + (z' >> 3) + (z' >> 4)$. We further apply Booth encoding, reducing the number of shift operations required, resulting in the approximation $z' log_2 e \approx z' + (z' >> 1) - (z' >> 4)$. This shift-and-add operation is efficiently computed using an adder trees. Leveraging the inherent characteristics of the fixed-point format, it becomes straightforward to extract both the integer and decimal parts of $z' \cdot log_2 e$.

Due to the input normalization described in Equation 4, the input to the exponential unit is consistently less than or equal to zero. Consequently, the values represented by u, v in Equation 6 are always less than or equal to zero as well. Building upon the insights presented in [13], we can further streamline the computation using the Taylor approximation, which is shown as follows:

$$e^{z'} = 2^{u+v} \approx 2^u(1 + v/2) \qquad u \leq 0, -1 < v \leq 0 \tag{7}$$

To be precise, the current intermediate results u and v are less than or equal to zero, but floating point format requires mantissa being a positive number. A simple FX2FP block is used to convert the exponent and mantissa fields for the floating-point representation $e^{z'}_{float}$ of $e^{z'}$ using the following formula:

$$e^{z'} \approx 2^u(1 + v/2) = 2^{u-1}(1 + (1 + v)) \tag{8}$$

where the exponent field and mantissa field of $e^{z'}_{float}$ is $u - 1$ and $1 + v$, respectively. Since $e^{z'}$ is always positive, the sign field is set to 0.

3.3 Hybrid Adder Tree

After computing the outputs $e^{z'}_{float}$ from the hybrid exponent unit, the subsequent step involves summing these values together. This summation is essential for calculating the denominator in Equation 4. However, it's worth noting that the summation operation among floating-point numbers can be computationally expensive. This is because it necessitates shifting the mantissas within each operand to align the exponent fields before summing the mantissas. The shift operation can consume a significant amount of hardware resources.

To reduce this computational cost, a floating point to fixed point converter (FP2FX) converts $e^{z'}_{float}$ to $e^{z'}_{fixed}$ to perform the summation in fixed-point format. Furthermore, since the inputs z' are non-positive, the values of $e^{z'}$ fall within the range of $(0, 1]$. As a result, the sign bit in the fixed-point representation $e^{z'}_{fixed}$ is unnecessary, and only one bit for the integer part is required. We also implements a fractional bits of $e^{z'}_{fixed}$ to be fully configurable to enable an better flexibility between hardware performance and accuracy performance. The resultant summation $\sum_{j=0}^{N-1} e^{z'_j}$ is then converted back to floating point format using a leading one detector (LOD) to facilitate the operation of the hybrid division-multiplication unit (Hybrid DIV-MUL unit in Figure 2), which is described next.

3.4 Division/Multiplication Unit

Following the generation of the denominator and numerator in Equation 4, the next step involves computing the division between them. Division is a computationally intensive operation, and several approaches have been proposed in prior research to mitigate the associated computational cost. For instance, when simplifying the Softmax function in CNNs, the authors of [19] and [13] approximated the divisor as a power of two integer, effectively substituting the division operation with a shift operation. However, this approximation introduces errors in Softmax calculations, resulting in reduced accuracy for Transformers. An alternative method by Vasyltsov et al. [23] involved the use of a lookup table (LUT) to optimize division computation, but it relies on prior knowledge of the input distribution. In our study, we implement the log-subtract

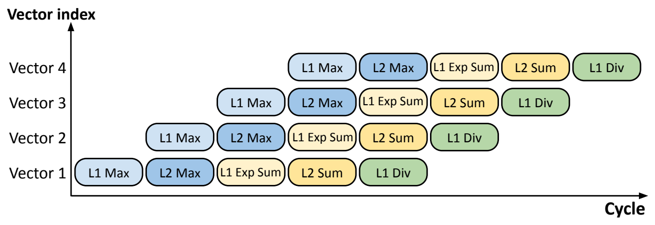

Figure 6: Timing diagram of pipelined Hyft. There are two layers of Hyfts ($L1$, $L2$) in a tree structure to accelerate max search and exponential summation. Division of each vector element is independent, so only L1 dividers are used.

strategy as proposed in [7]. However, we deviate from using fixed-point format, which can result in high computational costs for division operations. Instead, we opt for floating-point format, effectively reducing quantization errors and mitigating hardware costs. Given that both numerator and denominators are represented in floating-point format, let e_a, m_a and e_b, m_b represent the exponent and mantissa fields of the numerator a and b, respectively. The division $\frac{a}{b}$ can be detailed as follows:

$$\frac{a}{b} = 2^{e_a}(1+m_a)/2^{e_b}(1+m_b) = 2^{e_a-e_b+log_2(1+m_a)-log_2(1+m_b)}$$
$$\approx 2^{e_a-e_b+m_a-m_b} \approx 2^{e_a-e_b}(1+m_a-m_b) \tag{9}$$

where we have employed the Taylor approximation, specifically $log_2(1+x) \approx x$ for x in the range of [0,1], to simplify the computation. While [23] demonstrated that logarithm approximation led to decreased accuracy in the Transformer model, this was primarily due to the limited precision and range that an 8-bit fixed-point format can represent, resulting in the accumulation of quantization and approximation errors. In our system, the utilization of a 16-bit floating-point format can effectively address this limitation and result in negligible accuracy loss.

In addition, implementing the log-subtract operation in floating-point format is more efficient than using fixed-point representation. In the fixed-point approach [7], it requires two LODs and three shifters to convert the input to a power-of-2 format, perform subtraction, and then convert back to fixed-point format. In contrast, the floating-point system requires no shifters or LODs for Equation 9 since the input and output are already in power-of-2 format, reducing resource usage and latency.

3.5 Softmax Backpropagation For Training

The computation of the Softmax backward propagation (Equation 5) requires the calculation of ss^T, where s represents the results obtained from forward propagation. As each element of s is expressed in floating-point format, this computation involves the multiplication of two floating-point numbers as follows:

$$a \times b = 2^{e_a}(1+m_a) \times 2^{e_b}(1+m_b) = 2^{e_a+e_b}(1+m_a+m_b+m_a m_b) \tag{10}$$

To handle this multiplication, we employ the division/multiplication unit detailed in Section 3.4. Specifically, in comparison to Equation 9, Equation 10 introduces only one additional operation, which is $m_a m_b$. This operation can be efficiently processed using a fixed-point multiplier. For the remaining operations, such as exponentiation and mantissa addition, we can utilize the existing hardware within the division/multiplication unit for processing. To further save on computational resources, rather than utilizing a full-range multiplier, we opt to take advantage of only half the range of one of the multiplicands. This approach leads to a 50% reduction in hardware compared to a full-range multiplier without any accuracy degradation. With this method, the division/multiplication unit can handle the computations associated with both forward and backward propagations, leading to a significant hardware utilization rate.

3.6 Hyft Vector Processor

As described in the earlier sections, Softmax computation typically involves three stages of data processing: (1) maximum searching, (2) exponentiation and summation, and (3) division. These three stages cannot be pipelined for a single vector operation. However, unlike traditional CNNs that execute Softmax only in the last layer, Transformers require Softmax computation for multiple input vectors z within an attention block. This characteristic opens up the possibility for a vector-wise pipeline, which has not been explored in previous work. Hyft is divided by the 3 stages of Softmax and each stage will process different vectors for better hardware utilization and throughput enhancement. Hyft pipeline diagram is shown in Figure 6.

4 EXPERIMENTAL RESULTS

In this section, we initially evaluate the accuracy performance of Hyft in Section 4.1 across various datasets. Subsequently, we examine the hardware implementation cost in Section 4.2.

4.1 Accuracy Evaluation

To enable efficient Softmax computation, we have incorporated several approximations, as elaborated in Section 3. In this section, we explore the impact of these approximations on the training and inference accuracy of Hyft. Specifically, we emulate the software behavior of Hyft by implementing it in PyTorch. We then fine-tune a BERT model [5] on various downstream tasks specified in the GLUE [24] and SQuAD [18] benchmarks. Subsequently, we replace the Softmax layer in the resulting model with the customized Softmax implementation using Hyft and record the changes in accuracy performance. Specifically, we considered two configurations of Hyft: Hyft16 and Hyft32. Hyft16 and Hyft32 accept inputs and produce outputs in FP16 and FP32 formats, respectively. We also report the BERT accuracy with original Softmax implementation.

The results are presented in Table 1. We notice there is a negligible difference on the accuries with the BERT model with original Softmax implementation, indicating that both versions of Hyft can integrated well with the Transformer.

Moreover, to demonstrate the robustness of Hyft for Transformer training, we assess its accuracy performance for fine-tuning the BERT model. To be specific, we substituted the original Softmax layers in BERT with our customized Softmax implementation using Hyft16 or Hyft32. We then fine-tuned BERT on the GLUE and

Tasks	SQuAD 1.1	MRPC	CoLA	RTE	SST2	QNLI
Original	88.32%	86.03%	53.31%	67.02%	93.24%	91.10%
Hyft32	88.31%	86.12%	53.28%	67.00%	93.29%	91.02%
Hyft16	88.37%	85.97%	53.40%	66.91%	93.21%	91.08%
[29]	81.96%	80.04%	44.43%	53.89%	82.92%	84.60%
[13]	88.30%	83.57%	53.23%	66.72%	92.08%	90.01%

Table 1: Accuracy results of Hyft on GLUE and SQuAD benchmarks. We simulate the prior works [13, 29] over BERT for comparison.

Tasks	SQuAD 1.1	MRPC	CoLA	RTE	SST2	QNLI
Original	88.32%	86.03%	53.31%	67.02%	93.24%	91.10%
Hyft32	88.26%	86.06%	53.42%	67.08%	93.13%	91.19%
Hyft16	88.30%	86.03%	53.30%	67.10%	93.16%	91.21%

Table 2: Training Accuracy of Hyft on GLUE and SQuAD benchmarks

SQuAD datasets for 3 epochs, employing an initial learning rate of 5e-5 and a batch size of 32.

The resultant training accuracies are presented in Table 2. Similarly, we observed that Hyft did not have any noticeable impact on training accuracy, showing that both Hyft16 and Hyft32 can be integrated to Transformer training and inference.

4.2 Hardware Evaluation

We implemented Hyft on the Xilinx xc7z030 FPGA chip utilizing the Xilinx Vivado Design Suite. We then proceeded to evaluate the hardware performance and compare it with other FPGA implementations of Softmax accelerators.

Regarding the Softmax operation, we fixed the length of the input vector z to be 8 and assessed all the designs based on the following criteria: 1. Processing latency of the Softmax operation. 2. FPGA resource utilization, measured in terms of Look-Up Tables (LUT) and Flip-flops (FF). 3. The maximum achievable operating frequency (F_{max}). 4. FOM, a metric introduced in [13], defined as follows:

$$FOM = \frac{F_{max} \times N \times W}{LUT + FF} \tag{11}$$

where W denotes the precision of the z_i. FOM serves as a comprehensive indicator that assesses the overall performance of Softmax accelerators. A higher FOM value signifies better performance.

We compared Hyft with multiple baseline methods, the results are shown in Table 3. Among these baselines, Xilinx FP is a 32-bit floating-point Softmax engine implemented using Xilinx IPs reported in [13]. In contrast, Hyft not only consumes 15× fewer resources and operates at a 40% higher frequency compared to the Xilinx FP design, but also surpasses the performance of previous fixed-point accelerators. The primary reason for this superior performance is the adaptive precision selection mechanism supported in Hyft. As previously explained, shift operations within the float-point addition/subtraction constitute the critical path in a Softmax accelerator. Hyft optimizes the latency of each shift operation by dynamically controlling the precision, enabling it to support the highest operating frequency and facilitating deep pipelining for enhanced throughput. Furthermore, Hyft minimizes the number of shift operations by reducing the occurrences of floating-point addition and subtraction. This reduction is accomplished through adaptive input conversion to fixed-point format. Consequently, this approach results in substantial resource savings. Finally, the pipelining scheme described in Section 3.6 greatly reduces the processing latency and enhances the Hyft throughput.

Methods	Config. N, W	Format	Resource (LUT, FF)	F_{max} (MHz)	Latency (ns)	FOM
APCCAS'18[25]	8 16-bit	Fixed	2564,2794	436	NA	10.416
ISCAS'20[7]	1 16-bit	Fixed	2229,224	154	NA	1.004
TCAS-I'22[29]	10 16-bit	Fixed	1476,698	500	NA	36.798
ISCAS'23 FP[13]	8 16-bit	Floating	$\sim 1200, \sim 600$	476	14.7	33.849
Xilinx FP[13]	8 32-bit	Floating	13254,18664	435	232.3	3.488
Hyft16	8 16-bit	Floating	1072, 824	625	12.4	42.194
Hyft32	8 32-bit	Floating	2399, 1528	526	19	34.290

Table 3: Hardware evaluation of Hyft.

While [29] achieves a similar FOM as Hyft, it is important to note that their Softmax implementation is tailored for CNNs and relies entirely on fixed-point numbers. This approximation is acceptable for CNNs, where Softmax is primarily used in the final layer and insensitive to the accuracy, and additional fine-tuning is required over CNN to mitigate the accuracy. Nevertheless, when integrated into Transformer models, this approach can lead to notable accuracy degradation, as demonstrated in Table 1. Compared with [13], Hyft16 achieves a slightly better latency and FOM. However, [13] adopts a very aggressive approximation for Softmax, further leading to a larger accuracy degradation than Hyft, as shown in Table 1.

5 CONCLUSION

In this study, we introduced Hyft, a hardware-efficient floating-point Softmax accelerator designed for both training and inference purposes. Our evaluation results demonstrate that Hyft achieves outstanding outcomes, including a remarkable reduction in hardware resource utilization and processing latency while preserving Transformer accuracy to a negligible extent.

REFERENCES

[1] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. 2018. Scalable methods for 8-bit training of neural networks, In NeurIPS. *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, 5151–5159.

[2] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2014. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024* (2014).

[3] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*. 3123–3131.

[4] Jyotikrishna Dass, Shang Wu, Huihong Shi, Chaojian Li, Zhifan Ye, Zhongfeng Wang, and Yingyan Lin. 2022. ViTALiTy: Unifying Low-rank and Sparse Approximation for Vision Transformer Acceleration with a Linear Taylor Attention. arXiv:2211.05109 [cs.CV]

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL]

[6] Alexey Dosovitskiy et al. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. arXiv:2010.11929 [cs.CV]

[7] Yue Gao, Weiqiang Liu, and Fabrizio Lombardi. 2020. Design and Implementation of an Approximate Softmax Layer for Deep Neural Networks. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–5.

[8] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *International Conference on Machine Learning*. 1737–1746.

[9] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685* (2021).

[10] Itay Hubara et al. 2017. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research* 18, 1 (2017), 6869–6898.

[11] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2704–2713.

[12] Supriya Kapur, Asit Mishra, and Debbie Marr. 2017. Low precision rnns: Quantizing rnns without losing accuracy. *arXiv preprint arXiv:1710.07706* (2017).

[13] Nazim Altar Koca, Anh Tuan Do, and Chip-Hong Chang. 2023. Hardware-efficient Softmax Approximation for Self-Attention Networks. In *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–5.

[14] HT Kung, Bradley McDanel, and Sai Qian Zhang. 2019. Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 821–834.

[15] Bradley McDanel, Sai Qian Zhang, HT Kung, and Xin Dong. 2019. Full-stack optimization for accelerating cnns using powers-of-two weights with fpga validation. In *Proceedings of the ACM International Conference on Supercomputing*. 449–460.

[16] Eunhyeok Park, Junwhan Ahn, and Sungjoo Yoo. 2017. Weighted-entropy-based quantization for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 5456–5464.

[17] Jonas Pfeiffer, Aishwarya Kamath, Andreas Rücklé, Kyunghyun Cho, and Iryna Gurevych. 2020. AdapterFusion: Non-destructive task composition for transfer learning. *arXiv preprint arXiv:2005.00247* (2020).

[18] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250* (2016).

[19] Fanny Spagnolo et al. 2022. Aggressive Approximation of the SoftMax Function for Power-Efficient Hardware Implementations. *IEEE Transactions on Circuits and Systems II: Express Briefs* 69, 3 (2022), 1652–1656.

[20] Jacob R. Stevens et al. 2021. Softermax: Hardware/Software Co-Design of an Efficient Softmax for Transformers. In *2021 58th ACM/IEEE Design Automation Conference (DAC)* (San Francisco, CA, USA). IEEE Press, 469–474.

[21] Hugo Touvron et al. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]

[22] Ashish Vaswani et al. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[23] Ihor Vasyltsov and Wooseok Chang. 2021. Efficient Softmax Approximation for Deep Neural Networks with Attention Mechanism. arXiv:2111.10770 [cs.LG]

[24] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2018. GLUE: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461* (2018).

[25] Meiqi Wang et al. 2018. A High-Speed and Low-Complexity Architecture for Softmax Function in Deep Learning. In *2018 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. 223–226.

[26] Sai Qian Zhang, Bradley McDanel, and HT Kung. 2022. Fast: Dnn training under variable precision block floating point with stochastic rounding. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 846–860.

[27] Sai Qian Zhang, Bradley McDanel, HT Kung, and Xin Dong. 2021. Training for multi-resolution inference using reusable quantization terms. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 845–860.

[28] Sai Qian Zhang, Thierry Tambe, Nestor Cuevas, Gu-Yeon Wei, and David Brooks. 2023. CAMEL: Co-Designing AI Models and Embedded DRAMs for Efficient On-Device Learning. *arXiv preprint arXiv:2305.03148* (2023).

[29] Yuan Zhang, Yonggang Zhang, Lele Peng, Lianghua Quan, Shubin Zheng, Zhonghai Lu, and Hui Chen. 2022. Base-2 Softmax Function: Suitability for Training and Efficient Hardware Implementation. *IEEE Transactions on Circuits and Systems I: Regular Papers* 69, 9 (2022), 3605–3618. https://doi.org/10.1109/TCSI.2022.3175534

[30] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. 2016. Trained ternary quantization. *arXiv preprint arXiv:1612.01064* (2016).