uLog: a software-based approximate logarithmic number system for computations on SIMD processors

Saeedeh Jahanshahi1 · Amir Sabbagh Molahosseini2 · Azadeh Alsadat Emrani Zarandi3

Accepted: 7 July 2022 © The Author(s) 2022

Abstract
This paper presents a new number representation based on logarithmic number system (LNS) called unsigned logarithmic number system (uLog), as an alternative to the conventional floating-point (FP) number format, to use in approximate computing applications. uLog is tailored for software implementation on commercial general-purpose processors, and uses the same dynamic range as conventional IEEE Standard FP formats to prevent overflow and underflow. uLog converts FP numbers to fixed-point numbers and uses integer operations for all computations. Moreover, vectorization and approximate logarithmic addition have been used to increase the performance of the software implementation of uLog. Then, we used different BLAS benchmarks to evaluate the performance of the proposed format than IEEE standard formats. 16- and 32-bit uLog improve the runtime than double-precision at most 70.26% and 46.36%, respectively. Besides, accuracy analysis of the uLog based on different logarithm bases showed that base 4 has the lowest error in most cases.

Keywords Logarithmic number system (LNS) · Approximate computing · Floating point (FP) format · Single instruction multiple data (SIMD)

1 Introduction
Emerging computing paradigms such as Internet-of-Things (IoT) and edge intelligence require fast and energy-efficient computing systems suitable for real-time processing [3, 10]. The main sources of energy-consumption in computing systems...
are numerical computations and data movement between memory hierarchies [10].
In this regard, approximate arithmetic units have been widely investigated to provide efficient numerical computations [13]. However, the energy consumed by data movement can be higher than the energy used by calculation in some situations. For example, a typical 64-bit floating-point (FP) multiply–add operation costs around 200 pJ, while reading 64-bits from cache and main memory cost 800 pJ and 12,000 pJ, respectively [18]. Therefore, reducing data movement can decrease last-level cache misses and improve power-consumption and delay, especially in memory-intensive applications [3]. There are a wide range of applications such as multimedia/signal processing, data mining, and machine learning that do not need high-precision and accurate results [11]. Therefore, decreasing the bit-length of FP numbers by customizing the accuracy is one of the efficient ways to reduce data movement in memory [3]. However, most commercial computing systems including off-the-shelf processors support only IEEE 754 standard FP formats, i.e., double-precision (DP) with 64-bits length and single-precision (SP) with 32-bits length [40]. Therefore, FP numbers cannot use less than 32-bits in such systems, which restricts the reduction of data movement between memories and registers in applications where low-precision can be used such as machine learning (ML) algorithms [1, 28, 33, 36–38]. The SP is currently the dominant data type for deep learning training systems, and even recent studies have shown that low-precision FP formats such as 8- and 16-bits data types can be used [33, 36–38]. However, some high-performance computing applications such as graph processing systems still require high-precision at least in some parts of their computations [14–16]. One approach to reduce the runtime in such applications is to convert DP to SP. However, sometimes the conversion of DP to SP is not possible because SP has a smaller dynamic range than DP, leading to overflow or underflow. An alternative approach to reduce the energy consumption is to design new number systems and memory formats for FP numbers [3, 10, 11, 15, 16, 28, 34]. New memory formats try to change the layout of storing data in memory and differ from new number system. However most of current works in approximate computations have focused on new memory formats.

New memory formats can be designed by producing new hardware [27] or can be based on software developments [3, 14–16, 34]. The hardware-based memory formats (memory formats with dedicated hardware support) have high efficiency, while the software-based ones can run on existing hardware architectures without any change [30]. The main target of software-based memory formats is to reduce data movement, resulting in overall system speedup. Although, this type of number format incurs some overhead for software emulation of operations, conversion of the customized FP type to native SP or DP types before any computation can be used to increase performance [3, 14–16]. The separation of data in memory from computation in memory formats [16] helps to use high-performance native formats for operations and decreases memory transfer volume by customizable FP formats. In this regard, Anderson et al. [3] proposed a compiler-based scheme called “flyte” to customize the precision of FP numbers. The width of flyte types was chosen to be multiples of 8-bits (such as 24, 40, and 48 bits). Its exponent size is also equal to the next largest IEEE 754 standard to avoid overflow and underflow. flyte is a memory format therefore after loading from memory, it converted to the next largest IEEE 754, and
all operations were done in native hardware supported standard type. The cost of converting before and after computations, and the misalignment of data length with memory boundary are two problems of flyte. Therefore, they tried to overcome these challenges by using packing and unpacking of data before and after storing and loading, and vectorization. They used different rounding methods to control the error when converting an IEEE 754 data to flyte. The experimental evaluations showed that flyte had low overhead and could be faster than native types in some cases [3]. Moreover, Grützmacher and Anzt [14] introduced a customized precision format based on mantissa segmentation (CPMS). They split the standard DP format into several equally sized segments and changed the layout of DP in memory. The size of each segments was chosen based on hardware-specific data access for high-performance access and conversion, unlike flyte. The segments were rearranged based on performance and the platform-dependent length of data in memory read. CPMS can dynamically be adopted to the accuracy needing of applications, and when the application is error-tolerance, the subset of segments were read from memory, and the remaining bits in mantissa were replaced with zero to convert CPMS formats into DP. CPMS is a memory format and needs conversion before and after any computations, similar to flyte, since it depended to standard IEEE 754 FP types in operations, although it just used DP as the baseline type in analysis. Besides, CPMS was implemented on GPUs and CPUs with vectorization and tested on sparse linear algebra and page rank [15, 16].

There are many works in the areas of signal processing [23], video processing [4], and neural networks [28] that have studied logarithmic number system (LNS) as an alternative FP number system for embedded applications [1]. LNS converts FP numbers to fixed-point numbers, which can be represented as an integer in off-the-shelf CPUs. This feature helps to use simple, low energy, and low cost integer arithmetic units of CPUs to emulate LNS operations. Fewer bits are needed to show an extensive dynamic range in LNS numbers compared to standard fixed-point numbers [12]. LNS systems do not have time-consuming and slow division and reverse conversion to binary, unlike Residue Number System (RNS) [32]. RNS is also another unconventional number system which can provide parallel arithmetic operations over integers. It converts integers by pairwise coprime number called moduli, and has simple addition, subtraction and multiplication but complex division and reverse conversion compared to LNS [6]. In LNS, multiplication, exponentiation, and square root operations are reduced to addition, subtraction, multiplication, and shift [1]. Therefore, emulation of these LNS operations in software has a low cost. The properties of LNS have helped many applications to reduce power consumption and improve performance. Arnold [4] applied LNS on MPEG decoding to reduce power consumption and showed that the bit length of 8 to 10 bits in logarithmic format did not affect the output quality compared with fixed-point with the bit length of 14 to 16 bits. Base-tas et al. [5] used LNS for signal processing and concluded that LNS requires lower word length than linear representation and can save the power more than 60 percent in some cases. Moreover, Alam and Gustafsson [2] tried to optimize the word length of coefficients in FIR filters by replacing it with the logarithmic coefficients which resulted in LNS filters with lower errors than standard FIR filters. Furthermore, Gao et al. proposed a run-time estimation algorithm based on a data flow graph (DFG)
to identify the error-tolerance computations [11]. They used logarithmic representation for the input of DFG and achieved 32× more accurate results than static DFG with lower energy consumption. They also showed that their algorithm lead to high performance in k-means and perceptron machine learning algorithms. Besides, [7, 8] compared the performance and accuracy of the European logarithmic microprocessor (ELM) with commercial FP processors. Their experiments showed that LNS could be a suitable replacement for SP in some cases. Moreover, Parhami compared different techniques and architectures for applying LNS operations efficiently by focusing on ELM as a case study, and showed that LNS has a high potential for deployment in general-purpose systems [32]. The non-uniform distribution of weights and activations in neural networks (NN) is the same as the distribution of numbers in LNS [1]. Therefore, in recent years, LNS has been explored for NN such as the work of Miyashita et al. [28] which encoded the weights and activations of a trained network to 3 bits with base 2 logarithm representation and attained higher classification accuracy than fixed-point. They also used 5 bits logarithm encoding in the training process. Moreover, LOGNET [24] achieved more accuracy with 4 or fewer bits logarithmic resolution than linear encoding in convolutional neural networks (CNN) and matrix multiplication. Furthermore, some works analyzed the effect of choosing bases other than 2 on error and performance. For example, Vogel et al. [36] showed that the accuracy of using power-of-arbitrary-log bases for quantization of pre-trained NN is completely enough, and there is no need to retrain NN. [28] also concluded that the total quantization error on NN weights of base $\sqrt{2}$ is 2X smaller than base 2. Recently, Alam et al. [1] investigated the impact of choosing different logarithmic bases, and they found that a suitable base could change the distribution and re-accomplished due the average error. Indeed, careful adjustment of the LNS base can provide a suitable tradeoff between precision and dynamic range [29].

In this work, we propose a new software-based logarithmic number system representation for FP numbers and call it unsigned logarithmic number system (ulog) to decrease the data size and memory traffic. As mentioned before, flyte [3] and CPMS [14–16] tried to separate memory operation from computations because hardware does not support the computations on the customized formats and also emulation of FP operations has low performance. Thus, they used conversion between each memory read/write and computation. But ulog only needs one conversion from FP to ulog that could be done offline. Because integer type is the baseline format for ulog, and all ulog operations could be emulated with integer arithmetic units of commercial CPUs that is faster than FP unit. The proposed ulog number system uses the same exponent bits as IEEE 754 FP types to avoid overflow and underflow, similar to CPMS and flyte. However, unlike flyte, ulog length is chosen based on data access alignment in hardware to eliminate the overhead of packing and unpacking. Besides, in contrast to previous works such as [1, 11, 24, 28], which have designed hardware for LNS for specific applications, the proposed work introduces the entire software implementation of the logarithmic-based number system for using on commercial general-purpose processors. We vectorize the required software-based LNS functions to use the benefits of single instruction multiple data (SIMD) architecture and reduce the overhead of software emulation of ulog operations. Since addition and subtraction are complicated in LNS, previous works commonly used lookup tables.
to reduce the complexity of these operations [1]. The size of lookup tables increases significantly with bit-width, therefore previous studies used LNS with small bit length and low precision to speed up execution [32, 40]. However, approximation techniques can be used to reduce latency for long word length logarithmic addition [29]. Therefore, we use the equation proposed by Gao and Qu [11] for logarithmic addition to increase the accuracy of ulog. This equation approximates the addition and eliminates the need for lookup tables. However, the approximate LNS addition equation proposed by Gao and Qu [11] does not support vectorized reduction by addition. Therefore, we extended this equation and introduced a new formula for vectorization of the logarithmic reduction by addition, which has a lower cost than the method proposed by Gao and Qu [11]. Furthermore, most previous works were limited to logarithm base 2. However, the proposed method is not restricted to logarithm base 2 due to the proposed general equations for logarithmic addition and reduction by addition.

In the rest of paper, Sect. 2 briefly reviews the IEEE 754 standard for representing FP numbers. Then, Sect. 3 introduces the proposed ulog scheme. Moreover, the experimental evaluation of run time and error of ulog is demonstrated in Sects. 4, and 5 concludes the paper.

2 The IEEE 754 Standard for floating-point numbers

IEEE 754 standard defines a FP number system for representing real numbers, which have been widely used in computing systems. Each IEEE 754 FP number consists of (1) exponent, which is an integer that determines the dynamic range, (2) mantissa, which is a positive decimal number between 0 and 1, and (3) one bit for sign of mantissa, as shown in Fig. 1 [40]. In other words, exponent shows how big or small a FP number could be, and mantissa shows the accuracy of a FP number. Thus increasing/decreasing exponent bits enhance/reduce the range of representable numbers, and increasing/decreasing mantissa bits enhance/reduce the precision of FP numbers. Furthermore, all normalized FP numbers have a hidden number one that never saves in memory but adds to mantissa in computations. A constant integer called bias add to the exponent to convert it to a positive integer before storing it in memory. Length of exponent and mantissa and the value of bias depend on the type of FP in IEEE 754. IEEE 754 defines different types for FP numbers, but two of them are the most common types that all CPUs support: single- and double-precision floating-point number formats. SP and DP occupy 4 and 8 bytes in memory, respectively. Table 1 shows the bit length of exponent and mantissa, the value of bias, and the maximum representable FP number for each type. Besides, one bit has
been reserved for the sign of mantissa in Table 1. Although, IEEE half-precision FP format is available in some GPUs, there is no choice of using lower precision FP data with smaller width than 32 bits in conventional CPUs. Therefore, inflexibility is the biggest problem of the IEEE standard, especially for applications where approximate computations can be used. Another problem is the probability of overflow and underflow when DP converts to SP.

3 The proposed work

Vectorization facilitates improving the efficiency of the software-based implementation of all ulog operations and conversions. This work utilizes the vectorization technique to decrease the cost of implementation of ulog operations. Vectorization uses the “single instruction, multiple data (SIMD)” technique that concurrently runs one instruction on multiple data. Intel has different architectures with different registers width for vectorization: AVX512 with 512 bits registers [21], AVX/AVX2 with 256 bits registers, and SSE with 128 bits registers [26]. Furthermore, Intel has introduced intrinsic functions for different architectures to simplify vectorization on its CPUs [22]. In this work, we used AVX512 to implement ulog\(^1\) scheme, therefore, all pseudocodes also use AVX512 intrinsic functions.

This section illustrates different parts of the proposed number system in the following subsections. We first introduce the ulog scheme in Sects. 3.1, and then 3.2 describes the implementation of square root and multiplication. Moreover, Sect. 3.3 presents the required formula for addition and reduction by the addition, which are the most complex and time-consuming operations in LNS. Besides, Sect. 3.4 is introduced general equations for addition and reduction by addition on different logarithm bases. At the end of Sect. 3, the problem of unsupported AVX512BW CPUs is solved to enable 16-bits long ulog types running on most modern CPUs.

3.1 The “ulog” scheme

According to [11], LNS converts a FP number to a fixed-point base-2 number without the possibility of overflow and underflow. We use this capability to build a new

---

\(^1\) The AVX512 codes of ulog is available at this link: https://github.com/saeedeh-jahanshahi/ulog.git.
uLog: a software-based approximate logarithmic number system…

customizable and flexible number system for FP numbers. All previous works [1, 11, 28] try to design new hardware and using LNS for particular purposes. However, the target of this paper is to develop the first software-based logarithmic number system implementation for general-purpose computations. Therefore additional cost does not need for designing new hardware, and ulog can be used for different approximate applications running on conventional commercial processors. ulog utilizes unsigned integer operations to efficiently emulate the proposed LNS-based functions and conversions in the software. Moreover, software-based implementation of all ulog operations reduces the number of conversions to only one that could be done offline, causing a significant reduction in the conversion cost. However, the other software-based number systems like flyte [3] and CPMS [14–16] require conversion to and from native FP types at the start and end of every function. Moreover, unlike the conversion of DP to SP, overflow and underflow do not occur in the conversion of DP/SP to ulog because ulog keeps the exponent bits of DP/SP unchanged.

Division, multiplication, root, and exponential operations are costly on the hardware. LNS converts these operations into low cost hardware-supported operations. This feature optimizes the software-based emulated calculations in ulog domain. For instance, some useful features of LNS for two arbitrary positive decimal numbers A and B are as follows [11]:

\[
\log_2(A \times B) = \log_2 A + \log_2 B \tag{1}
\]

\[
\log_2(A \div B) = \log_2 A - \log_2 B \tag{2}
\]

\[
\log_2(A^n) = n\log_2 A \tag{3}
\]

\[
\log_2\left(\sqrt[n]{A}\right) = 1/n \log_2 A \tag{4}
\]

\[
\log_2 A < \log_2 B \text{ if } A < B \tag{5}
\]

The logarithmic approximation is defined as follows:

\[
\log_2(1 + x) \approx x \text{ for } 0 \leq x < 1 \tag{6}
\]

The base of the logarithm of any number can be changed as follows:

\[
\log_b A = \frac{\log_c A}{\log_c b} \tag{7}
\]

According to [32], logarithmic representation of signed numbers consists of separating the sign bit from the number, and then computing the logarithm of the absolute value of the number and taking the 2’s complement of the result as follows:
In this work, we ignore the sign of all numbers in LNS as a particular case. There are many applications that unsigned LNS can be used such as some graph processing algorithms. For example PageRank values are between zero and one [29], and messages in belief propagation are also normalized positive numbers [35].

Based on the logarithmic features, a FP number can be converted to LNS as follows [11]:

\[
A = 2^e \times (1+m) \Rightarrow \log_2 A = \log_2 (2^e \times (1+m)) = e + \log_2 (1+m)
\]  

Since \(0 \leq m < 1\), Eq. (9) can be rewritten using Eq. (6) as follows:

\[
e + \log_2 (1+m) \approx e + m = e.m
\]

where “.” denotes decimal point, \(e\) is an integer greater than one, and \(m\) is a decimal between zero and one. The result of addition is a decimal number in which the integer part is \(e\), and the fraction part is \(m\). Here, \(e\) and \(m\) represent the exponent and the mantissa of the unsigned decimal number \(A\). In the ulog number system, the respective exponent and mantissa are the integer and the fraction parts of a fixed-point number. Equation (10) shows the cost of converting to ulog is minimal, and is lower than the software-based types like flyte [3] and CPMS [14–16], since all FP numbers need to convert only one time.

This paper introduces three different ulog types with width of 16- and 32-bits for various purposes. Choosing 16- and 32-bit lengths reduces the memory access overhead and simultaneously improves the performance of data access by aligning with hardware-based characteristics of data movement. Table 1 demonstrates different ulog types and their comparison with the DP and the SP types.

One of the major problems of the logarithmic representation is how to deal with the sign of the numbers since the logarithm of negative numbers is undefined. Although signed FP numbers can be represented in LNS in a way [32], it increases the cost of software implementation, especially in subtraction and reduction operations. Therefore the proposed ulog scheme is a number system for unsigned FP numbers, and we ignore the sign of FP numbers. The types ulogd32 and ulogd16 are unsigned logarithmic formats with the same exponent width as DP numbers. Moreover, the type ulogs16 has the same exponent width as the SP numbers. As presented in Table 1, the biggest number represented in the SP type is significantly smaller than that in the DP type. However, using the same dynamic range as that of the DP in ulog avoids this problem in ulog types.

As shown in Fig. 2a–c, conversion from DP/SP to ulog only changes the mantissa bits while the exponent stays unchanged. By contrast, the hardware-based conversion of an IEEE 754 DP number to an IEEE 754 SP has a high cost because both the dynamic range and the mantissa bits must be converted (see Fig. 2d). To be more precise, in the conversion of DP to SP, the dynamic range of the SP needs to be recomputed with a new width and a new bias from the DP dynamic range while the mantissa of DP is
uLog: a software-based approximate logarithmic number system…

**Fig. 2** Conversion from DP/SP to ulog: a DP to ulogd32, b DP to ulogd16, c SP to ulogs16 and d DP to SP
rounded to build an SP mantissa. However, conversion from FP to ulog only requires truncation of the mantissa bits while the dynamic range of the FP number remains unchanged.

### 3.2 Multiplication and square root operations using ulog

One of the most common operations in many applications, particularly linear algebra, is multiplication. LNS converts the multiplication to addition, as shown in Eq. (1). Although the multiplication result is approximate because original numbers convert to ulog approximately, the bias of the particular ulog type has been added to the exponents of both input numbers in multiplication. When two exponents added, the final result includes extra bias that should be subtracted. As shown in Table 1, each ulog type has its own bias value. Therefore input vectors should be tested for zero numbers before any multiplication because the logarithm of zero is undefined. The following relation shows the details of multiplication in ulog.

\[
\begin{align*}
A & \approx 2^{e_A + \text{bias}} \times (1 + m_A) \\
B & \approx 2^{e_B + \text{bias}} \times (1 + m_B)
\end{align*}
\]

\[
\log_2 (A \times B) = \log_2 A + \log_2 B = \log_2 (2^{e_A + \text{bias}} \times (1 + m_A)) + \log_2 (2^{e_B + \text{bias}} \times (1 + m_B))
\]

\[
= \log_2 2^{e_A + \text{bias}} + \log_2 (1 + m_A) + \log_2 2^{e_B + \text{bias}} + \log_2 (1 + m_B)
\]

\[
\Rightarrow \approx e_A + \text{bias} + m_A + e_B + \text{bias} + m_B = e_A m_A + e_B m_B + 2 \text{bias}
\]

\[
\Rightarrow = e_A m_A + e_B m_B + 2 \text{bias} - \text{bias} = e_A m_A + e_B m_B + \text{bias}
\]

Moreover, Eq. (12) indicates that the square root, which is the division of ulog number by two could be done with only one shift to the right (\(\gg\) notation in Eq. (12) denotes shift to the right operation). As it can be seen in, after a right shift of the input number, the value of bias becomes incorrect. Therefore, bias should be removed before the shift and, at the end, should be added to the final result.

\[
A \approx 2^{e_A + \text{bias}} \times (1 + m_A) \Rightarrow \log_2 \left(\sqrt[2]{A}\right) = \log_2 A^{1/2} = \frac{1}{2} \log_2 A = \frac{1}{2} \log_2 (2^{e_A + \text{bias}} \times (1 + m_A)) =
\]

\[
\frac{1}{2} \left(\log_2 2^{e_A + \text{bias}} + \log_2 (1 + m_A)\right)
\]

\[
\Rightarrow \approx \frac{1}{2} (e_A + \text{bias} + m_A) = (e_A m_A \gg 1) + \frac{1}{2} \text{bias}
\]

### 3.3 Addition and reduction by add using ulog

Adding two numbers is difficult in LNS. Most of the previous works utilize a lookup table for addition. However, [11] proposed a new approximate method without requiring a lookup table. Assuming two arbitrary positive decimal numbers \(A\) and \(B\) such that \(A \geq B > 0\), the summation of these two numbers is as follows [11]:

\[
A + B \approx 2^{A_1} + 2^{B_1} = 2^{A_1} (1 + 2^{B_1 - A_1})
\]

where \(A_1 = \log_2 A, B_1 = \log_2 B\). Besides, \(0 < 2^{B_1 - A_1} \leq 1\) holds true since \(B_1 \leq A_1\) based on Eq. (5). Therefore, the logarithm of the summation can be approximated using Eq. (6) is as follows:

\[
\text{Springer}
\]
uLog: a software-based approximate logarithmic number system...

Which results in

$$S_i = \log_2(A + B) \approx A_i + (1 \gg \lfloor A_i - B_i \rfloor)$$

(15)

where \( \lfloor \cdot \rfloor \) denotes rounding to the nearest integer. Figure 3a, b show the pseudocode of rounding to the nearest integer in \( uLogd32 \) and \( uLogd16 \), respectively.

Unlike the normal addition where only one single hardware instruction is required, the logarithmic addition needs more instructions. The situation is even worse for the logarithmic reduction by addition. For example, for an array of \( n \) numbers, the time complexity of the logarithmic reduction is \( k(n - 1) \) where \( k \) is the number of instructions required for logarithmic addition of two numbers. On the other hand, the logarithmic reduction can be vectorized, which helps to decrease the overhead significantly. However, vectorization of Eq. (15) is difficult because it needs a top-down approach where every two numbers should be added at each level. Thus a new method is proposed here for vectorization of the logarithmic reduction in \( uLog \) number system. This new method reduces the time complexity of Eq. (15) by the factor of \( S_{num} \) where \( S_{num} \) is the number of data with specific width that can be stored on CPU registers.

Consider an array of \( n \) arbitrary positive decimal numbers \( A_i \) with \( i = 1, \ldots, n \). The logarithm of the array’s elements summation showed in Eq. (16) where \( a_i = \log_2 A_i \) and \( a_{\text{max}} = \max(a_1, a_2, \ldots, a_n) \), and the function \( \text{max} \) yields the largest value of the \( a_i \)'s. It can be seen that similar to Eq. (14), each \( A_i \) is substituted with \( 2^{a_i} \) and then \( 2^{a_{\text{max}}} \) factorized from each part of Eq. (16). The symbol \( \lfloor \cdot \rfloor \) in all the following formulas denotes rounding to the nearest integer.

$$\log_2(\sum_{i=1}^{n} A_i) \approx \log_2(2^{a_{\text{max}}} (1 + 2^{-[a_{\text{max}} - a_1]} + 2^{-[a_{\text{max}} - a_2]} + \cdots + 2^{-[a_{\text{max}} - a_n]}))$$

$$= \log_2 2^{a_{\text{max}}} + \log_2 (1 + 2^{-[a_{\text{max}} - a_1]} + 2^{-[a_{\text{max}} - a_2]} + \cdots + 2^{-[a_{\text{max}} - a_n]})$$

(16)
Moreover, since \(0 < 2[a_i - a_{\text{max}}] \leq 1\), we have:

\[
1 < \sum_{i=1}^{n} 2[a_i - a_{\text{max}}] \leq n \tag{17}
\]

The result of array additions in Eq. (17) can be considered as a decimal number with the integer part \(r\) and the fraction part \(f\) as follows:

\[
\sum_{i=1}^{n} 2[a_i - a_{\text{max}}] \approx r + f = rf \tag{18}
\]

where \(r \in \mathbb{N}, 1 \leq r \leq n, and 0 \leq f < 1\). Therefore, Eq. (16) can be rewritten as:

\[
\log_2\left(\sum_{i=1}^{n} a_i\right) \approx a_{\text{max}} + \log_2(r+f) = a_{\text{max}} + \log_2\left(r\left(1 + \frac{f}{r}\right)\right) = \log_2 r + \log_2(1 + \frac{f}{r}) \tag{19}
\]

The \(0 \leq \frac{f}{r} < 1\) is true since \(r\) is an integer number between 1 and \(n\). Therefore, the term \(\log_2\left(1 + \frac{f}{r}\right)\) can be approximated via Eq. (6) which results in:

\[
\log_2\left(\sum_{i=1}^{n} A_i\right) \approx a_{\text{max}} + \log_2(r) + \frac{f}{r} \tag{20}
\]

Since the numerical computation of the term \(\frac{f}{r}\) is not straightforward, we overcome this difficulty by using the approximate value of \(r\). The approximate value is obtained via finding a specific integer number \(k\) where the values of two raised to the power of \(k\) and \(k+1\) restricts the term \(r\). In other words:

\[
2^k \leq r < 2^{k+1}, \text{where } k \geq 0 \tag{21}
\]

Which results in the approximation \(r \approx 2^k\). Therefore, Eq. (19) can be approximated as follows:

\[
\log_2\left(\sum_{i=1}^{n} A_i\right) \approx a_{\text{max}} + k + \frac{f}{2^k} = a_{\text{max}} + k + (f \gg k) \tag{22}
\]

After calculating \(rf\) in Eq. (18), the final result can be computed with only one right shift and two additions. More instructions are required to calculate the value of Eq. (18). However, this computation’s cost can be reduced due to vectorization capability. It should be noted that the pseudocode of \(\text{ulogs16}\) and \(\text{ulogd16}\) has a slight difference from \(\text{ulogd32}\), because Intel does not have intrinsic functions for reduction by maximum and reduction by addition on 16 bits integers. Therefore, the reduction by maximum and by addition at first apply to the first half of each 32 bits parts, and then both reductions apply to the second half after setting the first half to zero.
As shown in Fig. 4, in order to implement the logarithmic reduction (by addition), we use Eqs. (15) and (22). Thus, according to this figure, we need $k'(n/S_{num})$ instructions for reducing $n$ numbers in ulog, where $k'$ is the total number of instructions in Eqs. (15) and (22).

### 3.4 ulog on other bases

According to [1], ulog can use bases other than two to increase precision or dynamic range and then decrease error. Although to simplify addition and reduction by addition operations the bases should have the form $2^t$ where $t \in \mathbb{Z}$, the bases move the decimal point to left or right. Therefore the complexity of operations does not change, and only mantissa and exponent bit length and the bias are changed. The decimal point moves to the right when using a base lower than two. Then, the exponent bit increases, and the mantissa bit decreases. Therefore, the resulting number could represent a bigger number than the number in base 2 but with lower precision. On the other hand, when using a base bigger than two, the reverse happens, and the resulting number can represent a smaller number than the base two numbers but with more precision.

$$\log_b A = \log_b (1 + m)2^e = \frac{\log_2 2^e + \log_2 (1 + m)}{\log_2 b} \approx \frac{e + m}{\log_2 2^t} = 2^{-t}(e + m)$$  \hspace{1cm} (23)

For example, suppose $b = 2^{2^{-1}} = \sqrt{2}$ then $\log_{\sqrt{2}} A \approx 2(e + m)$. It means that A’s logarithm shifts one bit to the left in this base, as shown in Fig. 5. Therefore, the accuracy of the $\log_{\sqrt{2}} A$ decreases one bit while the dynamic range increases. On the other hand, if $b = 2^{2^1} = 4$ then $\log_4 A \approx \frac{e + m}{2}$. Therefore, according to Fig. 5, the dynamic range decreases one bit but the precision increases.

All operations except addition and reduction by addition do not change. Therefore, the general equation for both of these operations with any base of $2^t$ can be achieved as follows.

Assume $A$ and $B$ as two positive numbers such that $A \geq C$ and $a_{ib} = \log_{ib} A$ and $c_{ib} = \log_{ib} C$. Then, we have:
As shown in Eq. (25), we can use Eq. (6) by changing the base with Eq. (7):

\[ b = 2 \]

Now, \( b \) can be replaced with \( 2^t \):

\[ a_{2t} = a_{2t} + 2^{-2^t(a_{2t} - c_{2t})} \]

Then, \( 2^t(a_{2t} - c_{2t}) \) can be rounded to the nearest integer to simplify computation:

\[ a_{2t} + 2^{-2^t(a_{2t} - c_{2t})} \]

The logarithmic reduction by addition in base \( b \) is similar to addition. Assume \( a_{i,lb} = \log_b A_i, a_{\text{max,lb}} = \max(a_{1,lb}, \ldots, a_{n,lb}) \) for \( i = 1, \ldots, n \). Then, we have:

\[
\log_b \left( \sum_{i=1}^{n} A_i \right) \approx \log_b (b^{a_{1,lb}} + \cdots + b^{a_{n,lb}}) = \log_b b^{a_{\text{max,lb}}} (1 + b^{a_{\text{max,lb}} - a_{\text{max,lb}}}) + \cdots + b^{a_{\text{max,lb}} - a_{\text{max,lb}}}
\]

As showed in Eq. (25), every \( b^{a_{\text{max,lb}} - a_{\text{max,lb}}} \) is between 0 and 1. Therefore:
1 < \sum_{i=1}^{n} b^{a_{i_{lb}}-a_{\text{max}_{lb}}} \leq n \Rightarrow \sum_{i=1}^{n} b^{a_{i_{lb}}-a_{\text{max}_{lb}}} = \sum_{i=1}^{n} 2^{2t(a_{i_{lb}}-a_{\text{max}_{lb}})} \quad (30)

Then, rounding $2^{2t(a_{\text{max}_{lb}}-a_{i_{lb}})}$ to the nearest integer and approximating the sum with $r+f$ where $1 \leq r \leq n$ and $0 \leq f < 1$ as follows:

$$\approx a_{\text{max}_{lb}} + \sum_{i=1}^{n} 2^{-2t(a_{\text{max}_{lb}}-a_{i_{lb}})} \approx a_{\text{max}_{lb}} + \log_{b}(r+f) = a_{\text{max}_{lb}} + \log_{b}(1 + \frac{f}{r})$$

$$= a_{\text{max}_{lb}} + \log_{b}r + \log_{b}\left(1 + \frac{f}{r}\right)$$

(31)

Similar to Eq. (21), $r$ can be approximated with $2^{k}$ which makes computation of Eq. (31) simpler. Then, the base of $\log_{b}(1 + \frac{f}{2^{t}})$ can be changed with Eq. (7) as follows:

$$\approx a_{\text{max}_{lb}} + \log_{b}2^{k} + \log_{b}\left(1 + \frac{f}{2^{k}}\right) = a_{\text{max}_{lb}} + \frac{\log_{2}2^{k}}{\log_{2}b} + \frac{\log_{2}\left(1 + \frac{f}{2^{k}}\right)}{\log_{2}b} \approx a_{\text{max}_{lb}}$$

$$+ \frac{k}{2^{t}} + \frac{f}{2^{k}} = a_{\text{max}_{lb}} + 2^{-t} \times k + (f \gg (k + t))$$

(32)

### 3.5 Comparison of ulog with other number systems

Posit is a number system that shows FP numbers in a new way. It was introduced in 2015 by Gustafson [20] as a replacement for IEEE 754 FP numbers. posit provides higher accuracy and dynamic range than IEEE 754 FP numbers without overflow to infinity or underflow to zero [19]. It belongs to tapered precision numbers, which allow variable size entries for precision and dynamic range. Therefore posit can adjust the trade-off between precision and dynamic range [9]. ulog take many characteristics of IEEE 754 FP to be able to run on current hardware, and has fixed size precision and dynamic range. However, it provides more choices for accuracy than IEEE 754. posit can offer more precision with a smaller number of bits, therefore, low-precision posits provide higher accurate solutions than approximate computing methods [20]. However, it could not execute on current CPUs and needs a new hardware design. Conversely, ulog and RNS are number systems that are not far from the existing architecture of CPUs. Both use integer arithmetic units to emulate their computations. however, ulog and RNS need conversion to execute on current hardware. ulog is based on LNS, which is designed for approximate computation that is not the main target of posit. RNS is a non-weighted number system that is inherently unsigned integer-based [29]. Therefore RNS is useless for approximate computation that is defined on decimal numbers. However, Konstantin Isupov (Isupov et al. 2020) recently tried to design multiple precision of RNS, but ulog supports both FP
and integer numbers without additional cost. RNS supports addition, subtraction and multiplication and converts each number to multiple smaller numbers that can operate on each part independently and in parallel. Division and comparison are not supported by RNS [6] but ulog supports both comparison and division, where division, multiplication, and exponentiation have lower overhead than FP numbers. On the other hand, addition and subtraction that have low overhead in FP, posit, and RNS are costly in ulog.

3.6 Implementation of ulogd16 and operations on unsupported AVX512BW Intel intrinsics

Most Intel AVX512 CPUs do not support AVX512BW intrinsics [21], which directly allows doing operations on 16-bit integers. Therefore in this work, we have two types of implementations for 16-bit ulog types: one for supported and the other for unsupported AVX512BW intrinsics. In unsupported CPUs, the 16-bit ulog types convert to ulogd32 after reading from memory, and at the end of the computation, convert back to 16 bits width. Every ulog16 type is a fixed-point number that appends zero at the end of each number which does not change the actual value. Therefore, the conversion inserts zero at the end of each 16-bits number and transforms it to 32-bit ulogd32. After conversion, every input vector is converted to two vectors with ulogd32 elements. Therefore, all
ulogd32 operations could be used in this situation. This implementation does not affect memory transactions because reading and writing from/to memory are not different from AVX512BW CPUs. Figure 6a shows the pseudocode of converting ulogs16/ulogd16 to ulogd32, and Fig. 6b shows the pseudocode of merging two ulogd32 vectors into one ulog16 vector. The merging input is two vectors with ulogd32 numbers, and the output is one ulog16 vector.

4 Experimental evaluation

Our analysis is performed on a machine with Intel Xeon platinum 8168, 527 GB RAM, and GCC version 10.2. The sizes of L1, L2, and L3 caches on this machine are 1.5 MB, 24 MB, and 33 MB, respectively. For testing ulog scheme, we used six linear algebra problems in BLAS levels 1, 2, and 3. These benchmark functions are shown in Table 2. All functions have many applications in different problems. Addition and reduction by addition are very time-consuming operations in LNS, so dot product (DOTP) and l1 normalization (NRMZ) functions are chosen to analyze the performance of addition in Eq. (15) and the proposed method in Eq. (22). Compared to general matrix–matrix multiplication (GEMM), general matrix–vector multiplication (GEMV), and sparse matrix–vector multiplication (SPMV), both functions (i.e., DOTP and NRMZ) have less data transfer and cache misses. Therefore, the performance of the software-based implementation of ulog operations is accomplished successfully. GEMM, GEMV, and SPMV

| Name                          | Abbreviation | BLAS level |
|-------------------------------|--------------|------------|
| Dot product                   | DOTP         | Level 1    |
| Scale                         | –            | Level 1    |
| l1 Normalization              | NRMZ         | Level 1    |
| General matrix–vector multiplication | GEMV | Level 2   |
| Sparse matrix–vector multiplication | SPMV | Level 2   |
| General matrix–matrix multiplication | GEMM | Level 3   |

Table 2 Benchmark functions for testing ulog number system

| Size  | m   | k   | n   |
|-------|-----|-----|-----|
| s1    | 930 | 128 | 128 |
| s2    | 930 | 320 | 320 |
| s3    | 930 | 512 | 512 |
| s4    | 930 | 704 | 704 |
| s5    | 930 | 928 | 928 |
| s6    | 1860| 1024| 1024|
| s7    | 2790| 2016| 2016|
| s8    | 3720| 3008| 3008|

Table 3 The size of matrices in GEMM
Fig. 7 The illustration of blocked GEMM algorithm [25]

```
for p = 0, 1, ..., k - 1 in steps of kb do
    Pack B(p : p + kb - 1, 0 : n - 1) into \( \tilde{B} \);
    for i = 0, 1, ..., m - 1 in steps of mb do
        Pack A(i : i + mb - 1, p : p + kb - 1) into \( \tilde{A} \);
        for jr = 0, 1, ..., n - 1 in steps of nr do
            for ir = 0, 1, ..., mb - 1 in steps of mr do
                \( \hat{A} = \tilde{A}(ir : ir + mr - 1, :) \);
                \( \hat{B} = \tilde{B}(jr : jr + nr - 1, :) \);
                \( \hat{C} + = \hat{A} \times \hat{B} \);
            end
        end
    end
end
```

Fig. 8 The pseudocode of the blocked GEMM algorithm [25]
are memory-intensive and have many applications in different problems, especially in machine learning and neural networks, so improving the performance of these functions brings about significant advantages in numerical computation. The matrices in GEMM and GEMV are not square, as shown in Table 3 where $m$, $k$, and $n$ denote the number of rows in the first input matrix, the number of columns/rows in the first/second input matrix, and the number of columns in the second input matrix, respectively. The resulting matrix is $m \times k$ on GEMM.

To deeply analyze the uLog number system, we use an optimized blocked matrix multiplication technique in GEMM to improve data reuse and significantly reduce cache misses [25]. The blocked matrix multiplication splits each matrix into small blocks to fit into the cache, and data are accessed quickly when each block is in the cache. This technique decreases the cost of the most intensive operations in GEMM, memory operation, and helps achieve more distinct results. We use the algorithm introduced in [17] since it has one of the best performances among different blocked matrix multiplications [25]. Figures 7 and 8 illustrate the blocked GEMM and the pseudocode of this algorithm, respectively. This paper uses the relations in [25] to calculate the blocked GEMM algorithm parameters.

According to [25], $m_b$, $k_b$, $n_r$, and $m_r$ are chosen such that $\hat{A}, \hat{B}, \hat{C}$ fit in L2 cache. The formulas in Eqs. (33)–(38) show the relation of [25] for computation of the blocked GEMM parameters:

$$(m_b \times k_b + k_b \times n_r + m_r \times n_r)S_{\text{length}} \leq \frac{\text{size of L2 cache}}{2}$$  \hspace{1cm} (33)

$$m_b = x \times m_r \text{ such that } x \geq 1$$  \hspace{1cm} (34)

$$n_r \% S_{\text{num}} = 0 \text{ and } \frac{n_r}{S_{\text{num}}}(m_r+1) \leq (\text{total number of registers})$$  \hspace{1cm} (35)

$$m = x \times m_b \text{ such that } x \geq 1$$  \hspace{1cm} (36)

$$n = x \times n_r \text{ such that } x \geq 1$$  \hspace{1cm} (37)

where $S_{\text{length}}$ is the length of each number in bytes, and $S_{\text{num}}$ is the total number of data that could be loaded on one register and computed as follows:
where $R_{\text{length}}$ is the bit length of each register, and its value in AVX512 is equal to 512. The blocked GEMM uses the total number of registers, that is 32 for computation, but this number is halved in $\text{ulogs16}$ and $\text{ulogd16}$ because each 512-bit register with $\text{ulogs16}/\text{ulogd16}$ numbers is converted to two 512-bit registers with $\text{ulogs32}/\text{ulogd32}$ numbers. Therefore, half of the registers store the second parts of each 512-bit register. The parameters of blocked GEMM are calculated based on Eqs. (33)–(38), and the specification of Intel Xeon platinum 8168 CPU are shown in Table 4 for different types.

Table 5 shows the values of $m$ and $k$ for GEMV, where $m$ is the number of rows in the input matrix and $k$ is the number of columns/elements in the input matrix/vector. Different formats represent sparse matrix in SPMV, and ELLPACK is suitable for vector and SIMD architecture [1, 25]. Therefore, this paper uses the ELLPACK method for sparse matrices in SPMV. Table 6 displays the size of two different samples for sparse matrices based on ELLPACK in SPMV.

\begin{equation}
S_{\text{num}} = \frac{R_{\text{length}}}{S_{\text{length}} \times 8} \tag{38}
\end{equation}

where $R_{\text{length}}$ is the bit length of each register, and its value in AVX512 is equal to 512. The blocked GEMM uses the total number of registers, that is 32 for computation, but this number is halved in $\text{ulogs16}$ and $\text{ulogd16}$ because each 512-bit register with $\text{ulogs16}/\text{ulogd16}$ numbers is converted to two 512-bit registers with $\text{ulogs32}/\text{ulogd32}$ numbers. Therefore, half of the registers store the second parts of each 512-bit register. The parameters of blocked GEMM are calculated based on Eqs. (33)–(38), and the specification of Intel Xeon platinum 8168 CPU are shown in Table 4 for different types.

Table 5 The size of matrices and vectors in GEMV

| Size | $m$   | $k$   |
|------|--------|--------|
| s1   | 27,776 | 6016   |
| s2   | 27,776 | 7008   |
| s3   | 27,776 | 8000   |
| s4   | 27,776 | 9024   |
| s5   | 27,776 | 10,016 |
| s6   | 27,776 | 20,000 |

Table 6 The size of matrix in SPMV

| Sample | Number of rows | Number of columns | Number of non-zero |
|--------|----------------|-------------------|-------------------|
| S1     | 16,240,000     | 16,240,000        | 232,232,816       |
| S2     | 8,345,600      | 8,345,600         | 118,931,856       |

\begin{equation}
S_{\text{num}} = \frac{R_{\text{length}}}{S_{\text{length}} \times 8} \tag{38}
\end{equation}
Fig. 9 Runtime comparison of DP, SP and ulogd types based on serial implementation
4.1 Experimental evaluation of runtime in Intel Xeon platinum 8168

This subsection evaluates the runtime of various number formats based on both serial and vectorized implementations. First, serial implementations of all number systems are analyzed. The Eq. (15) has been used for serial implementations of DOTP, NRMZ, GEMM, and GEMV. Figure 9 shows the serial mode execution time of DOTP, scale, NRMZ, GEMM, GEMV, and SPMV for DP and conversions to SP, ulogd32, and ulogd16. As expected, the SP has the best performance in all functions except scale because the length of data is halved, and all the operations are hardware-based as opposed to ulog scheme in that all operations are implemented in software. The serial implementation of all ulog types is faster than DP in scale function because the scale function is converted to addition in ulog compared to multiplication in DP and SP. Therefore ulogd32 and ulogd16 have better performance than SP for a large number of data in scale function. Moreover, the GEMM results in Fig. 9 indicate that the performance of ulogd16 gets better than DP with the growth of data number. Besides, ulogd16 has the best speed on the scale because the length of data is only 16 bits. The speedup of ulogd16 and ulogd32 in the scale function increases on average 60.89% and 41.32%, at most by 63.28% and 44.57%, respectively. Unlike other functions, on SPMV, ulogd16 has a lower speed than ulogd32, because many cache misses occur on every 16 bits of data in ulogd16, but in ulogd32 the cache misses appear on every 32 bits of data. Therefore, the performance of ulogd16 is worse than ulogd32 in this case. Figure 10 shows the total execution time of SP and ulogs16 for all functions. Furthermore, it can be seen from Fig. 10 that despite the software-based implementation of ulog, the serial implementation of ulogs16 is better than the serialized SP for scale function. The average and the maximum improvement of ulogs16, in this case, are 27.27% and 35.68%, respectively.

Figure 11 shows the results of the vectorized implementation of all types on all sample functions. The dot product function in DOTP, NRMZ, GEMM, and GEMV is implemented with the vectorized version of Eq. (15), which is presented in Eq. (22). ulogd16-bw is the fastest among others according to DOTP function in Fig. 11. All ulogd types in DOTP function have good performance compared to DP. However, DOTP does not have significant cache misses and includes reduction, which is the most time-consuming operation in LNS. Figure 11 shows that memory operations are time-consuming in DOTP function. Predictably, ulogd16-bw is better than ulogd16 because of the extra conversion of each ulogd16 to ulogd32 before any procedures. Vectorized SP is better than ulogd32 due to having the same data length, software-based operations in ulogd32, and slow reduction in ulog scheme. The average speedups of ulogd16-bw, ulogd16, and ulogd32 than DP are 51.42%, 46.17%, and 24.44%, respectively. Besides, ulogd16-bw, ulogd16, and ulogd32 improve runtime by at most 55.63%, 51.20%, and 35.07% than DP, respectively.

The result of NRMZ is slightly different from DOTP. The effect of decreasing data length in NRMZ is not the same as DOTP, because the amount of data transformation is lower than DOTP function. DOTP consists of dot multiplication of two separate vectors, but NRMZ consists of dot multiplication of only one vector. Therefore, halving data length could not reduce the cost of software-based operations in ulogd32, and therefore it is not better than DP. SP is the fastest format in NRMZ.
Fig. 10  Runtime comparison of SP and $u\log s16$ based on serial implementation
Fig. 11  Runtime comparison of DP, SP, and ulogd types based on vectorized implementation
Converting 64 bits FP numbers to 16 bits causes \textit{ulogd16} and \textit{ulogd16-bw} to perform better than DP. The improvement of DP in NRMZ by \textit{ulogd16-bw} and \textit{ulogd16} is on average 22.64% and 8.50%, and at most 31.20% and 16.63%, respectively.

\textit{ulogd16-bw} is the fastest for the GEMM and GEMV functions since they have many data movements and cache misses. However, \textit{ulogd16} is not faster than SP in GEMM due to the time-consuming operation of converting 16 bits data to 32 bits together with doing all operations on 32 bits data. Moreover, all \textit{ulogd} types in GEMM and GEMV are better than DP. In GEMM the average enhancement of the speed of DP is 41.87%, 19.16%, and 10.30% for \textit{ulogd16-bw}, \textit{ulogd16}, and \textit{ulogd32}, respectively. Furthermore, the maximum runtime improvement of \textit{ulogd16-bw}, \textit{ulogd16}, and \textit{ulogd32} than DP is 57.63%, 38.54%, and 46.05%, respectively. The speed of DP in GEMV is increased on average by 52.89%, 44.49% and 28.13% and at most by 68.40%, 61.40% and 38.52% with \textit{ulogd16-bw}, \textit{ulogd16} and \textit{ulogd32}, respectively.

The amount of memory access in scale is the same as NRMZ. However, all \textit{ulogd} types are the fastest compared to DP and SP because the multiplication operation is converted to addition in LNS. Moreover, \textit{ulogd16-bw}, \textit{ulogd16}, and \textit{ulogd32} can increase the performance of scale on average by 70.26%, 67.60%, and 46.36%, and at most by 84.24%, 75.85%, and 52.83% than DP type, respectively.

SPMV has the highest number of cache misses among all sample functions, and this, together with software-based implementation, causes to reduce the effect of data length decreasing. However, \textit{ulogd16-bw} and \textit{ulogd16} are slightly improved SPMV runtime in s2 sample by 8.02% and 2.45%, respectively. The hardware-based operations help SP to have the best performance among other types. Vectorized implementation in SP type has different results. SP is fast on the platform based on Intel Xeon Platinum. However, inefficient reduction prevents any improvement in DOTP and NRMZ functions. Without inefficient reduction, conversion of multiplication to addition and data length reduction lead to a very fast scale function in \textit{ulogs16} and \textit{ulogs16-bw}. Besides, \textit{ulogs16-bw} and \textit{ulogs16} improve runtime on average 51.89% and 49.68% and at most 53.72 and 51.17% than SP, respectively. Moreover, \textit{ulogs16-bw} increases the speed by 7.77% than SP on the biggest size, and on other sizes, we do not have any improvement. The cost of converting 16-bit \textit{ulogs16} numbers to 32-bit prevents \textit{ulogs16} from being fast in GEMM. Both \textit{ulogs16-bw} and \textit{ulogs16} could improve the performance on average by 28.15% and 16.40% and at most 49.86% and 3.63% than SP, respectively. As previously mentioned, the random memory access required in SPMV, fast SP operations, and software-based operation of \textit{ulogs} types diminish the effect of data length reduction. Therefore we do not have any improvement in SPMV function. Figure 12 shows the result of comparing runtimes of SP with \textit{ulogs16} and \textit{ulogs16-bw} based on vectorization.

4.2 The conversion time in platinum

This section evaluates the amount of time needed to convert DP to SP, \textit{ulogd32}, and \textit{ulogd16}. The conversion time of \textit{ulogd16-bw} is not considered because it is equal to \textit{ulogd16}. It can be seen from Fig. 13 that \textit{ulogd16} and then \textit{ulogd32} have lower
Fig. 12  Runtime comparison of SP and ulogs16 based on vectorized implementation
Fig. 13 Runtime of conversion from DP to other formats
execution times than SP with the growth of the number of data. The conversion in ulog scheme is done in software. However, it is straightforward since this conversion only includes truncation of mantissa bits of FP numbers, and dynamic range remains unchanged. On the other hand, conversion of DP to SP consists of simultaneously changing dynamic range and mantissa bits of the number, which degrades the performance even on hardware-based formats like SP. Therefore, the software-based conversion of DP to ulogd types has a lower cost than that for DP to SP. The speed of ulogd32 is on average 5.53%, 7%, 14.40%, 8.23%, and 5% and at most 13.35%, 17.66%, 29.23%, 34.48% and 8.97% faster than SP in DOTP, NRMZ and scale, GEMM, GEMV and SPMV, respectively. The performance of conversion of DP to ulogd16 increases on average by 39.34%, 27.59%, 52.24%, 46.28% and 41.73%, and at most by 50.35%, 51.88%, 72.22%, 67.66% and 45.39% compared to SP in DOTP, NRMZ, scale, GEMM, and GEMV, respectively.

4.3 Accuracy Evaluation of ulog

In this section, we analyze the accuracy of the ulog scheme. Three formulas add error to ulog: (1) conversion formula in Eq. (6), (2) formula of Eq. (28) for addition, and Eq. (3) reduction by addition in Eq. (32). Moreover, this error could get larger when the number and complexity of computations increase. Therefore evaluating the error of the ulog scheme is very important. On the other hand, limiting mantissa bits of ulog and using bases other than 2 are two methods we used to control the error. DOTP and scale functions are chosen to evaluate the error in the formulas of Eqs. (6), (28), (32) in the three bases $\sqrt{2}$, 2, and 4 to compare the effect of choosing different bases on the error. The error is analyzed for ulogd32, ulogd16, and ulogs16 types of ulog for different numbers of data. We implemented two versions of DOTP, the serial and vectorized version that uses the Eqs. (28) and (32), respectively. Scale function is also chosen because the only error-prone formula in it is Eq. (6). Besides, all inputs are considered between zero and one. The size in all figures is the number of data on each array in DOTP and scale functions. The mean squared error (MSE) is used to measure the error in all formulas. Moreover, we apply the base 10 logarithmic scale on the error in addition and reduction by addition to show the details of the difference in errors.

As shown in Fig. 14, using base 4 for reduction by addition in Eq. (32) has the minimum error compared to other bases. However, base 2 has the lowest error for the small numbers of data. Moreover, it can be seen that the errors in bases 2 and square root of 2 increase with the growth of sizes, but base 4 limits the error. However, the behaviour of the formula (28) is different, and the errors of all bases grow as the sizes of data enhance. The lowest error for ulogd32 and ulogd16 occurs in bases 2 and 4, respectively. The effect of base 4 in reduction by addition of ulogd16 is interesting, and it adds some variation to the figure. However, the total flow is ascendant by enhancing size.

The errors of addition and reduction by addition in ulogs16 behave almost the same (Fig. 15), and the lowest error in both is the base 4. However, for small sizes,
Fig. 14 The error analysis of ulogd32 and ulogd16 with different bases in comparison with DP
Fig. 15  the error analysis of \textit{ulogs16} on different bases in comparison with SP
the lowest error occurs in base 2. Furthermore, base 4 could limit the increase in error by the growth of size, although the growth restriction is more on ulogd32. All the results for addition (formula of (28)) and reduction by addition (formula of (32)) show that increasing the size of data could enhance the error, albeit choosing the proper base can decrease and restrict the error in most cases. Although base 4 can decrease the gradient of error growth, especially for ulogd32, choosing different bases has a smaller effect on the error growth for the addition function in ulogd32 and ulogd16.

The error in the scale function is independent of the size of the data. This function can represent the maximum error of conversion to ulog in formula Eq. (6). In most cases, the base 2 in all ulog types can reduce the maximum error. The best case occurs in ulogd32 and ulogd16, and as it can be seen in Fig. 14, the variation of minimum and maximum decreases significantly.

5 Conclusion and discussion

This paper introduced a logarithmic-based number system, ulog, which is completely implemented in software. After representing different equations for conversion, multiplication, exponentiation and addition in ulog in the first part of the paper, we proposed a new approximate equation for reduction by addition. Compared to the previous addition formula, the new equation has lower time complexity and could be utilized in the vectorized application. Both addition and reduction by addition formulas remove the need for lookup tables, limiting the number of fraction bits. Therefore we could increase the mantissa bits and the precision of ulog. Moreover, the structure of ulog in other logarithm bases is analyzed and general addition and reduction by addition equations for different logarithm bases are introduced. The runtime of the ulog scheme was analyzed on various BLAS functions and compared with native DP and SP types. Two versions of ulog were implemented for runtime evaluations: serial and vectorized. For precise analysis, the addition formula (28) was used for serial implementation of dot product, and the vectorized dot product function was implemented with the reduction by addition Eq. (32). Despite the cost of software implementation of all ulog operations, addition, and reduction, the ulogd types improve runtime of all sample functions except SPMV. The massive number of cache misses in SPMV prevents any improvement. The best performance of ulog types is in scale, GEMM and GEMV, even though we use a blocked GEMM algorithm that significantly reduces the cost of memory access. These results demonstrate that converting multiplication to low-cost addition and decreasing the length of FP numbers substantially impacts performance, especially in memory-intensive applications. Besides, this work showed that when the data size grows, the speed of hardware-based conversion of DP to SP is significantly lower than the software-based conversion of DP to ulogd types. Moreover, the error of all ulog types and the effect of choosing different bases on the error were evaluated. The formula of Eq. (6) for conversion to ulog, the addition formula of (28), and the reduction by addition formula of (32) are three approximate equations that add error to different types of ulog. Therefore, we chose DOTP and scale functions to analyze each equation’s
error. The results showed that base 4 could reduce the error of reduction by addition in ulogd32, ulogd16, and ulogs16, and the addition in ulogs16 significantly. The base should get bigger or smaller to influence error in some situations. For example, base 4 in the addition equation in ulogd32 and ulogd16 has a minor impact on error. In general, according to Alam et al. [1], we can conclude that input distribution strongly affects the base choice. For example, Miyashita et al. [28] showed that the error of weights in a neural network for base $\sqrt{2}$ is half of the base 2. Therefore, the logarithm base should be selected based on the analysis of errors on different bases. Indeed the logarithm bases help extend the types of ulog with diverse dynamic ranges and precision at no cost, because it only depends on logarithm base 2 as the backend support. Therefore, investigating different bases for different applications in detail could be an open research line. This research was used the truncation rounding method when converting FP to ulog. One can analyze the error when applying other rounding methods for future works. Combining the lookup table technique with the approximate addition and reduction by addition equations could be a better solution for error reduction for further works. For example, ulogd16 has 4 bits for mantissa, and it will have a lower error if implemented with a lookup table without removing mantissa bits. On the other hand, the ulog types with high precision like ulogd32 can be implemented by the approximate addition and reduction by addition equations, and the large width of mantissa can decrease the error of approximate equations.

Data availability The SPMV datasets are collected from the university of Florida sparse matrix collection and are available at https://sparse.tamu.edu/. All other data generated randomly during programs execution.

Declarations

Conflict of interest The authors have no conflicts of interest to declare that are relevant to the content of this article.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Alam SA, Garland J, Gregg D (2021) Low-precision logarithmic number systems: beyond base-2. ACM Trans Archit Code Optim 18(4):47. https://doi.org/10.1145/3461699
2. Alam SA, Gustafsson O (2014) Design of finite word length linear-phase FIR filters in the logarithmic number system domain. VLSI Des 2014:3. https://doi.org/10.1155/2014/217495
3. Anderson A, Muralidharan S, Gregg D (2017) Efficient multibyte floating point data formats using vectorization. IEEE Trans Comput 66(12):2081–2096. https://doi.org/10.1109/TC.2017.2716355
4. Arnold MG (2002) Reduced power consumption for MPEG decoding with LNS. In: Proceedings IEEE International Conference on Application-Specific Systems, Architectures, and Processors, 2002. IEEE, pp 65–75. https://doi.org/10.1109/ASAP.2002.1030705
5. Basetas C, Kouretas I, Paliouras V (2007) Low-power digital filtering based on the logarithmic number system. In: International Workshop on Power and Timing Modeling, Optimization and Simulation. Springer, pp 546–555. https://doi.org/10.1007/978-3-540-74442-9_53
6. Chang C-H, Molahosseini AS, Zarandi AAE, Tay TF (2015) Residue number systems: a new paradigm to datapath optimization for low-power and high-performance digital signal processing applications. IEEE Circuits Syst Mag 15(4):26–44. https://doi.org/10.1109/MCAS.2015.2484118
7. Coleman JN, Chester E, Softley CI, Kadlec J (2000) Arithmetic on the European logarithmic microprocessor. IEEE Trans Comput 49(7):702–715. https://doi.org/10.1109/12.863040
8. Coleman JN, Softley CI, Kadlec J, Matousek R, Tichy M, Pohl Z, Hermanek A, Benschop NF (2008) The European logarithmic microprocessor. IEEE Trans Comput 57(4):532–546. https://doi.org/10.1109/TC.2007.70791
9. Cook JD (2018) Anatomy of a posit number. John D. Cook. https://www.johndcook.com/blog/2018/04/11/anatomy-of-a-posit-number/. Accessed 30 June 2022
10. Flegar G, Scheidegger F, Novaković V, Mariani G, Tom’s AE, Malossi ACI, Quintana-Ortí ES, (2019) FloatX: a C++ library for customized floating-point arithmetic. ACM Trans Math Softw 45(4):40
11. Gao M, Gu G (2018) Estimate and recompute: a novel paradigm for approximate computing on data flow graphs. IEEE Trans Comput Aided Des Integ Circuits Syst 39(2):335–345. https://doi.org/10.1109/TCAD.2018.2889662
12. Gautschi M, Schaffner M, Gürkaynak FK, Benini L (2016) 4.6 A 65nm CMOS 6.4-to-29.2 plj/ FLOP® 0.8 V shared logarithmic floating point unit for acceleration of nonlinear function kernels in a tightly coupled processor cluster. In: 2016 IEEE International Solid-State Circuits Conference (ISSCC), 2016. IEEE, pp 82–83. https://doi.org/10.1109/ISSCC.2016.7417917
13. Ghabeli H, Sabbagh Molahosseini A, Emrani Zarandi AA, Sousa L (2021) Variable Latency Carry Speculative Adders with Input-based Dynamic Configuration. Comput Electr Eng 93:107247. https://doi.org/10.1016/j.compeleceng.2021.107247
14. Grützmacher T, Anzt H (2019) A modular precision format for decoupling arithmetic format and storage format. In: Euro-Par 2018: Parallel Processing Workshops. Springer, Cham, pp 434–443. https://doi.org/10.1007/978-3-030-10549-5_34
15. Grützmacher T, Cojean T, Flegar G, Anzt H, Quintana-Ortí ES (2020) Acceleration of pagerank with customized precision based on mantissa segmentation. ACM Trans Parallel Comput 7(1):4. https://doi.org/10.1145/3380934
16. Grützmacher T, Cojean T, Flegar G, Göbel F, Anzt H (2020) A customized precision format based on mantissa segmentation for accelerating sparse linear algebra. Concurr Comput Pract Experience 32(15):e5418. https://doi.org/10.1002/cpe.5418
17. Gunnels JA, Henry GM, van de Geijn RA (2001) A family of high-performance matrix multiplication algorithms. In: Computational Science—ICCS 2001. Springer Berlin Heidelberg, pp 51–60. https://doi.org/10.1007/3-540-45545-0_15
18. Gustafson J (2010) Exascale: Power, cooling, reliability, and future arithmetic. In: HPC User Forum Seattle
19. Gustafson J (2017) Posit arithmetic. Mathematica Notebook describing the posit number system 30
20. Gustafson JL, Yonemoto IT (2017) Beating floating point at its own game: Posit arithmetic. Supercomput Front Innov 4(2):71–86
21. Intel (2021a) Intel AVX512-FP16 Architecture Specification. Intel. https://software.intel.com/content/www/us/en/develop/download/intel-avx512-fp16-architecture-specification.html. Accessed 17 Aug 2021
22. Intel (2021b) Intel Intrinsics Guide. Intel. https://software.intel.com/sites/landingpage/IntrinsicsGuide/. Accessed 17 Aug 2021
23. Kouretas I, Basetas C, Paliouras V (2012) Low-power logarithmic number system addition/subtraction and their impact on digital filters. IEEE Trans Comput 62(11):2196–2209. https://doi.org/10.1109/TC.2012.111
24. Lee EH, Miyashita D, Chai E, Murmann B, Wong SS (2017) Lognet: energy-efficient neural networks using logarithmic computation. In: IEEE International Conference on Acoustics, Speech and
25. Lim R, Lee Y, Kim R, Choi J (2018) An implementation of matrix–matrix multiplication on the Intel KNL processor with AVX-512. Clust Comput 21(4):1785–1795. https://doi.org/10.1007/s10586-018-2810-y

26. Lomont C (2011) Introduction to intel advanced vector extensions. Intel. https://software.intel.com/content/dam/develop/external/us/en/documents/intro-to-intel-avx-183287.pdf. Accessed 17 August 2021

27. Lu J, Fang C, Xu M, Lin J, Wang Z (2020) Evaluations on deep neural networks training using posit number system. IEEE Trans Comput 70(2):174–187. https://doi.org/10.1109/TC.2020.2985971

28. Miyashita D, Lee EH, Murmann B (2016) Convolutional neural networks using logarithmic data representation. arXiv preprint arXiv:160301025

29. Molahosseini AS, De Sousa LS, Chang C-H (2017) Embedded systems design with special arithmetic and number systems. Springer. https://doi.org/10.1007/978-3-319-49742-6

30. Molahosseini AS, Vandierendonck H (2020) Half-precision floating-point formats for PageRank: opportunities and challenges. In: IEEE High Performance Extreme Computing Conference (HPEC), 2020. IEEE, pp 1–7. https://doi.org/10.1109/HPEC43674.2020.9286179

31. Molahosseini AS, Zarrandi AAE, Martins P, Sousa L (2017) A multifunctional unit for designing efficient RNS-based datapaths. IEEE Access 5:25972–25986. https://doi.org/10.1109/ACCESS.2017.2766841

32. Parhami B (2020) Computing with logarithmic number system arithmetic: Implementation methods and performance benefits. Comput Electr Eng 87:106800. https://doi.org/10.1016/j.compeleceng.2020.106800

33. Shlezinger N, Chen M, Eldar YC, Poor HV, Cui S (2020) UVeQFed: universal vector quantization for federated learning. IEEE Trans Signal Process 69:500–514. https://doi.org/10.1109/TSP.2020.3046971

34. Tagliavini G, Marongiu A, Benini L (2018) Flexfloat: a software library for transprecision computing. IEEE Trans Comput Aided Des Integr Circuits Syst 39(1):145–156. https://doi.org/10.1109/TCAD.2018.2883902

35. Van der Merwe M, Joseph V, Gopalakrishnan G (2019) Message scheduling for performant, many-core belief propagation. In: IEEE High Performance Extreme Computing Conference (HPEC), 2019. IEEE, pp 1–7. https://doi.org/10.1109/HPEC.2019.8916366

36. Vogel S, Liang M, Guntoro A, Stechele W, Ascheid G (2018) Efficient hardware acceleration of CNNs using logarithmic data representation with arbitrary log-base. Paper presented at the Proceedings of the International Conference on Computer-Aided Design, San Diego, California.

37. Wang N, Choi J, Brand D, Chen C-Y, Gopalakrishnan K (2018) Training deep neural networks with 8-bit floating point numbers. In: Proceedings of the 32nd International Conference on Neural Information Processing Systems, 2018. pp 7686–7695

38. Wang Y, Lin J, Wang Z (2017) An energy-efficient architecture for binary weight convolutional neural networks. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 26 (2):280–293. https://doi.org/10.1109/TVLSI.2017.2767624

39. Wolf W, Kandemir M (2003) Memory system optimization of embedded software. Proc IEEE 1:165–182. https://doi.org/10.1109/JPROC.2002.805823

40. Zuras D, Cowlishaw M, Aiken A, Applegate M, Bailey D, Bass S, Bhandarkar D, Bhat M, Bindel D, Boldo S (2008) IEEE standard for floating-point arithmetic. IEEE Std 754:1–70

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.