SCIENCE CHINA
Information Sciences

October 2024, Vol. 67, Iss. 10, 200404:1–200404:15
https://doi.org/10.1007/s11432-024-4137-4

• RESEARCH PAPER •
Special Topic: AI Chips and Systems for Large Language Models

Hardware-oriented algorithms for softmax and layer normalization of large language models

Wenjie LI[1], Dongxu LYU[1], Gang WANG[1], Aokun HU[1], Ningyi XU[1*] & Guanghui HE[1,2,3*]

[1]*School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai* 200241*, China;*
[2]*Department of Micro/Nano Electronics, Shanghai Jiao Tong University, Shanghai* 200241*, China;*
[3]*MoE Key Laboratory of Artificial Intelligence, Shanghai Jiao Tong University, Shanghai* 200241*, China*

Received 31 January 2024/Revised 17 June 2024/Accepted 20 August 2024/Published online 12 September 2024

Abstract While large language models (LLMs) have sparked a new revolution in the field of natural language processing (NLP), their hardware accelerators have garnered tremendous attention. However, softmax and layer normalization which are the most common non-linear operations in LLMs are frequently overlooked. This paper presents hardware-oriented algorithms for both softmax and layer normalization of LLMs. We propose an approximate approach to implementing division in softmax and extend it for simultaneously computing square root and performing division in layer normalization. It replaces the original computation by multiplication and shifting. For softmax, we further approximate the exponential function by truncating its exponent and then reuse the involved subtraction. For layer normalization, we additionally simplify the computation of denominator by directly removing the term regarding the square of the mean. Furthermore, hardware architectures are developed for the proposed algorithms of softmax and layer normalization. They can work as plug-and-play units for LLM accelerators, requiring no fine-tuning and introducing negligible performance loss. Compared with the state-of-the-art designs, the proposed softmax architecture can save up to 23.45% area cost and 17.39% power consumption, while the proposed layer normalization architecture can save up to 32.70% area cost and 14.29% power consumption.

Keywords large language model, softmax, layer normalization, hardware architecture, Transformer

1 Introduction

In the past few years, large language models (LLMs) which are a class of Transformer-based models have been developed rapidly. Thanks to their unprecedented performance in a wide range of applications, LLMs have attracted great attention in academic research and everyday life. A number of LLM families such as GPT3 [1], GPT4 [2], OPT [3], LLaMA [4] and LLaMA2 [5] have been introduced by both the academic and industrial communities. Undoubtedly, the trend of LLMs has quickly swept through the research community and is driving a new wave of artificial intelligence (AI) technology [6–8]. They are even considered to be a precursor of artificial general intelligence (AGI) systems in the future [9].

The unparalleled performance of LLMs comes at the expense of a surge in the number of parameters and the computational complexity. For example, the well-known GPT-3 is reported to have approximately 175 billion model parameters. Even deployed in several advanced GPUs, an LLM with such a large number of parameters typically has an inference speed of less than one token per second [10]. The staggering computational and storage requirements of LLMs hinder them from practical implementations, especially in resource-limited computing platforms. In this context, designing dedicated hardware accelerators for LLMs is a promising solution. As evidenced by previous extensive research about hardware accelerators for Transformers-based models [11–17], dedicated hardware accelerators can not only provide significant speedup but also achieve superior energy efficiency.

Softmax and layer normalization are the most common non-linear operations in LLMs. Although current activation functions of LLMs may require different non-linear operations, [18] demonstrates that using

* Corresponding author (email: xuningyi@sjtu.edu.cn, guanghui.he@sjtu.edu.cn)

the ReLU activation function has a negligible impact on model performance. Since ReLU is extremely simple for implementation, softmax and layer normalization will become the most critical non-linear operations in LLMs. As reported in some papers [19, 20], softmax and layer normalization account for a significant portion of the GPU runtime. For example, ref. [20] shows that softmax occupies nearly half of the GPU runtime when the sequence length is set to 4096. For a Transformer accelerator, the softmax and layer normalization units could also result in over 50% of the area overhead of the compute modules [21, 22]. However, they are usually overlooked in existing hardware accelerator designs. Some designs [11,12,17] do not provide details on the implementation of softmax and layer normalization, while some others [13–15] implement them using straightforward methods such as look-up tables (LUTs) or floating-point arithmetic units. Since such straightforward methods incur significant hardware overhead, optimization on hardware architectures of softmax and layer normalization becomes crucial.

The hardware architecture design of softmax has garnered attention earlier than layer normalization, as deep neural networks (DNNs) usually utilize softmax to output the probabilities of different classes in multi-category classification tasks [23]. Early hardware architectures reported for softmax were all designed for DNNs [24–26]. An efficient hardware and software co-design of softmax for Transformers is proposed in [20], which involves base replacement and linear fitting. Based on these approximations, a method called "online normalization" is presented to reduce the number of data reads. With the help of fine-tuning, it only introduces negligible accuracy loss for Transformers. Inspired by [20, 25], ref. [27] proposes an efficient softmax architecture for Transformers which can achieve extremely low area cost. Its main innovation lies in using advanced piecewise linear approximation computation (PLAC) [28] for linear fitting. However, it only estimates the accuracy on a single dataset and does not specify whether fine-tuning is required. Similar to [26], ref. [29] also replaces the division by right shifting. Instead of simply replacing the exponential function by a simple linear function, ref. [29] achieves a more precise approximation by employing both linear function and right shifting simultaneously. It introduces acceptable accuracy loss when applied to Transformers. However, it is hard to efficiently cooperate with the online normalization because of the absence of base replacement. Ref. [30] successfully adopts online normalization by means of approximation for exponents, instead of direct base replacement. In addition, it introduces an offset to compensate for the error caused by approximation. Consequently, the softmax architecture proposed in [30] is the state-of-the-art design which can not only achieve limited performance loss for Transformers without any fine-tuning, but also offer fairly low area cost.

Unlike softmax, there is limited research dedicated to optimizing the hardware architecture of layer normalization. In addition to its involved division, it also needs to compute square roots, which poses new challenges for efficient hardware implementation. In general, computing the square root of x and dividing by it can be replaced by multiplying with $x^{-0.5}$. To obtain $x^{-0.5}$, most of the reported works [30–32] use LUTs. The division of layer normalization is more sensitive than that of softmax and therefore directly approximating it with right shifting may bring significant performance loss for LLMs. Even though the division can be simplified, the computation of square roots remains difficult to avoid. The layer normalization proposed in [30] adopts a bit-level compression when computing the square of the input. However, it only simplifies the square unit. Besides, it is dedicated to the quantization scheme of [33], rather than a universal method.

This paper primarily proposes hardware-oriented approximate algorithms for softmax and layer normalization. Additionally, hardware architectures are developed to demonstrate the effectiveness and superiority of our proposed algorithms. We first propose an approximate approach for the division of softmax. The divisor is truncated such that it becomes a power-of-two integer. Besides, we introduce an offset which is determined by the difference between the divisor and the power-of-two integer, in order to compensate for the error brought by the approximation. We further truncate the exponents to approximate the exponential function such that all the involved computations can be realized by addition (subtraction) and shifting. Moreover, we reuse the subtraction required for finding the maximum input, which reduces the area cost of the corresponding hardware architecture. As for layer normalization, we extend the approximate approach for division. The extended approach can simultaneously compute square root and perform division. As a result, it avoids a large LUT for $x^{-0.5}$ but only introduces a small LUT (with 32 or fewer entries) for storing some constants. Moreover, we simplify the computation of denominator by directly removing the term regarding the square of the mean. Finally, based on the proposed algorithms, we develop hardware architectures for softmax and layer normalization. As demonstrated by our experimental results, our proposed algorithms for softmax and layer normalization can achieve better model performance and the developed architectures offer lower area cost and power consumption, when

Algorithm 1 Down-scaling softmax with online normalization

Input: x_i, $i = 0, 1, \cdots, n-1$;
Output: $y_i = f_{SM}(x_i)$, $i = 0, 1, \cdots, n-1$;
$m_{-1} \Leftarrow -\infty$;
$d_{-1} \Leftarrow 0$;
for $i = 0, 1, \cdots, n-1$ **do**
    $m_i \Leftarrow MAX\{m_{i-1}, x_i\}$;
    $d_i \Leftarrow d_{i-1}e^{m_{i-1}-m_i} + e^{x_i-m_i}$;
end for
for $i = 0, 1, \cdots, n-1$ **do**
    $y_i \Leftarrow \frac{e^{x_i-m_{n-1}}}{d_{n-1}}$;
end for

compared with the state-of-the-art design [30]. Our contributions can be briefly summarized as follows.

- For softmax, we use aggressive approximation for both the division and exponential function such that all the involved computations can be realized by addition (subtraction) and shifting. Besides, the subtraction is reused to reduce the area cost of the corresponding hardware architecture.
- For layer normalization, we extend the approximation approach for division of softmax such that a large LUT for $x^{-0.5}$ can be avoided. Besides, the square of the mean is eliminated from the denominator.
- We develop hardware architectures for the proposed algorithms of softmax and layer normalization.
- The developed architectures can work as plug-and-play units for LLM accelerators, requiring no fine-tuning and introducing negligible performance loss. They also exhibit superior hardware performance than other related designs, in terms of area cost and power consumption.

2 Background and preliminaries

In this section, we first introduce the fundamentals of softmax and layer normalization. Then we briefly review related works on hardware-oriented algorithms of softmax and layer normalization.

2.1 Fundamentals of softmax and layer normalization

Softmax is a widely used non-linear operation in deep learning. In LLMs, softmax is employed in the attention modules. The original version of softmax for n inputs can be formulated as

$$f_{SM}(x_i) = \frac{e^{x_i}}{\sum_{j=0}^{n-1} e^{x_j}},\ i = 0, 1, \cdots, n-1. \tag{1}$$

Obviously, e^x can potentially cause numerical overflow with large x. In contrast, down-scaling softmax is a more robust option which has been widely adopted in practical implementations. With $x_{max} = MAX\{x_0, x_1, \cdots, x_{n-1}\}$, it can be formulated as

$$f_{SM}(x_i) = \frac{e^{x_i-x_{max}}}{\sum_{j=0}^{n-1} e^{x_j-x_{max}}},\ i = 0, 1, \cdots, n-1. \tag{2}$$

Subtracting x_{max} from the exponents will not change the output values, but can help prevent numerical overflow.

Original softmax involves two passes over the n inputs. In the first pass, all the inputs are retrieved from the memory to compute $\sum_{j=0}^{n-1} e^{x_j}$. Then it performs the second pass over the inputs to generate the final outputs. Unfortunately, because of the requirement of subtracting x_{max} from the exponents, down-scaling softmax needs three passes over the inputs. The extra pass is used to find x_{max}. A method called online normalization can be invoked to address this issue [20]. Algorithm 1 shows the down-scaling softmax with online normalization. After the first pass, the maximum input $x_{max} = m_{n-1}$ and the denominator $d_{n-1} = \sum_{j=0}^{n-1} e^{x_j-m_{n-1}}$ are available. In the final pass, y_i is computed for each input.

Layer normalization is also a common non-linear operation in LLMs. For the sake of brevity, we also use n to denote the number of inputs of layer normalization. It is equal to the number of embedding dimensions. For n inputs x_0, x_1, $\cdots$, x_{n-1}, layer normalization can be formulated as

$$f_{LN}(x_i) = \frac{x_i - u}{\sigma}\gamma_i + \beta_i,\ i = 0, 1, \cdots, n-1, \tag{3}$$

Algorithm 2 Softmax proposed in [20]

1: **Input**: x_i, $i = 0, 1, \cdots, n-1$;
2: **Output**: $y_i = f_{SM}(x_i)$, $i = 0, 1, \cdots, n-1$;
3: $m_{-1} \Leftarrow -\infty$;
4: $d_{-1} \Leftarrow 0$;
5: **for** $i = 0, 1, \cdots, n-1$ **do**
6: $\quad m_i \Leftarrow MAX\{m_{i-1}, \lfloor x_i \rceil\}$;
7: $\quad d_i \Leftarrow d_{i-1} >> (m_i - m_{i-1}) + 2^{(x_i - m_i)_{frac}} >> |(x_i - m_i)_{int}|$;
8: **end for**
9: **for** $i = 0, 1, \cdots, n-1$ **do**
10: $\quad y_i \Leftarrow \frac{2^{(x_i - m_{n-1})_{frac}} >> |(x_i - m_{n-1})_{int}|}{d_{n-1}}$;
11: **end for**

where $u = \frac{\sum_{j=0}^{n-1} x_j}{n}$ is the mean of inputs and $\sigma = \sqrt{\frac{\sum_{j=0}^{n-1}(x_j - u)^2}{n}}$ is the standard deviation. γ_i and β_i are learnable parameters which are fixed during inference. To circumvent the third pass over the inputs, we can compute standard deviation by $\sigma = \sqrt{\frac{\sum_{j=0}^{n-1} x_j^2}{n} - (\frac{\sum_{j=0}^{n-1} x_j}{n})^2} = \sqrt{\frac{\sum_{j=0}^{n-1} x_j^2}{n} - u^2}$. After the first pass, u and $\frac{\sum_{j=0}^{n-1} x_j^2}{n}$ are computed. Before the second pass during which $f_{LN}(x_i)$ is computed for each x_i, σ is computed and stored.

In most cases, a linear layer follows directly after layer normalization. It implies that the normalized values will be multiplied with corresponding weights, i.e., $f_{LN}(x_i)w_i$. By merging γ_i into the weight w_i, layer normalization is able to be simplified:

$$f_{LN}(x_i) = \frac{x_i - u}{\sigma} + \frac{\beta_i}{\gamma_i}, \ i = 0, 1, \cdots, n-1. \tag{4}$$

Note that γ_i and w_i are merged offline, and so is the computation for $\frac{\beta_i}{\gamma_i}$. Compared with (3), (4) saves n multiplications during inference.

2.2 Related works

Ref. [20] is the first work that reports a softmax implementation without significant performance loss for Transformers. For efficient hardware implementation, it directly replaces the base e by 2. In addition, x_{max} is also rounded to the nearest integer. Algorithm 2 shows the softmax proposed in [20], in which the online normalization is also adopted. Note that for $i = 0, 1, \cdots, n-1$, m_i is an integer. As a result, multiplying with $2^{m_{i-1} - m_i}$ can be implemented by right shifting (line 7 of Algorithm 2. For exponential function 2^x, ref. [20] uses the approximation:

$$2^x = 2^{x_{int} + x_{frac}} = 2^{x_{frac}} >> |x_{int}|. \tag{5}$$

$2^{x_{frac}}$ is implemented using linear fitting. Unlike [24, 25] in which division is circumvented, Algorithm 2 employs linear fitting to compute $1/d_{n-1}$ and then multiplies it with $2^{x_i - m_{n-1}}$ (line 10 of Algorithm 2). Note that $2^{x_i - m_{n-1}}$ has been approximated as $2^{(x_i - m_{n-1})_{frac}} >> |(x_i - m_{n-1})_{int}|$. To alleviate the performance loss brought by approximation, fine-tuning is invoked in [20].

The softmax algorithm of [27] is almost the same as Algorithm 2. One difference is that line 10 is replaced by $y_i \Leftarrow 2^{x_i - m_{n-1} - \log_2 d_{n-1}}$ in the softmax algorithm of [27]. Its main innovation lies in using advanced PLAC [28] to achieve linear fitting. Namely, it approximates exponential function 2^x and logarithmic function $\log_2 x$ by PLAC, which is not considered in [20,25]. However, ref. [27] only estimates the accuracy on a single dataset and does not specify whether fine-tuning is required.

The softmax algorithm proposed in [29] is given in Algorithm 3. Since online normalization is not adopted, lines 4∼7 are used to find x_{max} and round it to the nearest integer that is no smaller than x_{max}. In lines 10 and 11, $e^{x_i - m_{n-1}}$ is approximated by $2^{(\log_2 e)(x_i - m_{n-1})} \approx 2^{1.5(x_i - m_{n-1})}$. Denoting $v = 1.5(x_i - m_{n-1})$, 2^v is approximated using (5). $2^{v_{frac}}$ is further approximated by a simple linear function $v_{frac}/2 + 1$, in which dividing by 2 is implemented using shifting in hardware (line 11 of Algorithm 3). It also adopts the idea that replaces the division by right shifting from [26] (line 16 of Algorithm 3). The difference is that Algorithm 3 rounds d_{n-1} to the nearest power-of-two integer (line 14 of Algorithm 3). In contrast, ref. [26] only considers the largest power-of-two integer which is no larger than d_{n-1}. As demonstrated in [29], the performance loss brought by Algorithm 3 is limited when applying it to Transformers. On the other hand, one significant shortcoming of Algorithm 3 is the lack

Algorithm 3 Softmax proposed in [29]

1: **Input**: x_i, $i = 0, 1, \cdots, n-1$;
2: **Output**: $y_i = f_{SM}(x_i)$, $i = 0, 1, \cdots, n-1$;
3: $m_{-1} \Leftarrow -\infty$;
4: **for** $i = 0, 1, \cdots, n-1$ **do**
5: $\quad m_i \Leftarrow MAX\{m_{i-1}, \lfloor x_i \rfloor\}$;
6: **end for**
7: $m_{n-1} \Leftarrow m_{n-1} + 1$;
8: $d_{-1} \Leftarrow 0$;
9: **for** $i = 0, 1, \cdots, n-1$ **do**
10: $\quad v \Leftarrow 1.5(x_i - m_{n-1})$;
11: $\quad p_i \Leftarrow (1 + v_{frac} >> 1) >> |v_{int}|$;
12: $\quad d_i \Leftarrow d_{i-1} + p_i$;
13: **end for**
14: $s_d \Leftarrow LOD_{round}(d_{n-1})$;
15: **for** $i = 0, 1, \cdots, n-1$ **do**
16: $\quad y_i \Leftarrow p_i >> s_d$;
17: **end for**

Algorithm 4 Softmax proposed in [30]

1: **Input**: x_i, $i = 0, 1, \cdots, n-1$;
2: **Output**: $y_i = f_{SM}(x_i)$, $i = 0, 1, \cdots, n-1$;
3: $m_{-1} \Leftarrow -\infty$;
4: $d_{-1} \Leftarrow 0$;
5: **for** $i = 0, 1, \cdots, n-1$ **do**
6: $\quad m_i \Leftarrow MAX\{m_{i-1}, x_i\}$;
7: $\quad d_i \Leftarrow d_{i-1} >> \lfloor 1.4375(m_i - m_{i-1}) \rceil + (1 >> \lfloor 1.4375(m_i - x_i) \rceil)$;
8: **end for**
9: $2^k(1 + s) \Leftarrow d_{n-1}$;
10: $sf \Leftarrow (s \geqslant 0.5)?0.568 : 0.818$;
11: **for** $i = 0, 1, \cdots, n-1$ **do**
12: $\quad ns \Leftarrow \lfloor 1.4375(m_{n-1} - x_i) \rceil + k$;
13: $\quad$ **if** $ns >= 0$ **then**
14: $\quad\quad y_i \Leftarrow sf >> ns$;
15: $\quad$ **else**
16: $\quad\quad y_i \Leftarrow sf << |ns|$;
17: $\quad$ **end if**
18: **end for**

of online normalization, resulting in three passes over the inputs. Unlike [20, 27] in which base e has been replaced by base 2, computing $d_{i-1}e^{m_{i-1}-m_i}$ brings additional hardware overhead if we introduce online normalization to Algorithm 3.

The state-of-the-art softmax design is presented in [30]. Similar to [29], it can achieve limited performance loss without fine-tuning. Moreover, online normalization is adopted, requiring fewer data reads than [29]. The softmax algorithm of [30] is given in Algorithm 4. When computing d_i (line 7 of Algorithm 4), $e^{m_{i-1}-m_i}$ is approximated by $2^{(\log_2 e)(m_{i-1}-m_i)} \approx 2^{1.4375(m_{i-1}-m_i)}$. In addition, the same approximation is applied to $e^{x_i-m_i}$. Furthermore, the exponents $1.4375(m_{i-1} - m_i)$ and $1.4375(x_i - m_i)$ are rounded to nearest integers, and thus $2^{1.4375(m_{i-1}-m_i)}$ and $2^{1.4375(x_i-m_i)}$ can be implemented simply using shifting. Lines 9∼18 correspond to the approximation for division. In line 9, $0 \leqslant s < 1$ and k is an integer. According to the value of s, the scaling factor sf is selected from two constants, 0.568 and 0.818. They are obtained by assuming $s = 1$ and $s = 0$, respectively. Moreover, an offset has been subtracted from each constant to compensate for the error caused by approximation. For more details, please refer to [30].

Due to the existence of $x^{-0.5}$, reported works on layer normalization [30–32] fail to make an efficient approximation that can avoid a large LUT. In [30], a so-called dynamic compression is proposed to reduce the hardware overhead of the square unit for computing x_i^2. It dynamically prunes the inputs at the bit level based on the magnitude of their values. As a result, the bitwidth of the square unit can be reduced from 8 bits to 4 bits. However, such optimization is built upon the 8-bit quantization scheme proposed in [33]. As a growing body of research has emerged in the field of quantization for LLMs [17, 34–41], a universal approach orthogonal to quantization is preferred. In this context, the current literature still lacks effective approximation for layer normalization.

3 Proposed algorithms for softmax and layer normalization

In this section, we first propose an approximation approach for the division of softmax. Subsequently, the proposed softmax algorithm is presented. Furthermore, the division approximation approach is extended for layer normalization and then we present the proposed layer normalization algorithm.

3.1 Approximation for division

Division is an intractable issue for efficient hardware implementations since it requires complicated architecture design, which leads to high area cost or additional clock cycles. However, if the divisor is an integer and also a power of two, division can be implemented by shifting. Based on this idea, [26] and [29] directly approximate the denominator using a power-of-two integer. In order to compensate for the error caused by approximation, offset is introduced in [30]. Similarly, the underlying idea of our approximation is to replace denominator by a power-of-two integer and compensate for the error by means of offsets.

Denote numerator and denominator by a and d, respectively. d can be represented in the form of $2^k(1+s)$, where k is an integer and $0 \leqslant s < 1$. Then we have

$$\frac{a}{d} = \frac{a}{2^k(1+s)} = \begin{cases} a(1+s)^{-1} >> k, \text{ if } k >= 0; \\ a(1+s)^{-1} << |k|, \text{ otherwise.} \end{cases} \tag{6}$$

In fact, k is the index of leading one of d. For example, 0100.1001 and 0000.0111 (both in binary format) have $k = 2$ and -2, respectively. Moreover, they have $s = 0.001001$ and $s = 0.11$, respectively. Consider a positive integer α, which is the number of bits used to represent s. For example, $s = 0.0011001$ will become $s = 0.0011$, with the restriction of $\alpha = 4$. In a sense, this is equivalent to quantizing s into α bits. The quantized s is denoted by $Q_\alpha(s)$.

Next we present a method to compensate for the error caused by quantizing s. When s is in the interval $[k_\alpha 2^{-\alpha}, (k_\alpha+1)2^{-\alpha})$, where k_α is an integer and $0 \leqslant k_\alpha < 2^\alpha$, it will be quantized into $k_\alpha 2^{-\alpha}$. In this interval, the average value of the difference between $\frac{1}{1+Q_\alpha(s)}$ and $\frac{1}{1+s}$ is

$$\begin{aligned} \frac{\int_{k_\alpha 2^{-\alpha}}^{(k_\alpha+1)2^{-\alpha}} (\frac{1}{1+k_\alpha 2^{-\alpha}} - \frac{1}{1+s}) \mathrm{d}s}{(k_\alpha+1)2^{-\alpha} - k_\alpha 2^{-\alpha}} &= \frac{1}{1+k_\alpha 2^{-\alpha}} - 2^\alpha \int_{k_\alpha 2^{-\alpha}}^{(k_\alpha+1)2^{-\alpha}} \frac{1}{1+s} \mathrm{d}s \\ &= \frac{1}{1+k_\alpha 2^{-\alpha}} - 2^\alpha \ln \frac{1+(k_\alpha+1)2^{-\alpha}}{1+k_\alpha 2^{-\alpha}}. \end{aligned} \tag{7}$$

Here we assume that s is uniformly distributed in the interval $[k_\alpha 2^{-\alpha}, (k_\alpha+1)2^{-\alpha})$. The average value varies across the intervals. We regard it as an offset and subtract it from $\frac{1}{1+Q_\alpha(s)}$ for error compensation. When s is in the interval $[k_\alpha 2^{-\alpha}, (k_\alpha+1)2^{-\alpha})$, the approximation for division can be written as

$$\frac{a}{d} = \begin{cases} 2^\alpha \ln \frac{1+(k_\alpha+1)2^{-\alpha}}{1+k_\alpha 2^{-\alpha}} a >> k, \text{ if } k >= 0; \\ 2^\alpha \ln \frac{1+(k_\alpha+1)2^{-\alpha}}{1+k_\alpha 2^{-\alpha}} a << |k|, \text{ otherwise.} \end{cases} \tag{8}$$

Note that the scaling factor $2^\alpha \ln \frac{1+(k_\alpha+1)2^{-\alpha}}{1+k_\alpha 2^{-\alpha}}$ is a constant for the interval $[k_\alpha 2^{-\alpha}, (k_\alpha+1)2^{-\alpha})$. Now consider a specific example in which $d = 5.6$, and $\alpha = 4$. Obviously, we have $k = 2$ and $s = 0.4$ if we denote $d = 2^k(1+s)$. Since 0.4 lies in the interval $[6\times 2^{-4}, 7\times 2^{-4})$, the final output is $2^4 \ln \frac{1+7\times 2^{-4}}{1+6\times 2^{-4}} a >> 2 = 0.1778a$. Without the error compensation, it will be $\frac{1}{1+6\times 2^{-4}} a >> 2 = 0.1818a$. Note that it will be $0.1786a$ if floating-point arithmetic is used.

In [26], the denominator is approximated using the largest power-of-two integer which is no larger than it. This is equivalent to removing s from (6). In [29], the denominator is rounded to the nearest power-of-two integer, which is equivalent to quantizing s into 0 and 1 when $0 \leqslant s < 0.5$ and $0.5 \leqslant s < 1$, respectively. Compared with [29], ref. [30] additionally considers the error compensation and modifies the scaling factors from 0.5 and 1 to 0.568 and 0.818. It is similar to the special case of our approach with $\alpha = 1$. In contrast, our approach can achieve better performance by setting α to larger values.

Algorithm 5 Our proposed softmax (without subtraction reuse)

1: **Input**: α, x_i, $i = 0, 1, \cdots, n-1$;
2: **Output**: $y_i = f_{SM}(x_i)$, $i = 0, 1, \cdots, n-1$;
3: $m_{-1} \Leftarrow -\infty$;
4: $d_{-1} \Leftarrow 0$;
5: **for** $i = 0, 1, \cdots, n-1$ **do**
6: $\quad m_i \Leftarrow MAX\{m_{i-1}, Trunc_{2k}(x_i)\}$;
7: $\quad$ **if** $x_i - m_i >= 0$ **then**
8: $\quad\quad d_i \Leftarrow d_{i-1} >> 1.5(m_i - m_{i-1}) + 1 << Trunc(1.5(x_i - m_i))$;
9: $\quad$ **else**
10: $\quad\quad d_i \Leftarrow d_{i-1} >> 1.5(m_i - m_{i-1}) + 1 >> Trunc(1.5(m_i - x_i))$;
11: $\quad$ **end if**
12: **end for**
13: $2^k(1+s) \Leftarrow d_{n-1}$;
14: Identify the interval $[k_\alpha 2^{-\alpha}, (k_\alpha+1)2^{-\alpha})$ that s lies in;
15: **for** $i = 0, 1, \cdots, n-1$ **do**
16: $\quad ns \Leftarrow Trunc(1.5(m_{n-1} - x_i)) + k$;
17: $\quad$ **if** $ns >= 0$ **then**
18: $\quad\quad y_i \Leftarrow 2^\alpha \ln \frac{1+(k_\alpha+1)2^{-\alpha}}{1+k_\alpha 2^{-\alpha}} >> ns$;
19: $\quad$ **else**
20: $\quad\quad y_i \Leftarrow 2^\alpha \ln \frac{1+(k_\alpha+1)2^{-\alpha}}{1+k_\alpha 2^{-\alpha}} << |ns|$;
21: $\quad$ **end if**
22: **end for**

Algorithm 6 Sub-algorithm with subtraction reuse for our proposed softmax (acting as a substitute for lines 5–12 of Algorithm 5)

1: **for** $i = 0, 1, \cdots, n-1$ **do**
2: $\quad \Delta \Leftarrow Trunc_{2k}(x_i) - m_{i-1}$;
3: $\quad$ **if** $\Delta >= 0$ **then**
4: $\quad\quad \Delta_m \Leftarrow \Delta$;
5: $\quad\quad \Delta_x \Leftarrow < (x_i)_0, (x_i)_{frac} >$;
6: $\quad$ **else**
7: $\quad\quad \Delta_m \Leftarrow 0$;
8: $\quad\quad \Delta_x \Leftarrow < \Delta, (x_i)_0, (x_i)_{frac} >$;
9: $\quad$ **end if**
10: $\quad$ **if** $\Delta >= 0$ **then**
11: $\quad\quad d_i \Leftarrow d_{i-1} >> 1.5\Delta_m + 1 << Trunc(1.5\Delta_x)$;
12: $\quad$ **else**
13: $\quad\quad d_i \Leftarrow d_{i-1} >> 1.5\Delta_m + 1 >> Trunc(-1.5\Delta_x)$;
14: $\quad$ **end if**
15: **end for**

3.2 Approximation for softmax

Algorithm 5 presents the algorithm of our proposed softmax. Online normalization is adopted to reduce the number of data reads. The exponential function is approximated as $e^x = 2^{(\log_2 e)x} \approx 2^{1.5x}$, which is the same as [29]. With online normalization, the computation for d_i becomes $d_i \Leftarrow d_{i-1}2^{1.5(m_{i-1}-m_i)} + 2^{1.5(x_i-m_i)}$. To make the exponential function $2^{1.5(x_i-m_i)}$ able to be implemented using shifting, we only retain the integer part of $1.5(x_i - m_i)$, directly using truncation. In addition, we truncate x_i when computing m_i such that m_i is always a multiple of 2, i.e., an even number. This operation is denoted as $Trunc_{2k}$. It implies that $1.5(m_i - m_{i-1})$ is an integer and $d_{i-1}2^{1.5(m_{i-1}-m_i)}$ can be implemented using shifting (lines 8 and 10 of Algorithm 5). Note that we retain the bits using truncation and therefore we probably have $x_i - m_i > 0$ (lines 7 and 8 of Algorithm 5). Lines 13–22 correspond to the approximation for division which is discussed in previous subsection.

When comparing m_{i-1} and $Trunc_{2k}(x_i)$ (line 6 of Algorithm 5), we actually calculate their difference and determine whether it is negative. Besides, there are two other subtractions in each loop of lines 5–12 of Algorithm 5: $m_i - m_{i-1}$ and $x_i - m_i$. In fact, there is an opportunity to reuse subtraction. Algorithm 6 presents the reusing sub-algorithm for lines 5–12 of Algorithm 5. $(x_i)_{frac}$ and $(x_i)_0$ denote the fractional part and the least-significant integer bit of x_i, respectively. $< A, B >$ is the concatenation of A and B. In this sub-algorithm, only one subtraction is required in each loop. For a software implementation, Algorithm 5 is sufficient. However, we propose to use Algorithm 6 to replace lines 5–12 of Algorithm 5, for the sake of a hardware architecture with lower area cost.

3.3 Approximation for layer normalization

Directly adopting the approximation approach proposed for the division of softmax cannot completely solve the inefficiency issue of hardware implementation of layer normalization, due to the existence of calculation for square root. Next we extend the approximation approach, making it able to simultaneously compute square root and perform division in layer normalization.

Using the same notation as Section 3.1, we have

$$\frac{a}{\sqrt{d}}=\frac{a}{\sqrt{2^k(1+s)}}=\begin{cases}a(1+s)^{-0.5}>>\frac{k}{2}, \text{ if } k \text{ is even and } k>=0;\\ a(1+s)^{-0.5}<<\frac{|k|}{2}, \text{ if } k \text{ is even and } k<0;\\ a(2+2s)^{-0.5}>>\lfloor\frac{k}{2}\rfloor, \text{ if } k \text{ is odd and } k>=0;\\ a(2+2s)^{-0.5}<<\lceil\frac{|k|}{2}\rceil, \text{ otherwise.}\end{cases}\tag{9}$$

When s is in the interval $[k_\alpha 2^{-\alpha},(k_\alpha+1)2^{-\alpha})$, the average value of the difference between $\frac{1}{\sqrt{1+Q_\alpha(s)}}$ and $\frac{1}{\sqrt{1+s}}$ is

$$\begin{aligned}\frac{\int_{k_\alpha 2^{-\alpha}}^{(k_\alpha+1)2^{-\alpha}}(\frac{1}{\sqrt{1+k_\alpha 2^{-\alpha}}}-\frac{1}{\sqrt{1+s}})\mathrm{d}s}{(k_\alpha+1)2^{-\alpha}-k_\alpha 2^{-\alpha}}&=\frac{1}{\sqrt{1+k_\alpha 2^{-\alpha}}}-2^\alpha\int_{k_\alpha 2^{-\alpha}}^{(k_\alpha+1)2^{-\alpha}}\frac{1}{\sqrt{1+s}}\mathrm{d}s\\&=\frac{1}{\sqrt{1+k_\alpha 2^{-\alpha}}}-2^{\alpha+1}(\sqrt{1+(k_\alpha+1)2^{-\alpha}}-\sqrt{1+k_\alpha 2^{-\alpha}}).\end{aligned}\tag{10}$$

Similar to the proposed softmax, we regard the average value as an offset and subtract it from $\frac{1}{\sqrt{1+Q_\alpha(s)}}$. When $s\in[k_\alpha 2^{-\alpha},(k_\alpha+1)2^{-\alpha})$, (9) becomes

$$\frac{a}{\sqrt{d}}=\begin{cases}2^{\alpha+1}(\sqrt{1+(k_\alpha+1)2^{-\alpha}}-\sqrt{1+k_\alpha 2^{-\alpha}})a>>\frac{k}{2}, \text{ if } k \text{ is even and } k>=0;\\ 2^{\alpha+1}(\sqrt{1+(k_\alpha+1)2^{-\alpha}}-\sqrt{1+k_\alpha 2^{-\alpha}})a<<\frac{|k|}{2}, \text{ if } k \text{ is even and } k<0;\\ 2^{\alpha+0.5}(\sqrt{1+(k_\alpha+1)2^{-\alpha}}-\sqrt{1+k_\alpha 2^{-\alpha}})a>>\lfloor\frac{k}{2}\rfloor, \text{ if } k \text{ is odd and } k>=0;\\ 2^{\alpha+0.5}(\sqrt{1+(k_\alpha+1)2^{-\alpha}}-\sqrt{1+k_\alpha 2^{-\alpha}})a<<\lceil\frac{|k|}{2}\rceil, \text{ otherwise.}\end{cases}\tag{11}$$

Once again, we emphasize that $2^{\alpha+1}(\sqrt{1+(k_\alpha+1)2^{-\alpha}}-\sqrt{1+k_\alpha 2^{-\alpha}})$ and $2^{\alpha+0.5}(\sqrt{1+(k_\alpha+1)2^{-\alpha}}-\sqrt{1+k_\alpha 2^{-\alpha}})$ are two constants for interval $[k_\alpha 2^{-\alpha},(k_\alpha+1)2^{-\alpha})$.

Algorithm 7 presents the proposed algorithm for layer normalization. Similar to the online-normalization version of softmax, it only needs two passes over the inputs. The original denominator is $\sigma=\sqrt{\frac{\sum_{i=0}^{n-1}x_i^2}{n}-u^2}$, which is simplified by removing u^2 in our algorithm. In line 10 of Algorithm 7, uu denotes the simplified denominator. In Subsection 5.2, our experimental results will demonstrate that the introduced performance loss is negligible. Lines 11–24 correspond to the approximation for division and square root.

4 Hardware architectures for the proposed algorithms

The design of the above algorithms is aimed at achieving efficient hardware implementation while maintaining the model performance. This section develops hardware architectures for the proposed algorithms of softmax and layer normalization.

Figure 1 depicts the architecture of the proposed softmax, which is in a pipelined manner. The upper part of the architecture corresponds to the first loop of the algorithm, while the lower part corresponds to the second one. After n cycles, d_{n-1} and m_{n-1} are available in the upper two registers. Then they are fed into the lower registers for the following computations of n cycles. There are two sets of inputs: $x_i^{(0)}$ and $x_i^{(1)}$, for $i=0,1,\cdots,n-1$. The unit labelled as "Trunc-2k" is used to exclude the fractional part and the first integer bit from x_i, i.e., computing $Trunc_{2k}(x_i)$ (line 2 of Algorithm 6). The unit labeled as "Rem" is used to output the excluded bits, i.e., $<(x_i)_0,(x_i)_{frac}>$ (line 5 of Algorithm 6). Since $1.5x=x+x>>1$, multiplication with 1.5 can be implemented by an adder. Rounding operation will lead to additional hardware overhead, due to the need of checking for carry and the possible carry

Algorithm 7 Our proposed layer normalization

Input: x_i, γ_i, β_i $i = 0, 1, \cdots, n-1$;
Output: $y_i = f_{LN}(x_i)$, $i = 0, 1, \cdots, n-1$;
$u_{-1} \Leftarrow 0$;
$uu_{-1} \Leftarrow 0$;
for $i = 0, 1, \cdots, n-1$ **do**
  $u_i \Leftarrow u_{i-1} + x_i$;
  $uu_i \Leftarrow uu_{i-1} + x_i^2$;
end for
$u \Leftarrow \frac{u_{n-1}}{n}$;
$uu \Leftarrow \frac{uu_{n-1}}{n}$;
$2^k(1+s) \Leftarrow uu$;
Identify the interval $[k_\alpha 2^{-\alpha}, (k_\alpha+1)2^{-\alpha})$ that s lies in;
for $i = 0, 1, \cdots, n-1$ **do**
  if k is even and $k >= 0$ **then**
    $y_i \Leftarrow 2^{\alpha+1}(\sqrt{1+(k_\alpha+1)2^{-\alpha}} - \sqrt{1+k_\alpha 2^{-\alpha}})(x_i - u) >> \frac{k}{2}$;
  else if k is even and $k < 0$ **then**
    $y_i \Leftarrow 2^{\alpha+1}(\sqrt{1+(k_\alpha+1)2^{-\alpha}} - \sqrt{1+k_\alpha 2^{-\alpha}})(x_i - u) << \frac{|k|}{2}$;
  else if k is odd and $k >= 0$ **then**
    $y_i \Leftarrow 2^{\alpha+0.5}(\sqrt{1+(k_\alpha+1)2^{-\alpha}} - \sqrt{1+k_\alpha 2^{-\alpha}})(x_i - u) >> \lfloor \frac{k}{2} \rfloor$;
  else
    $y_i \Leftarrow 2^{\alpha+0.5}(\sqrt{1+(k_\alpha+1)2^{-\alpha}} - \sqrt{1+k_\alpha 2^{-\alpha}})(x_i - u) << \lceil \frac{|k|}{2} \rceil$;
  end if
  $y_i \Leftarrow y_i\gamma_i + \beta_i$;
end for

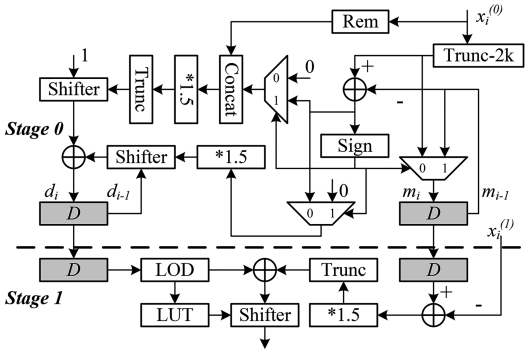

Figure 1 Hardware architecture of the proposed softmax.

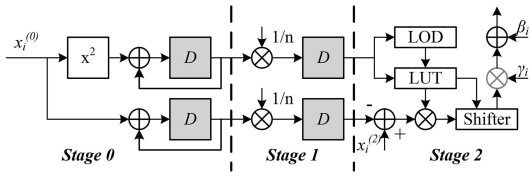

Figure 2 Hardware architecture of the proposed layer normalization.

propagation. In contrast, our architecture directly performs truncating at the bit level, leading to no hardware overhead. Leading one detector (LOD) is used to identify k and s from d_{n-1}. As α is a fixed integer which is defined in advance, the number of possible scaling factors $2^\alpha \ln \frac{1+(k_\alpha+1)2^{-\alpha}}{1+k_\alpha 2^{-\alpha}}$ (lines 18 and 20 of Algorithm 5) is also fixed. For example, when $\alpha = 2$, there are $2^2 = 4$ possible scaling factors for the four intervals: $[0, 0.25)$, $[0.25, 0.5)$, $[0.5, 0.75)$ and $[0.75, 1)$. A small LUT is employed to store the scaling factors.

Figure 2 depicts the architecture of the proposed layer normalization, which is also in a pipelined manner. There are three parts in total, with the first one and the last one corresponding to the two loops of Algorithm 7, respectively. After n cycles, the values of $\sum_{i=0}^{n-1} x_i^2$ and $\sum_{i=0}^{n-1} x_i$ are available in the two registers on the left. In the following cycle, the two values are multiplied with $\frac{1}{n}$ and transferred into the two registers on the right. Subsequently, they will be involved in the computations of the second loop. Differing from the proposed architecture of softmax, both k and s are fed into the LUT employed in Figure 2. As we have discussed in Section 2.1, γ_i can be merged into weights. In this case, the multiplier for γ_i can be removed.

5 Experimental results and comparisons

In this section, we present experimental results and comparisons with related designs to demonstrate the effectiveness and superiority of our work.

Table 1 Bitwidth configuration of the proposed softmax architecture for OPT

Variable	x_i	y_i	$Trunc(1.5(m_i - x_i))$	d_i (upper registers)	d_i (lower registers)	Scaling factor
(sign,int,frac)	(1,12,4)	(0,1,14)	(1,4,0)	(0,12,11)	(0,12,1)	(0,0,8)

Table 2 Bitwidth configuration of the proposed layer normalization architecture for OPT

Variable	x_i	y_i	$\frac{1}{n}$	u_i	u
(sign,int,frac)	(1,9,9)	(1,7,12)	(0,0,18)	(1,9,9)	(0,0,11)
Variable	x_i^2	uu_i (left registers)	uu_i (inputs to stage 1)	uu	Scaling factor
(sign,int,frac)	(0,18,14)	(0,18,14)	(0,18,3)	(0,7,15)	(0,0,8)

5.1 Experimental settings

Our experiments are based on two classic LLMs, BERT [42] and OPT [3], both of which are representative models in their respective eras. For BERT, we follow the convention of the most related works [11, 29, 30, 42] and evaluate BERT-base on a multi-task benchmark GLUE [43], which includes nine natural language understanding (NLU) tasks. For OPT, we evaluate OPT-1.3b, OPT-6.7b, and OPT-13b on three language modeling tasks, i.e., WikiText2 (WIKI) [44], Pen Treebank (PT) [45], and C4 [46], following the work of [41]. The software code in our experiments is implemented using PyTorch. We replace the original softmax or/and layer normalization by our proposed ones, while keeping the other parts of the models unchanged. We implement the hardware architectures in Verilog HDL and synthesize them using Synopsys Design Compiler with a 65-nm CMOS technology library, in order to estimate area and power. The voltage is set to 1.08 V. Note that there is one exception. When compared with the softmax architecture of [27], we use a 90-nm CMOS technology with a voltage of 1 V. As [27] evaluates its design using a 90-nm CMOS technology, we follow it for a fair comparison.

Quantization scheme may have a significant impact on the hardware performance due to the difference in resulting bitwidths. Note that our designs are independent of specific quantization schemes. Consequently, when making a comparison, we adopt the same naive quantization (directly truncate the bits for larger or smaller values) for all the involved designs, for the sake of fairness. Our goal in determining the quantization bitwidth is to avoid significant performance loss. Our experimental results revealed differences between the quantization bitwidth configurations of BERT and OPT. If we report the results for both the quantization bitwidth configurations, it will appear redundant. Without loss of generality, we only consider quantization for OPT in this paper, for the sake of simplicity. When adopting quantization for OPT, we only quantize the softmax or/and the layer normalization blocks while the other parts of the models remain unchanged. The bitwidth configurations for the proposed architectures of softmax and layer normalization are provided in Tables 1 and 2, respectively. When reproducing the softmax architecture proposed in [30], we use the same bitwidth configuration, except that the number of integer bits of d_i is 11. When reproducing the softmax architecture proposed in [29], we also use the same bitwidth configuration, except that $1.5(m_i - x_i)$ is quantized using $(1, 5, 2)$, and d_i is quantized using $(0, 10, 13)$ and $(0, 10, 5)$ for the upper and lower registers, respectively.

As there is no work reporting their implementation details for $x^{-0.5}$ of layer normalization, we employ PLAC of [28] to perform linear fitting for it. It is also employed in the softmax approximation of [27]. PLAC uses several linear segments $kx + b$ to approximate the target function, i.e., $x^{-0.5}$ in layer normalization. Its hardware architecture is composed of a multiplier, an adder and an LUT for storing the coefficients of all the segments. In our experiments, the number of linear segments is set to 24, while k and b are quantized using (1,15,12) and (0,6,7).

5.2 Model performance

Table 3 presents the accuracy results for BERT, while Table 4 presents the perplexity scores (lower is better) for OPT. We first evaluate the impact of the softmax approximation on model performance. For BERT, refs. [29, 30] and our proposed design with $\alpha = 1$ achieve similar average accuracy. When increasing α to 4, our proposed design outperforms the others. For OPT-6.7b and OPT-13b, [29], [30] and our proposed design with $\alpha = 1$ also have similar perplexity scores, while our proposed design with $\alpha = 4$ performs slightly better. However, the performance gap becomes more pronounced for OPT-1.3b. It shows that [30] and our proposed design with $\alpha = 1$ can decrease the average perplexity score by more than 0.1 when compared with [29]. It is not surprising since [30] and our proposed design both introduce

Table 3 Accuracy (↑) comparison of BERTs that adopt different implementations of softmax or/and layer normalization

Approach	CoLA	SST-2	MRPC	STS-B	QQP	MNLI	QNLI	RTE	WNLI	Average
Baseline	58.80	92.09	88.89	88.57	90.76	83.62	91.31	65.70	56.34	79.56
Only softmax										
Base 2 [27]	57.82	91.74	88.82	88.13	90.13	83.26	90.76	65.34	57.75	79.31
[29]	58.81	91.86	89.19	88.22	90.61	83.18	91.05	63.90	56.34	79.32
[30]	58.55	91.86	88.74	88.44	90.76	83.80	91.01	64.62	56.34	79.35
Prop. ($\alpha = 1$)	59.05	91.97	88.89	88.32	90.78	83.62	90.76	64.98	56.34	79.41
Prop. ($\alpha = 4$)	59.56	92.32	89.50	88.43	90.74	83.45	91.20	65.34	56.34	79.65
Only layer normalization										
Prop. ($\alpha = 1$)	60.10	91.86	89.12	88.11	90.64	83.28	90.81	63.54	56.34	79.31
Prop. ($\alpha = 4$)	58.81	92.09	88.74	88.56	90.80	83.70	91.38	65.34	56.34	79.53
Both softmax and layer normalization										
Prop. Softmax ($\alpha = 1$) + **Prop. LayerNorm** ($\alpha = 4$)	59.05	91.97	89.35	88.37	90.72	83.37	90.96	65.70	56.34	79.54
Prop. Softmax ($\alpha = 4$) + **Prop. LayerNorm** ($\alpha = 4$)	59.31	92.43	88.96	88.44	90.66	83.53	91.18	65.70	56.34	79.62

offsets to compensate for approximation error. Moreover, we can see that the quantization will lead to negligible performance loss.

The softmax design of [27] directly replaces base e by base 2. Moreover, it employs the PLAC of [28] to perform linear fitting for exponential and logarithmic functions. For simplicity, we only replace the base but do not approximate the exponential and logarithmic functions. The resulting accuracy should not be worse than that of [27]. As demonstrated in Table 4, the base replacement leads to unacceptable performance loss for OPT models. The softmax proposed in [20] also uses base 2. However, fine-tuning is leveraged to improve the performance. Unfortunately, it is extremely expensive to fine-tune large models with billions of parameters, such as the OPT models considered in this paper. Therefore, our design is more promising since it requires no fine-tuning.

Unlike softmax, layer normalization is more sensitive to our approximation. It is shown that α needs to be set to 4, in order to achieve similar accuracy to the baseline. With $\alpha = 1$, the performance loss becomes significant, especially for OPT models. Henceforth, we adopt $\alpha = 4$ for our proposed layer normalization. Furthermore, we can see that the performance loss is negligible if we set α to 4 for both the softmax and layer normalization. In addition, the layer normalization in which $x^{-0.5}$ is implemented using PLAC can achieve similar performance, in conjunction with our proposed softmax with $\alpha = 4$.

Note that our goal is not to devise approximate algorithms to achieve better model performance than other designs for softmax or/and layer normalization. Instead, we aim to provide hardware-oriented algorithms that offer low-complexity hardware architectures for softmax and layer normalization, at the cost of negligible model performance loss.

5.3 Hardware implementation and discussion

Regarding the softmax design, refs. [27, 29, 30] are the recently proposed works. As we have discussed before, ref. [30] presents the state-of-the-art approximation for softmax. The softmax architecture proposed in [27] also exhibits impressive hardware results. While both of [27] and [30] employ the online normalization, the softmax architecture of [29] still uses the naive computing manner, which requires three passes over the inputs. It is also in a pipelined manner, implying that three buffers are required for the three stages: finding maximum values, computing the denominator, and outputting. In contrast, the online normalization allows the softmax architectures of [27] , [30] and ours to employ only two buffers. Additionally, the model performance of the softmax of [29] is inferior to that of [30] and ours. As a result, we only consider the softmax designs of [27] and [30] for hardware comparison.

The hardware implementation results for softmax architectures are provided in Table 5. When compared with [30], we use the bitwidth configuration given in Table 1. The clock period is set to 5 ns in order to eliminate the impact that timing constraint imposes on the area cost. A larger α increases the area cost of LOD and LUT of Figure 1. As revealed by Table 4, the proposed softmax architecture can achieve almost the same model performance as that of [30], when setting α to 1. From the hardware

Table 4 Perplexity (↓) comparison of OPT models that adopt different implementations of softmax or/and layer normalization

Approach	OPT-1.3b				OPT-6.7b				OPT-13b			
	WIKI	**PT**	**C4**	**Average**	**WIKI**	**PT**	**C4**	**Average**	**WIKI**	**PT**	**C4**	**Average**
Baseline	14.62	16.96	14.72	15.43	10.86	13.09	11.74	11.90	10.13	12.34	11.20	11.22
Only softmax												
Base 2 [27]	17.88	23.90	16.77	19.52	13.12	16.65	12.79	14.19	11.56	13.52	11.94	12.34
[29]	14.79	17.29	14.91	15.66	10.93	13.27	11.85	12.02	10.15	12.48	11.28	11.30
[30]	14.73	17.07	14.79	15.53	10.95	13.15	11.78	11.96	10.24	12.39	11.23	11.29
Prop. ($\alpha = 1$)	14.73	17.08	14.82	15.54	10.91	13.17	11.80	11.96	10.19	12.41	11.24	11.28
Prop. ($\alpha = 4$)	14.63	17.03	14.76	15.47	10.85	13.13	11.77	11.92	10.12	12.37	11.22	11.24
Quant. [29]	14.71	17.33	14.94	15.66	10.87	13.29	11.87	12.01	10.13	12.50	11.29	11.31
Quant. [30]	14.70	17.09	14.79	15.53	10.92	13.15	11.78	11.95	10.23	12.39	11.23	11.28
Quant. Prop. ($\alpha = 1$)	14.69	17.08	14.82	15.53	10.89	13.18	11.80	11.96	10.15	12.42	11.24	11.27
Quant. Prop. ($\alpha = 4$)	14.55	17.07	14.80	15.47	10.78	13.14	11.78	11.90	10.06	12.39	11.23	11.23
Only layer normalization												
Prop. ($\alpha = 1$)	15.04	17.64	15.12	15.93	11.68	14.35	12.31	12.78	11.70	15.37	12.83	13.30
Prop. ($\alpha = 4$)	14.62	16.97	14.73	15.44	10.87	13.10	11.75	11.91	10.14	12.36	11.21	11.24
PLAC	14.62	16.97	14.73	15.44	10.89	13.08	11.75	11.91	10.15	12.34	11.21	11.23
Quant. Prop. ($\alpha = 1$)	15.09	17.68	15.16	15.98	11.65	14.35	12.32	12.77	11.73	15.92	12.85	13.50
Quant. Prop. ($\alpha = 4$)	14.62	16.96	14.73	15.44	10.87	13.10	11.74	11.90	10.14	12.36	11.21	11.24
Both softmax and layer normalization												
Prop. Softmax ($\alpha = 1$) + **Prop. LayerNorm** ($\alpha = 4$)	14.73	17.11	14.82	15.55	10.92	13.18	11.80	11.97	10.20	12.44	11.25	11.30
Prop. Softmax ($\alpha = 4$) + **Prop. LayerNorm** ($\alpha = 4$)	14.63	17.03	14.77	15.48	10.85	13.14	11.77	11.92	10.13	12.39	11.23	11.25
Quant. Pro. Softmax ($\alpha = 1$) + LayerNorm with PLAC	14.68	17.13	14.83	15.55	10.90	13.18	11.80	11.90	10.18	12.43	11.24	11.28
Quant. Pro. Softmax ($\alpha = 4$) + LayerNorm with PLAC	14.55	17.07	14.80	15.47	10.82	13.16	11.78	11.92	10.09	12.39	11.23	11.24
Quant. Prop. Softmax ($\alpha = 1$) + **Quant. Prop. LayerNorm** ($\alpha = 4$)	14.70	17.11	14.83	15.55	10.89	13.18	11.79	11.95	10.16	12.44	11.25	11.28
Quant. Prop. Softmax ($\alpha = 4$) + **Quant. Prop. LayerNorm** ($\alpha = 4$)	14.57	17.06	14.81	15.48	10.78	13.15	11.78	11.90	10.08	12.41	11.24	11.24

Table 5 Hardware implementation results of different softmax architectures[a)]

Design	**Tech.** (nm)	**Bitwidth**	**Clock period** (ns)	**Area** (μm^2)	**Power** (mW)	**Remark**
[30]	65	As in Table 1	5	3051.60	0.23	Without fine-tuning
Prop. ($\alpha = 1$)				2336.00	0.19	
Prop. ($\alpha = 4$)				2492.40	0.19	
[27]	90	That used in [27]	2	3052.43	0.34	Require fine-tuning
Prop. ($\alpha = 1$)				2656.58	0.35	Without fine-tuning
Prop. ($\alpha = 4$)				2765.25	0.36	

a) Voltage is not given in [27].

perspective, it can additionally save 23.45% area cost and 17.39% power consumption.

Since the details about the approximation on exponential and logarithmic functions (such as the bitwidths used for the coefficients of the linear segments) are not given in [27], it is hard to make a reasonable reproduction. Thereby, we directly cite the hardware implementation results reported in [27] and evaluate our design with a 90-nm technology, which is consistent with that of [27]. Besides, we adopt the bitwidth configuration of [27] to implement our proposed softmax architecture. With $\alpha = 1$ and $\alpha = 4$, the proposed architecture outperforms that of [27] in terms of area cost. On the other hand, the power consumption is slightly higher (less than 6%) than that of [27]. Note that voltage is not given in [27]. Since [27] only estimates its approximate model on a single dataset, it is uncertain whether such an aggressive approximation would still guarantee accuracy in other tasks. As shown in Table 4, the

Table 6 Hardware implementation results of different layer normalization architectures[a)]

Design	**Tech.** (nm)	**Bitwidth**	**Clock period** (ns)	**Area** (μm^2)	**Power** (mW)
With γ					
PLAC	65	As in Table 2	20	25342.40	0.35
Prop. ($\alpha = 4$)				17056.00	0.30
PLAC			11.1	28882.40	0.72
Prop. ($\alpha = 4$)			8.0	19685.60	0.94
Without γ					
PLAC	65	As in Table 2	20	22062.40	0.25
Prop. ($\alpha = 4$)				13730.40	0.22
PLAC			7.7	24936.80	0.70
Prop. ($\alpha = 4$)			4.2	17388.00	1.31

a) Minimum achievable clock period.

performance loss brought by [27] is unacceptable if fine-tuning is not applied.

In implementing layer normalization, the most straightforward approach is to deploy a unit to compute $x^{-0.5}$ [30, 31]. Therefore, we consider another layer normalization architecture that replaces the LOD, LUT and the shifter of Figure 1 with a $x^{-0.5}$ unit. In this paper, we adopt PLAC [28] to implement the $x^{-0.5}$ unit. Table 6 presents the hardware implementation results of different layer normalization architectures. When setting the clock period to 20 ns, our goal is to eliminate the impact that timing constraint imposes on the area cost. With $\alpha = 4$, the LUT employed in our proposed architecture has $2^5 = 32$ entries for storing the scaling factors. It shows that our proposed architecture can reduce 32.70% area cost and 14.29% power consumption when compared with the PLAC-based architecture. As revealed by the values of minimum achievable clock period, our proposed architecture also enjoys a shorter critical path. As mentioned before, we can further remove a multiplier if γ is merged with weights. Table 6 also gives the implementation results without the multiplier corresponding to γ. Obviously, both the area cost and power consumption can be further saved.

In addition to the works dedicated to the softmax and layer normalization unit design, there exist some works that attempt to use an LUT to approximate various non-linear functions. Both [32] and [47] propose trainable LUTs, acting as universal approximators for non-linear functions such as exponential, division and square root. Softmax and layer normalization can also be implemented by employing multiple such LUTs or a single LUT for multiple cycles, along with the addition of some other hardware units. Table 7 provides the comparisons between our designs and the LUTs of [32] and [47]. Note that a complete softmax or layer normalization unit needs more than an LUT of [32] or [47]. In other words, for a softmax or layer normalization unit, the hardware cost of [32] and [47] has been underestimated in this table. For example, the LOD and LUT modules account for no more than 5% area overhead in our proposed layer normalization unit (Figure 2). If using the same architecture shown in Figure 2 (just replacing LOD and LUT with the trainable LUTs), the area cost for [32] and [47] will be increased by approximately 16518.6 um^2 ($17388.00 \times 95\%$). Although hardware cost of [32] and [47] is underestimated in Table 7, our designs still have lower area cost and better area-delay product (ADP). Moreover, both [32] and [47] require training the parameters of their LUTs, with [47] even requiring model fine-tuning. In contrast, our designs have no requirement on training nor fine-tuning.

The significance of softmax and layer normalization units within an accelerator is profoundly influenced by the architectural design of the accelerator. If the linear units responsible for the multiply-accumulate operations have higher parallelism, the proportion of softmax and layer normalization units in the overall hardware cost could be relatively small. In some recently proposed Transformer accelerator designs [21, 22], softmax and layer normalization units occupy a significant area proportion in the compute modules. For example, softmax and layer normalization units account for 44.7% and 10.3% area cost in [21], respectively. Recall that our proposed softmax and layer normalization designs can respectively save 23.45% and 32.70% area cost when compared with the state-of-the-art designs. Although [21] provides no details on the architectures of softmax and layer normalization, we can expect that replacing its softmax and layer normalization units with our designs would result in a larger reduction in area cost, exceeding 23.45% and 32.70%, respectively. Consequently, the overall reduction on area cost will be larger than $44.7\% \times 23.45\% + 10.3\% \times 32.70\% = 13.85\%$. In [22], softmax and layer normalization units account for 64.5% and 11.6% area cost, respectively. Similarly, our designs are expected to save the overall area cost by more than $64.5\% \times 23.45\% + 11.6\% \times 32.70\% = 18.92\%$. In addition to reducing the

Table 7 Rough comparisons with non-linear units based on LUT[a)b)c)]

Design		Area (μm^2)	Clock period (ns)	Average cycles	ADP ($\mu m^2 \times$ ns)	Remark
Softmax	**Prop.** ($\alpha = 1$)	2336.00	5.0	1	11680.00	Without fine-tuning
	Prop. ($\alpha = 4$)	2492.40			12462.00	
	[32]	42972.56	12.6	6 (2EXP+DIV)	3256460.60	Require training LUT parameters
	[47]	18974.78	4.6	3 (2EXP+DIV)	264128.94	Require training LUT parameters and also fine-tuning the models
LayerNorm	**Prop.** ($\alpha = 4$)	17388.00	4.2	1	73029.60	Without fine-tuning
	[32]	42972.56	12.6	2 (1/SQRT)	1082908.51	Require training LUT parameters
	[47]	18974.78	4.6	1 (RSQRT)	87283.99	Require training LUT parameters and also fine-tuning the models

a) In this table, we only consider one LUT of [32] or [47]. For [32], we choose the LUT with precision of FP16 and 16 entries. For [47], we choose the LUT with precision of INT16 and 16 entries. The area and clock period of [32] and [47] have been scaled to 65-nm technology based on the scaling equations used in [48] and [49]: $Area_{tec}/Area_{65} = s^2$ and $T_{65}/T_{tec} = s$, where s=Technology/65 nm.

b) In this table, the softmax hardware cost for [32] and [47] does not include that of the module for finding the maximum value (line 6 of Algorithm 1) and registers.

c) In this table, the layer normalization hardware cost for [32] and [47] does not include that of the square unit, the multiplier for $1/\sigma$, the multiplier for $1/n$ and registers.

hardware overhead of existing Transformer accelerator designs, we emphasize that our work also provides important insights for future research involving softmax and layer normalization.

6 Conclusion

In this paper, we present hardware-oriented algorithms for softmax and layer normalization of LLMs. We propose an approximate approach for the division of softmax and then extend it for simultaneously computing square root and performing division for layer normalization. The softmax is further optimized by means of exponent truncation and subtraction reuse. For layer normalization, the square of the mean is additionally eliminated from the denominator. Based on the proposed algorithms, we develop two architectures for softmax and layer normalization, respectively. Our experimental results demonstrate that our proposed architectures can not only achieve state-of-the-art model performance but also offer superior area cost and power consumption, among all the existing related works. They can work as plug-and-play units since the resulting performance loss is negligible, without any requirement for fine-tuning.

Acknowledgements This work was supported by National Natural Science Foundation of China (Grant No. 62074097).

References

1 Brown T, Mann B, Ryder N, et al. Language models are few-shot learners. In: Proceedings of Advances in Neural Information Processing Systems, 2020. 1877–1901
2 OpenAI. GPT-4 technical report. 2023
3 Zhang S, Roller S, Goyal N, et al. OPT: Open pre-trained transformer language models. arXiv: 2205.01068, 2022
4 Touvron H, Lavril T, Izacard G, et al. LLaMA: Open and efficient foundation language models. arXiv: 2302.13971, 2023
5 Touvron H, Martin L, Stone K, et al. LLaMA 2: Open foundation and fine-tuned chat models. arXiv: 2307.09288, 2023
6 Zhao W X, Zhou K, Li J, et al. A survey of large language models. arXiv: 2303.18223, 2023
7 Chang Y, Wang X, Wang J, et al. A survey on evaluation of large language models. arXiv: 2307.03109, 2023
8 Yang J, Jin H, Tang R, et al. Harnessing the power of llms in practice: A survey on ChatGPT and beyond. arXiv: 2304.13712, 2023
9 Bubeck S, Chandrasekaran V, Eldan R, et al. Sparks of artificial general intelligence: Early experiments with GPT-4. arXiv: 2303.12712, 2023
10 Sheng Y, Zheng L, Yuan B, et al. FlexGen: High-throughput generative inference of large language models with a single GPU. In: Proceedings of the 40th International Conference on Machine Learning, 2023. 31094–31116
11 Lu L, Jin Y, Bi H, et al. Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture. In: Proceedings of 54th Annual IEEE/ACM International Symposium on Microarchitecture, 2021. 977–991
12 Yang T, Ma F, Li X, et al. DTATrans: Leveraging dynamic token-based quantization with accuracy compensation mechanism for efficient Transformer architecture. IEEE Trans Comput-Aided Design Integr Circuits Syst, 2023, 42: 509–520
13 Zhou Z, Liu J, Gu Z, et al. Energon: Toward efficient acceleration of Transformers using dynamic sparse attention. IEEE Trans Comput-Aided Design Integr Circuits Syst, 2023, 42: 509–520
14 Ham T J, Jung J S, Kim S, et al. A^3: Accelerating attention mechanisms in neural networks with approximation. In: Proceedings 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2020. 328–341
15 Wang H, Zhang Z, Han S. SpAtten: Efficient sparse attention architecture with cascade token and head pruning. In: Proceedings of 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), 2021. 97–110
16 Dass J, Wu S, Shi H, et al. ViTALiTy: Unifying low-rank and sparse approximation for vision Transformer acceleration with a linear Taylor attention. In: Proceedings of 2023 IEEE International Symposium on High-Performance Computer Architecture

(HPCA), 2023. 415–428
17 Guo C, Tang J, Hu W, et al. OliVe: Accelerating large language models via hardware-friendly outlier-victim pair quantization. In: Proceedings of the 50th Annual International Symposium on Computer Architecture, 2023. 1–15
18 Mirzadeh I, Alizadeh K, Mehta S, et al. ReLU strikes back: Exploiting activation sparsity in large language models. arXiv: 2310.04564, 2023
19 Peng H, Huang S, Chen S, et al. A length adaptive algorithm-hardware co-design of Transformer on FPGA through sparse attention and dynamic pipelining. In: Proceedings of 2022 59th ACM/IEEE Design Automation Conference (DAC), 2022. 1135–1140
20 Stevens R J, Venkatesan R, Dai S, et al. Softermax: Hardware/software co-design of an efficient softmax for Transformers. In: Proceedings of 2021 58th ACM/IEEE Design Automation Conference (DAC), 2021. 469–474
21 Tuli S, Jha N K. AccelTran: A sparsity-aware accelerator for dynamic inference with Transformers. IEEE Trans Comput-Aided Design Integr Circuits Syst, 2023, 42: 4038–4051
22 Wang J, Zhang L, Li X, et al. ULSeq-TA: Ultra-long sequence attention fusion Transformer accelerator supporting grouped sparse softmax and dual-path sparse LayerNorm. IEEE Trans Comput-Aided Design Integr Circuits Syst, 2024, 43: 892–905
23 Sze V, Chen Y H, Yang T J, et al. Efficient processing of deep neural networks: A tutorial and survey. Proc IEEE, 2023, 105: 2295–2329
24 Wang M, Lu S, Zhu D, et al. A high-speed and low-complexity architecture for softmax function in deep learning. In: Proceedings of 2018 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS), 2018. 223–226
25 Zhu D, Lu S, Wang M, et al. Efficient precision-adjustable architecture for softmax function in deep learning. IEEE Trans Circuits Syst II: Exp Briefs, 2020, 67: 3382–3386
26 Spagnolo F, Perri S, Corsonello P. Aggressive approximation of the softmax function for power-efficient hardware implementations. IEEE Trans Circuits Syst II: Exp Briefs, 2022, 69: 1652–1656
27 Mei Z, Dong H, Wang Y, et al. TEA-S: A tiny and efficient architecture for PLAC-based softmax in Transformers. IEEE Trans Circuits Syst II: Exp Briefs, 2023, 70: 3594–3598
28 Dong H, Wang M, Luo Y, et al. PLAC: Piecewise linear approximation computation for all nonlinear unary functions. IEEE Trans VLSI Syst, 2020, 28: 2014–2027
29 Koca N A, Do A T, Chang C H. Hardware-efficient softmax approximation for self-attention networks. In: Proceedings of 2023 IEEE International Symposium on Circuits and Systems (ISCAS), 2023. 1–5
30 Wang W, Zhou S, Sun W, et al. SOLE: Hardware-software co-design of softmax and layerNorm for efficient Transformer inference. In: Proceedings of 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD), 2023. 1–9
31 Lu S, Wang M, Liang S, et al. Hardware accelerator for multi-head attention and position-wise feed-forward in the Transformer. In: Proceedings of 2020 IEEE 33rd International System-on-Chip Conference (SOCC), 2020. 84–89
32 Yu J, Park J, Park S, et al. NN-LUT: Neural approximation of non-linear operations for efficient Transformer inference. In: Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC), 2022. 577–582
33 Shen S, Dong Z, Ye J, et al. Q-BERT: Hessian based ultra low precision quantization of BERT. In: Proceedings of the AAAI Conference on Artificial Intelligence, 2020. 8815–8821
34 Frantar E, Ashkboos S, Hoefler T, et al. GPTQ: Accurate post-training quantization for generative pre-trained Transformers. arXiv: 2210.17323, 2022
35 Lee C, Jin J, Kim T, et al. OWQ: Lessons learned from activation outliers for weight quantization in large language models. arXiv: 2306.02272, 2023
36 Lin J, Tang J, Tang H, et al. AWQ: Activation-aware weight quantization for LLM compression and acceleration. arXiv: 2306.00978, 2023
37 Dettmers T, Svirschevski R, Egiazarian V, et al. SpQR: A sparse-quantized representation for nearlossless LLM weight compression. arXiv: 2306.03078, 2023
38 Kim S, Hooper C, Gholami A, et al. SqueezeLLM: Dense-and-sparse quantization. arXiv: 2306.07629, 2023
39 Xiao G, Lin J, Seznec M, et al. LLM.int8 (): 8-bit matrix multiplication for Transformers at scale. In: Proceedings of the 40th International Conference on Machine Learning, 2023. 38087–38099
40 Wei X, Zhang Y, Li Y, et al. Outlier Suppression+: Accurate quantization of large language models by equivalent and optimal shifting and scaling. arXiv: 2304.09145, 2023
41 Yuan Z, Niu L, Liu J, et al. RPTQ: Reorder-based post-training quantization for large language models. arXiv: 2304.01089, 2023
42 Devlin J, Chang M W, Lee K, et al. BERT: Pre-training of deep bidirectional Transformers for language understanding. arXiv: 1810.04805, 2018
43 Wang A, Singh A, Michael J, et al. GLUE: A multi-task benchmark and analysis platform for natural language understanding. arXiv: 1804.07461, 2018
44 Merity S, Xiong C, Bradbury J, et al. Pointer sentinel mixture models. arXiv: 1609.07843, 2016.
45 Marcus M, Kim G, Marcinkiewicz M A, et al. The Penn treebank: Annotating predicate argument structure. In: Proceedings of Human Language Technology: Proceedings of a Workshop, Plainsboro, 1994. 8–11
46 Raffel C, Shazeer N, Roberts A, et al. Exploring the limits of transfer learning with a unified text-to-text Transformer. J Mach Learn Res, 2020. 21: 1–67
47 Dong P, Tan Y, Zhang D, et al. Genetic quantization-aware approximation for non-linear operations in Transformers. arXiv: 2403.19591, 2024
48 Yang J, Tu F, Li Y, et al. GQNA: Generic quantized DNN accelerator with weight-repetition-aware activation aggregating. IEEE Trans Circuits and Syst I: Reg Papers, 2022, 69: 4069–4082
49 Li W, Hu A, Xu N, et al. A precision-scalable deep neural network accelerator with activation sparsity exploitation. IEEE Trans Comput-Aided Design Integr Circuits Syst, 2024, 43: 263–276