Simplified Hardware Implementation of the Softmax Activation Function

→ Reduces the exponent Look up table size to $(0, 1]$

→ Applicable only for certain type of inputs

Softmax function

$$f_j(z) = \frac{e^{z_j}}{\sum_{k=1}^{n} e^{z_k}},$$

Taking log

$$\begin{aligned}
\log(f_j(z)) &= \log\left(\frac{e^{z_j}}{\sum_{k=1}^{n} e^{z_k}}\right) \\
&= \log(e^{z_j}) - \log\left(\sum_{k=1}^{n} e^{z_k}\right) \\
&= z_j - \log\left(\sum_{k=1}^{n} e^{z_k}\right).
\end{aligned}$$

$$\begin{aligned}
\log\left(\sum_{k=1}^{n} e^{z_k}\right) &= \log\Big(\sum_{k=1}^{n} \frac{e^m}{e^m} e^{z_k}\Big) \\
&= \log\Big(e^m \sum_{k=1}^{n} \frac{1}{e^m} e^{z_k}\Big) \\
&= \log e^m + \log\Big(\sum_{k=1}^{n} e^{-m} e^{z_k}\Big) \\
&= m + \log\Big(\sum_{k=1}^{n} e^{z_k - m}\Big),
\end{aligned}$$

where $m = \max_k(z_k)$. From (4) and (8) it follows that

$$\log(f_j(z)) = z_j - \left(m + \log\Big(\sum_{k=1}^{n} e^{z_k - m}\Big)\right).$$

$$q_j = z_j - \max_k (z_k).$$

$$q' = (q'_1, q'_2, q'_3, \ldots, 0, \ldots, q'_n)$$

$$q'_j = \frac{q_j}{\mu_q},$$

Corresponding to $z_j = \max(z_k)$

$$L^1(x) = \sum_{i=1}^{n} |x_i|,$$

$$Q = \frac{\left\| \frac{q}{\mu_q} \right\|_1}{n-1}.$$

Distance of maximum component from remainder of components

As $\bar{Q} \to 0$

q' becomes sparse

⇒ significant difference between maximum and other components

$$\begin{aligned} z_j &\ll m \Rightarrow \\ z_j - m &\ll 0 \Rightarrow \\ e^{(z_j-m)} &\ll 1 \Rightarrow \\ e^{(z_j-m)} &\simeq 0, j \neq i, z_i = m. \end{aligned}$$

$$\log(f_j(z)) = z_j - \left(m + \log\left(\sum_{k=1}^{n} e^{z_k - m}\right)\right).$$

⇓ → When $\bar{Q} \to 0$

$$\log(\widehat{f_j(z)}) = z_j - m$$

Hardware implementation is simplified under the following conditions.

$$\begin{aligned} z_j &\leq m \Rightarrow \\ z_j - m &\leq 0 \Rightarrow \\ e^{z_j-m} &\leq 1. \end{aligned}$$

⇓

e^x values are restricted to (0, 1]

[Simplified]

No conversion from the logarithmic to the real domain is required, since fj(z) represents the final classification layer.

$$z_j = \begin{cases} 3, & j = 1 \\ 6, & j = 2 \\ 4, & j = 3 \\ 2, & j = 4 \\ < 1, & j = 5, \ldots, 30, \end{cases}$$

Here most of the values are less than $\max(z_j) = 6$

$$f_j(z) = \begin{cases} 0.0382, & j = 1 \\ 0.7675, & j = 2 \\ 0.1039, & j = 3 \\ 0.0141, & j = 4 \\ < 0.005, & j = 5, \ldots, 30. \end{cases}$$

$$\widehat{f_j(z)} = \begin{cases} 0.0469, & j = 1 \\ 1, & j = 2 \\ 0.1250, & j = 3 \\ 0.0156, & j = 4 \\ 0, & j = 5, \ldots, 30. \end{cases}$$

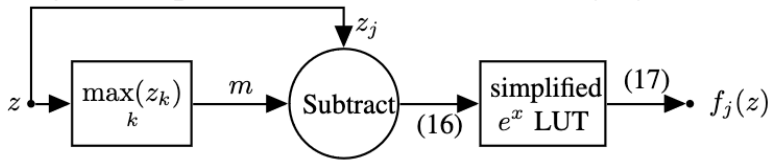


Fig. 5: Proposed Softmax architecture.

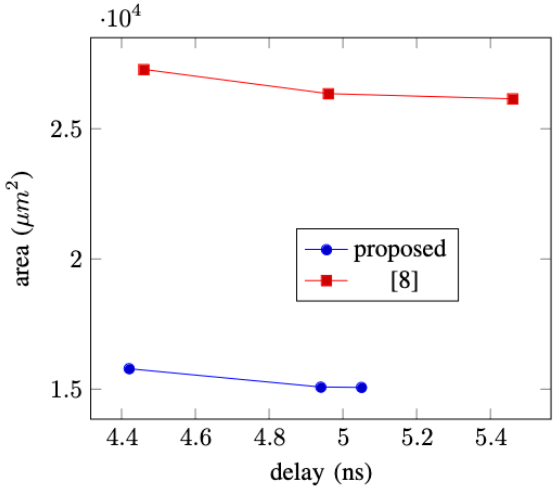
VI. Hardware Implementation Results

(vs)

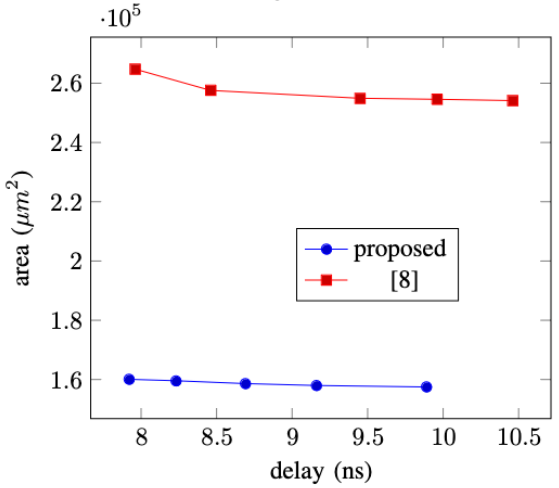
$$= m + \log\left(\sum_{k=1}^{n} e^{z_k - m}\right), \qquad (8)$$

⇓

This requires Log LUT & adder tree



(a) Softmax layer of size 10.



(b) Softmax layer of size 100.

Fig. 6: Area *vs* Delay plots for the proposed and [8] circuits in the case of 10-bit wordlength implemented in a standard-cell library.

For a fair comparison, both the proposed architecture and the architecture for normal LogSoftmax have been synthesized in a 90-nm 1.0 V CMOS standard-cell library with Synopsys Design Compiler.

Softermax: Hardware/Software Co-Design of an Efficient Softmax for Transformers (2021)

https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9586134

→ Removes Max calculation by using running Max

Normal Softmax ⇒ Optimized Softmax with Running Max

1. $m_0 \leftarrow -\infty$
2. **for** $k \leftarrow 1, V$ **do**
3. $m_k \leftarrow \max(m_{k-1}, x_k)$
4. **end for**
5. $d \leftarrow 0$
6. **for** j $\leftarrow 1, V$ **do**
7. $d \leftarrow d + 2^{x_j - m_V}$
8. **end for**
9. **for** i $\leftarrow 1, V$ **do**
10. $y_i \leftarrow \frac{2^{x_j - m_v}}{d}$
11. **end for**

(1)

1. $m_0 \leftarrow -\infty$
2. $d \leftarrow 0$
3. **for** j $\leftarrow 1, V$ **do**
4. $m_j \leftarrow \max(m_{j-1}, x_j)$
5. $d \leftarrow d \cdot 2^{m_{j-1} - m_j} + 2^{x_j - m_j}$
6. $y_i \leftarrow 2^{x_j - m_j}$
7. **end for**
8. **for** i $\leftarrow 1, V$ **do**
9. $y_i \leftarrow \frac{y_i \cdot 2^{m_i - m_V}}{d}$
10. **end for**

(2)

Assume we are processing the vector [2,1,3]. For the first element, the running max is two and thus the running sum will be $d = 2^{2-2} = 1$. For the second element, the running max is still two and thus the running sum will be $d = 1 + 2^{1-2} = 1.5$. For the final element, however, we encounter a new maximum value of three. We cannot simply add to the existing sum as is (i.e., $d \text{ !=} 1.5 + 2^{3-3} = 2.5$)

We must instead renormalize the running sum to account for this by multiplying by $2^{\text{OldMax}-\text{NewMax}}$, so that $d = 1.5 \times 2^{2-3} + 2^{3-3} = 0.75 + 1 = 1.75$. Note that this result is the same as if we had computed the accumulation using the true global maximum from the start: $d = 2^{2-3} + 2^{1-3} + 2^{3-3} = 1.75$.

Optimized Softmax ⇒ Same but with bit shift

1. $m_0 \leftarrow -\infty$
2. $d \leftarrow 0$
3. **for** j $\leftarrow 1, V$ **do**
4. $\quad m_j \leftarrow \max(m_{j-1}, x_j)$
5. $\quad d \leftarrow d \cdot 2^{m_{j-1}-m_j} + 2^{x_j-m_j}$
6. $\quad y_i \leftarrow 2^{x_j-m_j}$
7. **end for**
8. **for** i $\leftarrow 1, V$ **do**
9. $\quad y_i \leftarrow \frac{y_i \cdot 2^{m_i-m_V}}{d}$
10. **end for**

(2)

1. $m_0 \leftarrow -\infty$
2. $d \leftarrow 0$
3. **for** j $\leftarrow 1, V$ **do**
4. $\quad m_j \leftarrow \mathrm{IntMax}(m_{j-1}, x_j)$
5. $\quad d \leftarrow d \gg (m_j - m_{j-1}) + 2^{x_j-m_j}$
6. $\quad y_i \leftarrow 2^{x_j-m_j}$
7. **end for**
8. **for** i $\leftarrow 1, V$ **do**
9. $\quad y_i \leftarrow \frac{y_i \gg (m_V-m_i)}{d}$
10. **end for**

(3)

Detailed line by line explanation in Appendix

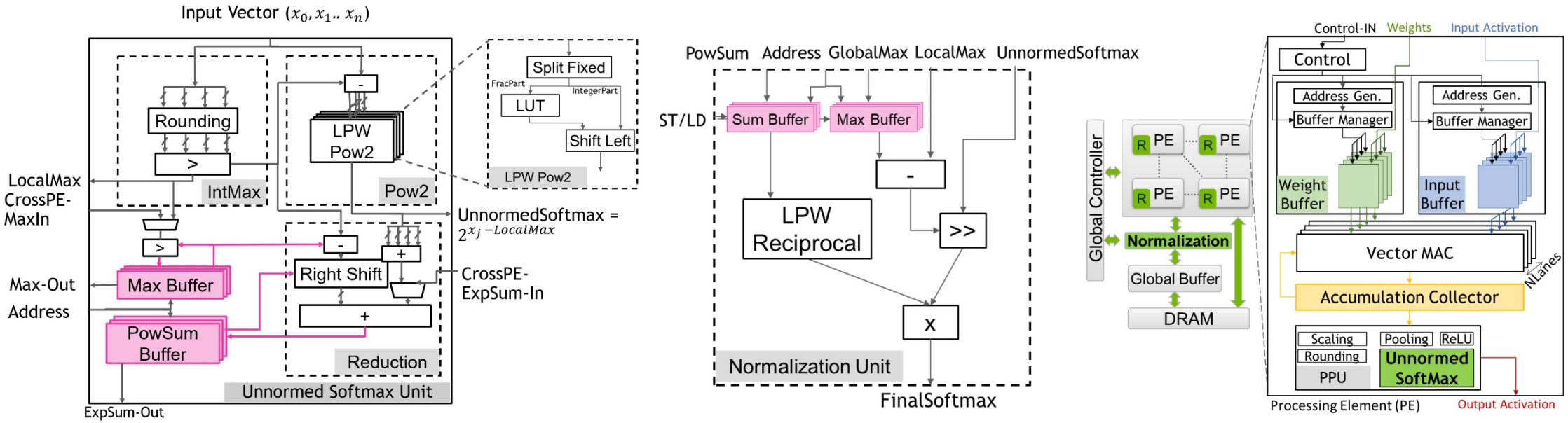


Fig. 4: (a) The Unnormed Softmax Unit determines the local max, performs the power of 2 calculation using the current max, and accumulates the denominator. (b) The Normalization Unit performs the renormalization of the numerator, as well as the final division of the numerator by the accumulated sum. (c) In an example accelerator [17], the Unnormed Softmax can be integrated into the post-processing vector unit on a per PE basis, while the Normalization Unit can be shared across multiple PEs and integrated between the PEs and the Global Buffer

A. Unnormed Softmax Unit Overview

The Unnormed Softmax unit is a composite hardware component comprised of three distinct subunits designed to perform calculations related to the softmax function in a neural network.

IntMax Unit

- **Functionality**: Operates on segments of an output vector.
- **Process**: Applies a ceiling function to each vector element in parallel to determine the maximum (IntMax) of the segment, enhancing parallel processing efficiency.

Power of Two Unit

→ Explained in detail in next slide

Reduction Unit

- **Input Handling**: Accepts the output from the Power of Two unit.
- **Summation Tree**: Utilizes a summation tree to aggregate softmax outputs, facilitating partial computations when dealing with large vectors.
- **Buffer Reading**: For large vectors, it reads from buffers to process the vector in slices.
- **Renormalization Process**: Adjusts the running sum based on the local maximum using shifting operations to maintain accuracy across slices.

Power of Two-Designed to replace the exponential function

1. **Fractional Part Calculation**:
   - The input value `x` is assumed to be in fixed-point representation.
   - The value `x` is scaled up by shifting it left by two bits (equivalent to multiplying by 4), creating `x_scaled`.
   - This scaling prepares `x` for a piece-wise linear approximation using a lookup table (LUT).
   - The `x_scaled` value is split into an integer part (used to index the LUT) and a fractional part.
2. **Linear Piece-Wise (LPW) Function**:
   - The fractional part of `x_scaled` is used in a linear piece-wise function, which approximates the power of two for the fractional part.
   - This function uses two LUTs: `m_lut` for the slope and `c_lut` for the y-intercept of the linear segments.
   - The output of the LPW is calculated using the formula:
     $\text{lpw} = m_{\text{lut}}[\text{int}(x_{\text{scaled}})] \times \text{frac}(x_{\text{scaled}}) + c_{\text{lut}}[\text{int}(x_{\text{scaled}})]$
3. **Handling Small Fractional Inputs**:
   - For inputs with less than two fractional bits, the LPW simplifies to just the y-intercept from the LUT, since the fractional part would be zero.
4. **Final Result**:
   - The output of the LPW, which represents the power of two for the fractional part, is then combined with the integer part by shifting it appropriately (equivalent to multiplying by $2^{\text{integer part}}$).

This method is optimized for hardware implementation by reducing the number of segments in the LPW and thus the size of the LUTs. General-purpose hardware might use 64-128 segments, while this optimized version uses only four. This reduces memory overhead and complexity, while still providing an adequate approximation for the power of two function

B. Normalization Unit Functions

The Normalization Unit is responsible for finalizing the softmax probability calculations.

Renormalization Mechanism

- **Integer-Based Adjustments**: Utilizes integer max values to ensure differences between local and global maxima are integers, simplifying calculations.
- **Shifter Utilization**: Employs a shifter to perform renormalization of the numerator, taking advantage of the base change in Softermax.

Division Implementation

- **Linear Piece-Wise Reciprocal Unit**: Precedes the actual division, preparing the values.
- **Integer Multiplier**: Completes the division process, resulting in the final probabilities.

C. Integration into Accelerators

The Unnormed Softmax Unit is designed for easy integration into existing hardware accelerators, enhancing the computation of softmax functions.

Compatibility with Hardware

- **Tensor Processing Hardware**: Compatible with GPU tensor cores, TPUs, and other DNN inference accelerators.
- **Integration Example**: Can be incorporated into post-processing units of architectures like MAGNet to perform operations in conjunction with pooling and ReLU.

Efficiency and Throughput

- **Sizing Considerations**: Should be scaled to match the MAC (Multiply-Accumulate) throughput to maximize efficiency.
- **Low-Overhead Advantage**: The Softermax-enabled hardware promises reduced computational overhead.
- **Normalization Unit Placement**: Positioned strategically to perform softmax calculations off the critical data processing path, ensuring optimal flow of operations.

Layer Norm

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

→ Use Dynamic Compress

→ Complex Algorithm (Difficult to understand)

AILayerNorm

AILayerNorm algorithm utilizes dynamic compression to achieve low-precision statistic calculation based on the PTF quantization.

Challenge in Low-Precision Environments:

In environments where computation precision is limited (like 8-bit integers), calculating the square of inputs (which is necessary for variance computation) can be computationally expensive and less accurate.

Dynamic Compression:

To address this, AILayerNorm introduces a dynamic compression method.
This method reduces the precision of input values for calculation purposes, thereby simplifying the computation.
It does this by compressing an 8-bit input x into a 4-bit approximation y.
The method involves filtering out less significant bits of x and dynamically changing the length of the bit-filtered based on the value of x. This is summarized in the function Dynamic Compress(x).

$$y, s = \text{Dynamic Compress}(x):$$

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

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 this only induces errors of 0.2% over E(x2) and 0.4% over standard deviation with uniformly distributed input data.

Compression Methodology:

•Inputs are divided by 2^4 or 2^2 depending on their magnitude, determined by the two most significant bits.

•This division reduces the precision but retains a significant portion of the information.

•A 1-bit signal s is generated alongside to keep track of the compression level (whether it was divided by 2^4 or 2^2).

Square and Decompress:

•After compression, the squared values are calculated using 4-bit arithmetic, which is less resource-intensive than using 8-bit arithmetic.

•The << (4s) operation in the algorithm is a bit shift that helps in decompressing or scaling back the squared values to an appropriate range.

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

⇒ Complex Algorithm
⇒ Paper does not answer what is zp, α_i

⇒ I researched to find the original implementation of PTF algorithm & found calculations for zp & α_i

Original Power of Two Factor (PTF)

$$X_Q = Q(X|b) = \text{clip}(\lfloor \frac{X}{2^{\alpha} s} \rceil + zp, 0, 2^b - 1), \quad (8)$$

with

$$s = \frac{\max(X) - \min(X)}{2^b - 1} / 2^K, \quad (9)$$

→ Paper assumes k=3

$$zp = \text{clip}(\lfloor -\frac{\min(X)}{2^K s} \rceil, 0, 2^b - 1), \quad (10)$$

$$\alpha_c = \underset{\alpha_c \in \{0,1,\cdots,K\}}{\arg\min} \left\| X_c - \lfloor \frac{X_c}{2^{\alpha_c} s} \rceil \cdot 2^{\alpha_c} s \right\|_2. \quad (11)$$

Note: This 's' is different from the 's' mentioned in previous slide

→ Choose α_c such that it minimizes the non 2 error

layer-wise quantization parameter

PTF $\alpha \in \mathbb{N}^C$,

$s, zp \in \mathbb{R}^1$

At this point, each channel has its own Power-of-Two Factor α and layer-wise parameters s, zp. During inference, layer-wise parameters s and zp can be extracted, so the computation of μ,σ could be done in the integer domain rather than floating-point

Summary of PTF working

Phase 1: Shift the quantized activation with Power-of-Two Factor α:

$$\hat{X}_Q = (X_Q - zp) << \alpha. \quad (12)$$

Phase 2: Calculate the mean and variance based on the shifted activation $\hat{X}_Q$:

$$\mu(X) \approx \mu(2^{\alpha} s \cdot (X_Q - zp)) = s \cdot \mu(\hat{X}_Q), \quad (13)$$

$$\sigma(X) \approx \sigma(2^{\alpha} s \cdot (X_Q - zp)) = s \cdot \sigma(\hat{X}_Q). \quad (14)$$

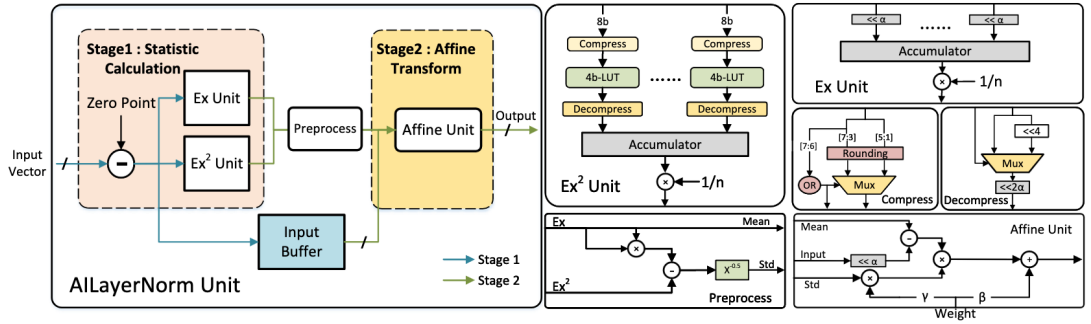


Fig. 5. Hardware design of AILayerNorm Unit

Stage 1: Statistic Calculation

This stage is responsible for calculating the mean and variance of the inputs, which are crucial steps in the normalization process.

1.Input Vector: This is where the input data (an array of values) enters the AILayerNorm Unit.

2.Zero Point: The zero point is a reference value subtracted from the input data to center it, typically used in quantization to ensure symmetry around zero.

3.Ex Unit:

1. Processes the input after zero point adjustment.
2. Scales the input with power-of-two factors, which is a form of quantization that allows for efficient computation by using shifts instead of multiplications.
3. Performs a 12-bit reduction, which aggregates the inputs for the mean calculation.

4.Ex2 Unit:

1. Applies dynamic compression to the input, reducing the 8-bit input to a 4-bit representation to lessen the computational load.
2. Utilizes a 16-entry Look-Up Table (LUT) to implement the square function. Since there are only 16 possible outcomes for squaring a 4-bit input, a LUT can efficiently replace a multiplier.

5.Input Buffer: Temporarily stores the input data as it is being processed. It supports a pipelined design that allows for continuous data processing without stalling.

6.Decompress Unit:

1. Compensates for the scaling that happened during the dynamic compression and power-of-two factor scaling.
2. Adjusts the scale of the squared inputs back to an appropriate range for accumulation.

7.Accumulator:

1. Collects the processed inputs from the Ex Unit and Ex2 Unit.
2. Sums up the scaled inputs for the mean (Ex) and the scaled squared inputs for the variance (Ex2).

Preprocess Unit

This unit prepares the calculated mean and variance for the affine transformation.

1.Mean and Std (Standard Deviation):

1. It takes the accumulated values for Ex and Ex2 to compute the mean and standard deviation.
2. Implements the power of -0.5 function (required for the inverse of the standard deviation) using another LUT to save area and power.

Stage 2: Affine Transform

After the statistics are computed, this stage performs the normalization and rescaling using the affine parameters (gamma and beta).

1.Affine Unit:

1. Fuses the normalization and rescaling operations into two multiplications and two additions.
2. Uses the first multiplier to compute 'A' with weights (gamma) and the standard deviation.
3. Scales the inputs with the power-of-two factor and subtracts the mean.
4. Performs the final multiplication and addition to obtain the output $Y = A \cdot X + B$.

2.Weights (gamma and beta):

1. These are the affine parameters used in the normalization process.
2. They are quantized to 8-bit integers for consistency with the low-precision approach of the hardware.

GELU optimizations

Hardware-Friendly Activation Function Designs and Its Efficient VLSI Implementations for Transformer-Based Applications(2023)

https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10168591

→ Replace GELU with other functions

→ No approximations / difference between Software & hardware implementation

Approximations for GELU

A. Dying Neuron Resuscitation (DNR) Series

$$DNR(x) = \begin{cases} 0, & x \leq -2, \\ x, & x \geq 0, \\ \frac{x \times (x+2)}{n}, & -2 < x < 0, \end{cases} \quad (1)$$

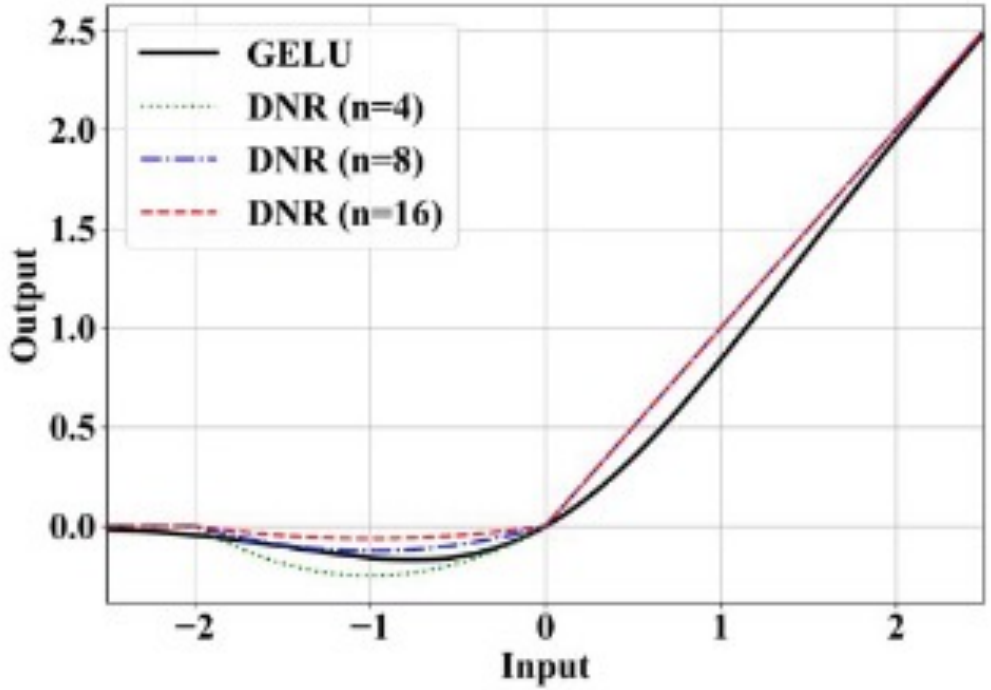


Fig. 1. DNR series with different n.

where n belongs to {4, 8, 16, 32}.
As x is positive or less than -2, DNR acts exactly the same as ReLU.
A quadratic polynomial is used to create a convex curve as −2 < x < 0.
A programmable parameter n determines the depth of the convex curve as shown; the larger n is, the shallower the curve is.
All valid values of n are powers of two so that the division in (1) can be simply replaced using a right-shift by log2 n bits.

A. Implementation of DNR Series

- The term x(x+2) in (1) is carried out as x^2+2x instead of the original trivial form, where a carry propagate adder (CPA) and a full multiplier (including a final-stage CPA) are needed.
- First, x^2 is calculated using a partial product multiplier, which does not need a CPA and thus is much smaller than a complete multiplier.
- Then, x is left-shifted by n_frac bits for Q-format alignment with the multiplier outputs.
- The trailing 1-bit left-shift is to get 2x, which requires no logic.
- Next, a CPA follows a carry save adder (CSA) to get x^2+2x. The output of CPA is then right-shifted by n_frac bits to ensure that the result is represented in the same Q format of the input x.
- At last, the result is further right-shifted by a user-specified d=log2 n bits, specified by the signal mode, to carry out the divide-by-n operation required in (1).
- The output y of a DNR function can be divided into three cases: 1) y = x if x >= 0; 2) y = 0 if x <= -2; 3) y equals to the output of the quadratic function q otherwise. A trivial implementation
- requires an explicit comparator to determine whether x <= -2 (Case 2) or not (Case 1, Case 3). Since the support of multiple Q formats is essential in the proposed design, the hardware cost of such comparator is pretty high. However, it is observed that Case 1 can be identified if the sign bit of x is 0 (i.e. nonnegative) and Case 2 can also be distinguished if the sign bit of the quadratic function output q is 1 (i.e., negative).
- Their implementation can correctly classify all cases through checking two sign bits only and needs no comparator at all, which helps minimize the hardware cost further.

Diagram next slide

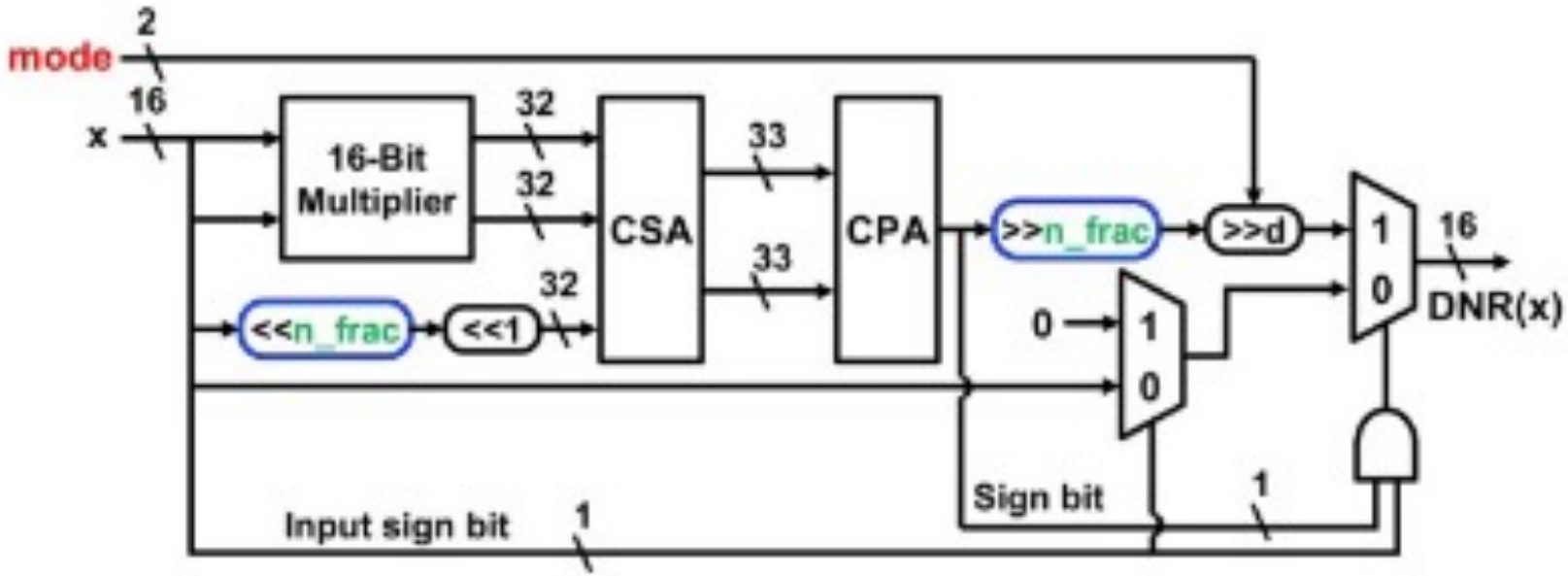


Fig. 3. Block diagram of the hardware implementation for DNR series.

B. Piecewise Linear (PWL) Series

$$PWL(x) = \begin{cases} x, & x \geq 0, \\ m_1 \times x, & P \leq x < 0, \\ m_2 \times x + b, & a \leq x < P, \\ 0 & x < a, \end{cases} \qquad (2)$$

$$a = \frac{m_2 - m_1}{m_2} \times P,$$

$$b = (m_1 - m_2) \times P,$$

where $m_1 \in \{1, 0.5, 0.25, 0.125\}$, $m_2 \in \{-0.5, -0.25, -0.125, -0.0625\}$ and $P \in \{-1, -0.75, -0.5, -0.25\}$.

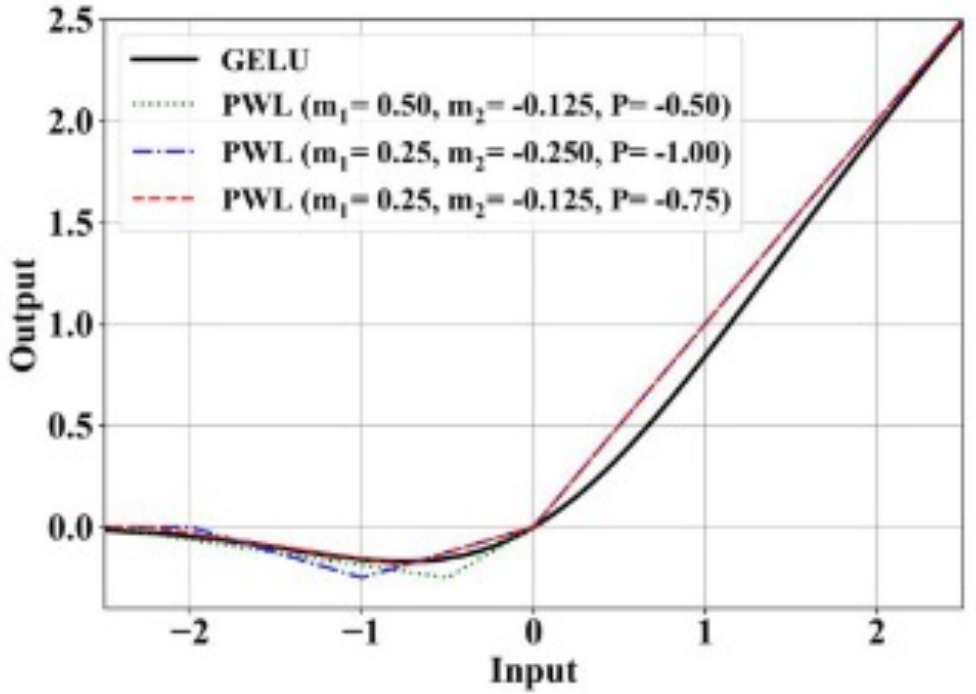


Fig. 2. PWL series with different m_1, m_2, and P.

- Unlike DNR, two line segments are used in PWL to create a polygonal curve in the near-zero negative input region
- The slope of the right segment is m1 and the slope of the left one is m2;
- two line segments intersect at the point (P, m1*P).
- Since the magnitudes of m1 and m2 are all powers of two, two multiplication operations in (2) can then be safely replaced using an equivalent right-shift of proper bits.
- That is, a PWL function actually needs neither multipliers nor LUTs, which further makes its hardware implementation even smaller, faster, and more power efficient. Furthermore, similar to DNR, a PWL function introduces no approximation errors between its software and hardware versions because it does not try to mimic GELU at all, either.

Diagram next slide

PWL hardware implementation.

- Output of a PWL function can be partitioned into four cases: 1) x >= 0; 2) $P \leq x < 0$; 3) $a \leq x < P$; 4) x < a according to (2).
- Unlike DNR, PWL needs a Q-format-aware comparator to determine whether x is no less than a given P (Cases 1~2 or Cases 3~4).
- Again, the sign bit of x can be used to distinguish Case 1 and Case 2; the output value $m1 \times x$ in Case 2 can be calculated via right-shifting x by $r_1=|\log_2(m_1)|$ bits, where no multiplier is actually required.
- Next, the process to get the output value of $v = m_2 \times x + b$ in Case 3 is twofold: getting $m2 \times x$ first then adding the negative
- constant b using a subtractor.
- In Step 1, the term $m2 \times x$ can be computed via right-shifting x by $r_2=|\log_2(-m_2)|$ bits, where no multiplier is required again. In Step 2, a selector is first used to pick the right version of the constant b because it is jointly determined by the configurable parameters m1, m2, and P.
- The selected b is then left-shifted by n_frac bits before addition to ensure its Q format matches that of the input x. To distinguish Case 3 and Case 4, a trivial solution demands a Q-format- aware comparator to see if x < a specified in (2). However, our implementation performs the same trick: x is less than a only if v is positive; i.e., no extra comparator is required.
- At the end, three multiplexers are used to select the correct output from four possible cases. Note that the proposed implementation is multiplier-free, which suggests PWL is expected to be smaller and faster than DNR.
- The 6-bit signal mode is used to specify one of 64 valid configurations and each indicates a different combination of m1, m2, and P.
- The hardware cost can be reduced if those three parameters are fixed.
- Similarly, the cost can be further decreased if only one Q format is supported. It is a common tradeoff between design flexibility and cost.

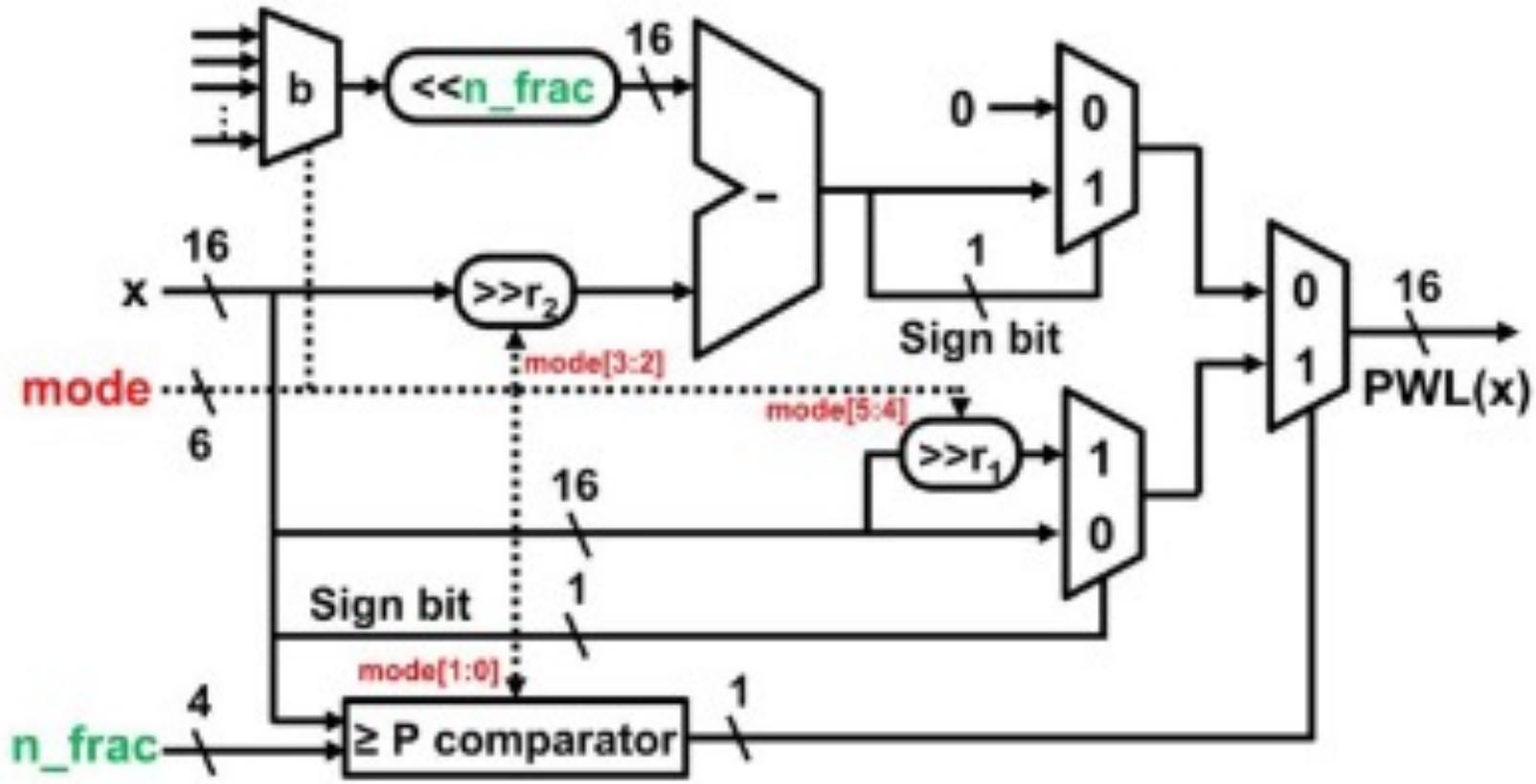


Fig. 4. Block diagram of the hardware implementation for PWL series.

TABLE I. Accuracy of various activation functions without post-training.

	Top-1 Accuracy	
	DeiT-Small	**Swin-Tiny**
GELU (Baseline, 300 epochs)	81.17	81.15
ReLU	11.88	-
DNR (n=4)	75.76	74.87
DNR (n=8)	**79.22**	**76.78**
DNR (n=16)	64.29	40.72
PWL (m_1=1/4, m_2=-1/8, P=-0.75)	**80.34**	80.20
PWL (m_1=1/4, m_2=-1/8, P=-1.00)	25.50	69.68
PWL (m_1=1/4, m_2=-1/4, P=-1.00)	79.46	**80.41**
PWL (m_1=1/2, m_2=-1/8, P=-0.50)	78.67	78.15
PWL (m_1=1/4, m_2=-1/16, P=-0.50)	77.82	74.65
PWL (m_1=1/4, m_2=-1/8, P=-0.50)	58.72	52.84

TABLE II. Accuracy of various activation functions after post-training.

	Top-1 Accuracy	
	DeiT-Small	**Swin-Tiny**
GELU (Baseline, 300 epochs)	81.17	81.15
GELU (post-training)	81.42	81.26
ReLU	81.12	-
DNR (n=4)	81.15	81.17
DNR (n=8)	**81.35**	**81.27**
DNR (n=16)	81.26	81.26
PWL (m_1=1/4, m_2=-1/8, P=-0.75)	**81.39**	81.20
PWL (m_1=1/4, m_2=-1/8, P=-1.00)	81.27	81.20
PWL (m_1=1/4, m_2=-1/4, P=-1.00)	81.24	81.16
PWL (m_1=1/2, m_2=-1/8, P=-0.50)	81.27	81.18
PWL (m_1=1/4, m_2=-1/16, P=-0.50)	81.32	**81.23**
PWL (m_1=1/4, m_2=-1/8, P=-0.50)	81.23	81.16

TABLE III. Accuracy of various activation functions if train-from-scratch.

	Top-1 Accuracy	
	DeiT-Small	**Swin-Tiny**
GELU[a,b]	81.17	81.15
DNR (n=8)[b]	80.94	81.17
PWL (m_1=1/4, m_2=-1/8, P=-0.75)[b]	80.73	81.26

[a] Source code imported from the official Github repository, [20]–[21].
[b] Accuracy reported in our experiment environment, 300 epochs.

I-ViT: Integer-only Quantization for Efficient Vision Transformer Inference

→ Transform GELU to Sigmoid like/Softmax like function.

→ Complex Algorithm for Integers

$$\begin{aligned}\mathrm{GELU}(x) &= x \cdot \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2} dt \\ &\approx x \cdot \sigma(1.702x) \\ &= S_x \cdot I_x \cdot \sigma(S_x \cdot 1.702 I_x)\end{aligned}$$

$1.702\, I_x = I_p = I_x + (I_x \gg 1) + (I_x \gg 3) + (I_x \gg 4)$

$$2^{S_\Delta \cdot I_p} = 2^{(-q)+S_\Delta \cdot (-r)} = 2^{S_\Delta \cdot (-r)} \gg q$$

where $S\Delta \cdot (-r) \in (-1, 0]$ is the decimal part, and q and r are both positive integer value

$$\begin{aligned}\sigma(S_x \cdot I_p) &= \frac{1}{1 + e^{-S_x \cdot I_p}} \\ &= \frac{e^{S_x \cdot I_p}}{e^{S_x \cdot I_p} + 1} \\ &= \frac{e^{S_x \cdot (I_p - I_{max})}}{e^{e^{S_x \cdot (I_p - I_{max})}} + e^{S_x \cdot (-I_{max})}}\end{aligned}$$

$$\begin{aligned}2^{S_\Delta \cdot (-r)} &\approx [S_\Delta \cdot (-r)]/2 + 1 \\ &= S_\Delta \cdot [((-r) \gg 1) + I_0]\end{aligned}$$ where $I_0 = \lfloor 1/S_\Delta \rfloor$.

$$\begin{aligned}e^{S_\Delta \cdot I_\Delta} &= 2^{S_\Delta \cdot (I_\Delta \cdot \log_2 e)} \\ &\approx 2^{S_\Delta \cdot (I_\Delta + (I_\Delta \gg 1) - (I_\Delta \gg 4))}\end{aligned}$$

observe that this is little bit similar to softmax formula (especially numerator)

Algorithm 2: Integer-only GELU: **ShiftGELU**

Input: I_{in} : Integer input
S_{in} : Input scaling factor
k_{out} : Output bit-precision
Output: I_{out} : Integer output
S_{out} : Output scaling factor

Function `ShiftGELU` $(I_{in}, S_{in}, k_{out})$ **:**

$I_p \leftarrow I_{in} + (I_{in} \gg 1) + (I_{in} \gg 3) + (I_{in} \gg 4)$; ▷ $1.702I$

$I_\Delta \leftarrow I_p - \max(I_p)$;
$(I_{exp}, S_{exp}) \leftarrow$ `ShiftExp`(I_Δ, S_{in});
$(I'_{exp}, S'_{exp}) \leftarrow$ `ShiftExp`$(-\max(I_p), S_{in})$;
$(I_{div}, S_{div}) \leftarrow$ `IntDiv`$(I_{exp}, I_{exp}+I'_{exp}, k_{out})$; ▷ Eq. 18

$(I_{out}, S_{out}) \leftarrow (I_{in} \cdot I_{div}, S_{in} \cdot S_{div})$;
return (I_{out}, S_{out}); ▷ $I_{out} \cdot S_{out} \approx \text{GELU}(I_{in} \cdot S_{in})$

End Function

Function `ShiftExp` (I, S) **:**

$I_p \leftarrow I + (I \gg 1) - (I \gg 4)$; ▷ $I \cdot \log_2 e$
$I_0 \leftarrow \lfloor 1/S \rceil$;
$q \leftarrow \lfloor I_p/(-I_0) \rfloor$; ▷ Integer part
$r \leftarrow -(I_p - q \cdot (-I_0))$; ▷ Decimal part
$I_b \leftarrow ((-r) \gg 1) + I_0$; ▷ Eq. 15
$I_{exp} \leftarrow I_b \ll (N - q)$; [3] ▷ Eq. 14
$S_{exp} \leftarrow S/(2^N)$;
return (I_{exp}, S_{exp}); ▷ $S_{exp} \cdot I_{exp} \approx e^{S \cdot I}$

End Function

[3] To avoid too small values after right shifting, we first have a N-bit left shifting.

IntDiv(I1, I2, k) implements the integer division function, and I1, I2, and k are integer dividend, integer divisor and output bit width, respectively.

$$\text{Softmax}(x_i) = \frac{e^{S_{\Delta_i} \cdot I_{\Delta_i}}}{\sum_j^d e^{S_{\Delta_j} \cdot I_{\Delta_j}}} = \frac{e^{S_{x_i} \cdot (I_{x_i} - I_{max})}}{\sum_j^d e^{S_{x_j} \cdot (I_{x_j} - I_{max})}}$$

No Hardware implementation specified.
They use similar calculation for Softmax like ↗
⇒ Hardware can be reused

High speed reconfigurable architecture for softmax and GELU in vision transformer

https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/ell2.12751

ViT uses the following formula to calculate GELU:

$$g(x_i) = 0.5\,x_i\left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}\left(x_i + 0.044715\,{x_i}^3\right)\right)\right) \tag{2}$$

We set $h(x_i) = \sqrt{\frac{2}{\pi}}\left(x_i + 0.044715\,{x_i}^3\right)$. Formula (2) can be expressed as:

tanh formula

$$\begin{aligned} g(x_i) &= 0.5\,x_i\left(1 + \frac{e^{h(x_i)} - e^{-h(x_i)}}{e^{h(x_i)} + e^{-h(x_i)}}\right) \\ &= \frac{x_i\,e^{h(x_i)}}{e^{h(x_i)} + e^{-h(x_i)}} = \frac{x_i}{1 + e^{-2h(x_i)}} \end{aligned} \tag{3}$$

Formula (3) is transformed to:

using exponent 2

$$g(x_i) = \frac{x_i}{1 + 2^{-2\log_2 e \cdot h(x_i)}} \tag{5}$$

The exponent in formula (5) can be expressed as:

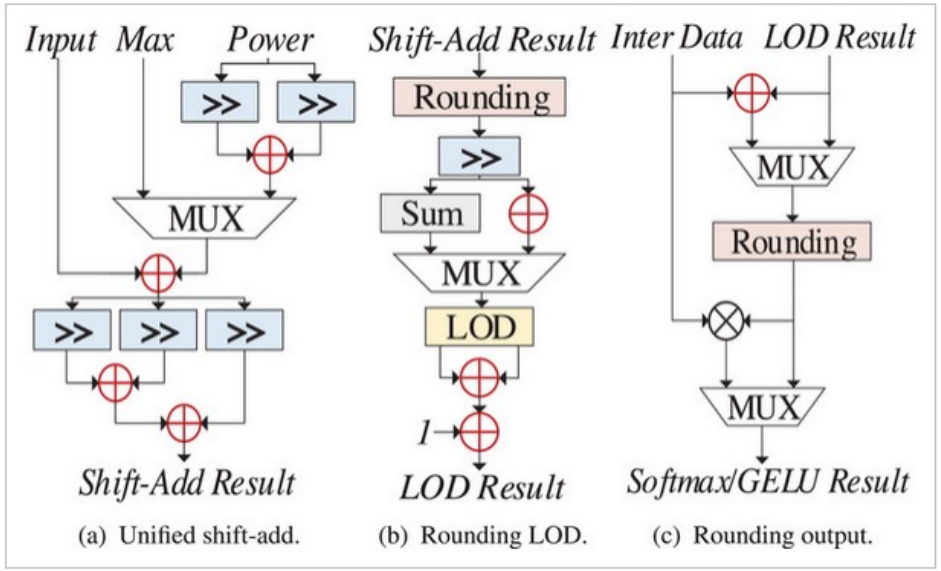
$$v(x_i) = -2\log_2 e \cdot h(x_i) = -2\log_2 e \cdot \sqrt{\frac{2}{\pi}}\left(x_i + 0.044715\,{x_i}^3\right) \tag{7}$$

They use 16-bit fixed-point (INT16) to perform all GELU operations.

- They first approximate the constant coefficients in formulas (7).
- Log_2e is approximated as 1.0111= 1+0.1-0.0001
- 0.044715 is approximated as 0.000011= 0.00001+0.000001
- $-2\log_2 e\sqrt{\frac{2}{pi}}$ = -10.0101=-10-0.01-0.0001
- All multiplication operations can be implemented using the unified shift-add circuit structure with only shifters and adders as shown in Fig. 3a.
- Note that base 2 power calculation, while allowing a somewhat more hardware-friendly implementation, cannot directly substitute the entire power calculation with a simple shift because the exponents are not guaranteed to be integers.
- To achieve this, they round the calculation results of formulas (7) to approximately convert all exponents to integers. We demonstrate that this substitution leads to negligible loss in accuracy when ViT inference is performed.

Table 2. Accuracy comparison results

Model	Method	Data format	Top-1 accuracy(%)	Loss(%)
ViT-S	Baseline	FP32	77.96	-
	Only softmax	INT16	77.67	0.29
	Only GELU	INT16	77.41	0.55
	Softmax+GELU	INT16	77.13	0.83
ViT-B	Baseline	FP32	83.99	-
	Only softmax	INT16	83.68	0.31
	Only GELU	INT16	83.97	0.02
	Softmax+GELU	INT16	83.61	0.38
ViT-L	Baseline	FP32	85.15	-
	Only softmax	INT16	84.87	0.28
	Only GELU	INT16	84.83	0.32
	Softmax+GELU	INT16	84.78	0.37



APPENDIX

Algorithm (1) - A Basic Softmax Calculation using Powers of 2

1. Initialize the maximum value $m_0$to negative infinity to prepare for finding the maximum logit.
2. Iterate over all elements k in the vector V.
3. Update m_k to be the maximum of the previous maximum m_{k-1} and the current element x_k.
4. End of the loop for finding the maximum value.
5. Initialize the sum d for the denominator of the softmax to 0.
6. Iterate over all elements j in the vector V.
7. Update the sum d by adding 2 raised to the power of the difference between the current logit x_j and the maximum value m_v (scaled by j for precision).
8. End of the loop for computing the sum for the denominator.
9. Iterate over all elements i in the vector V.
10. Calculate the softmax probability y_i for each element by taking 2 raised to the power of $x_j - m_v$ and dividing by d.
11. End of the loop for calculating the softmax probabilities.

Algorithm (2) - An Optimized Version

1. Similar initialization of m_0 to negative infinity.
2. Initialize the sum d to 0.
3. Iterate over all elements j in the vector V.
4. Update m_j to be the maximum of the previous maximum m_{j-1} and the current element x_j.
5. Update the sum d by adding the result of 2 raised to the power of $m_j - m_{j-1}$ and the element x_j minus m_j.
6. Calculate the intermediate softmax value y_i by taking 2 raised to the power of $x_j - m_j$.
7. End of the loop for computing intermediate values.
8. Iterate over all elements i in the vector V.
9. Calculate the final softmax probability y_i by taking the intermediate value y_i multiplied by 2 raised to the power of $m_i - m_v$ and dividing by d.
10. End of the loop for calculating final softmax probabilities.

Algorithm (3) - Further Optimized with Integer Operations

1. Similar initialization of m_0 to negative infinity.
2. Initialize the sum d to 0.
3. Iterate over all elements j in the vector V.
4. Update m_j to be the maximum of the previous maximum m_{j-1} and the current element x_j, using an integer maximum function (IntMax).
5. Update the sum d by applying a bitwise right shift (>>) to d by the difference between m_j and m_{j-1}, and then adding 2 raised to the power of $x_j - m_j$.
6. Calculate the intermediate softmax value y_i by taking 2 raised to the power of $x_j - m_j$.
7. End of the loop for computing intermediate values.
8. Iterate over all elements i in the vector V.
9. Calculate the final softmax probability y_i by applying a bitwise right shift (>>) to the numerator by the difference between m_v and m_i, and then dividing by d.
10. End of the loop for calculating final softmax probabilities.