On the Compressibility of Floating-Point Data in Posit and IEEE-754 Representation

Andrew Rodriguez
andrew.rodriguez@txstate.edu
Department of Computer Science
San Marcos, Texas, USA

Martin Burtscher
burtscher@txstate.edu
Department of Computer Science
San Marcos, Texas, USA

Abstract

The IEEE 754 floating-point standard is the most used representation for real numbers in modern computer systems, despite issues in accuracy for certain applications. The posit format, which has several advantages, has been proposed as a direct drop-in replacement for IEEE floats. Many works compare the use of posits to floats in a wide range of scientific computing domains. However, there has not been any work looking into the compressibility of posit data. In this paper, we compare the compression ratios of different algorithms when the input is encoded in IEEE format and in posit format. We evaluate 5 lossless general-purpose compressors as well as several new compression algorithms synthesized by our LC framework on 14 single-precision inputs from the SDRBench suite, encoded in float and posit format. Our results show that that 4 of the 6 compressors yield an average of 2.59% reduction in compression ratio on posit data, whereas bzip2 provides a 1.74% increase in compression ratio, with XZ providing the highest ratios for both encodings.

CCS Concepts

• **Theory of computation → Data compression**.

Keywords

Data compression, IEEE 754 standard, Posit format

ACM Reference Format:
Andrew Rodriguez and Martin Burtscher. 2025. On the Compressibility of Floating-Point Data in Posit and IEEE-754 Representation. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3731599.3767372

1 Introduction

The IEEE 754 floating-point standard is the de facto numerical representation for real numbers in modern computer systems. Floating-point data is used extensively in scientific computing applications, especially in high-performance computing. However, floating-point operations are among the slowest and most energy-hungry machine instructions due to IEEE 754 functional units using substantial chip area and power to handle the many corner cases and exceptions described in the standard [10]. Furthermore, some studies have shown that the standard is error-prone [18], and different implementations of the standard may produce different results [31].

Some areas of scientific computing, particularly machine learning, have started using less precise floating-point types (e.g., float16, float8, etc.) in an effort to speed up computation—and in some cases reduce energy consumption—while reducing the accuracy. Google's bfloat16, NVIDIA's TensorFloat-32, and the practice of mixed precision all attempt to reduce the accuracy while preserving the dynamic range when possible. However, reducing precision is not a solution for applications where accuracy is key. As a consequence, some research has looked into alternative representations for real numbers. Gustafson introduced "posits" as a direct drop-in replacement for the IEEE 754 standard to alleviate some of the issues with floats [19]. Posits use a variable number of bits for the fraction and exponent fields, offering a larger dynamic range and higher accuracy compared to IEEE floats of the same size.

Since the introduction of posits, there have been many works comparing the use of posits to floats in different scientific computing domains. In general, posits are more accurate and, in some cases, faster than floats. Some prior works have proposed hardware units for posit arithmetic but only use hardware simulators in their evaluation. Most prior works studying the accuracy of posits convert existing IEEE 754 encoded data (input files, machine learning weights, etc.) into posit format before computation and do not store any values in posit format.

Scientific applications and instruments often emit huge amounts of floating-point data. For example, the Hardware/Hybrid Accelerated Cosmology Code (HACC) generates petabytes of data in a single simulation [20], and the Large Hadron Collider (LHC) produces approximately one petabyte of data per second [7]. Given the massive amount of data used and produced by many scientific applications, storing said data in posit format for applications that require it eliminates the need for converting between posits and floats before and after the computation. However, there are no prior studies looking into whether doing so affects the compressibility.

In this paper, we study the compressibility of posit data by comparing the compression ratios of different compression algorithms when the input is encoded in IEEE 754 format versus in posit format. We use single-precision IEEE floating-point data and convert each input file to use posit encoding. We investigate 5 existing general-purpose lossless compression algorithms and several new compression algorithms generated by our LC synthesis framework [13]. Note that there are no special-purpose compressors designed for

SC Workshops '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1871-7/2025/11
https://doi.org/10.1145/3731599.3767372

posit data, so we are unable to compare against compressors designed for floating-point data, in particular, lossy floating-point compressors. We use 14 single-precision datasets from 7 scientific domains from the SDRBench suite [33] as inputs for our study.

This paper makes the following main contributions.

- It provides the first study on the compressibility of posit data converted from IEEE formatted data, evaluating 6 lossless general-purpose compressors on 14 inputs from the SDRBench suite, encoded in IEEE 754 single-precision format and in posit<32,3> format.
- It shows that the XZ [3] compressor provides the highest compression on both float and posit encoded data, and that posit data can be compressed at a similar rate to float data with the same algorithm.
- It highlights that customized compression algorithms offer higher compression ratios than using a single algorithm for all inputs.

The rest of the paper is organized as follows. Section 2 provides background into the posit and IEEE floating-point formats. Section 3 summarizes related work. Section 4 describes the experimental methodology. Section 5 presents and discusses the results. Section 6 concludes the paper.

2 Background

The IEEE 754 floating-point standard describes the format and rules for floating-point data and arithmetic and is by far the most commonly used standard in computing for real numbers. A float has 3 fields: the sign, a biased exponent, and the mantissa. For a 32-bit float, there is 1 sign bit, an 8-bit exponent, and a 23-bit mantissa. Figure 1 shows the layout of these bits. The mantissa, or significand, is the fractional part of the number. It is stored as a binary fraction with an implicit 1 added to the fraction. The exponent is stored with a bias of half the maximum possible exponent value (with an 8-bit exponent, the bias is 127).

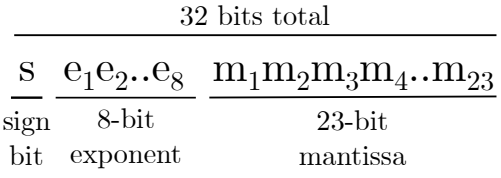

Figure 1: Bit layout of a 32-bit IEEE 754 float

Floats can represent normal finite numbers and several special numbers, including subnormals, Not-a-Number (NaNs), positive or negative 0, and positive or negative infinity. A float is a subnormal if the exponent is all zeros with a non-zero mantissa. Subnormals do not use the implicit 1. A float is a NaN if the exponent is all ones with a non-zero mantissa. A float is zero if all the bits are zero, and negative zero if all the bits except the sign bit are zero. A float is infinity if the exponent is all ones with an all zero mantissa.

Let s be the sign bit, e the biased exponent, and m the mantissa bit string. Assuming the float is a normal number, the value of the float v can be computed by the following formula:

$$v = (-1)^s \times 1.m \times 2^{e-127}$$

Introduced in 2017, posits are designed to be a drop-in replacement for IEEE 754 floats [19]. A posit has 4 fields: the sign, the regime, the exponent, and the fraction. The regime, exponent, and fraction fields are variable in length, and a posit is parameterized by the total number of bits n and the maximum number of exponent bits es, creating a posit<n, es> number. The addition of a regime field acts as additional scaling for the exponent of the fraction. For a 32-bit posit with es exponent bits, there is a sign bit, a run of regime bits, a es-bit exponent, and a run of fraction bits. Figure 2 shows the layout of the bits in a posit. The number of regime bits is computed by counting the run of identical bits k after the sign bit, terminated by an opposite bit. If the first bit of the regime is 0, then the regime r is equal to $-k$. If the first bit is 1, $r = k - 1$. The exponent begins after all regime bits. The exponent in posit format is not stored with a bias; it is stored as an unsigned integer. Similar to floats, the fraction in posit format has an implicit 1. Negative numbers are encoded in 2's complement format.

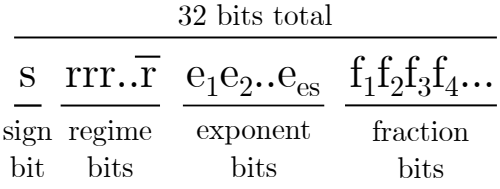

Figure 2: Bit layout of a 32-bit posit with es exponent bits

There are only 2 special numbers in the posit format: 0 and Not-a-Real (NaR), an umbrella value for anything not mathematically definable as a real number. Early iterations of the format referred to NaR as ± inf. Posits have no subnormals. If the posit bits are all zero, then the value is 0. If the sign bit is 1 and all other bits are zero, the value is NaR. There is only one 0 value.

Let s be the sign bit value, r the regime value, e the exponent value, m the number of fraction bits, F the bit string in the posit, and f the fraction of the posit, computed by the following expression:

$$f = 2^{-m}\Sigma_{l=0}^{m-1}F_l 2^l$$

Assuming the posit is neither 0 nor NaR, the value of the posit v can be computed with the following formula:

$$v = ((1 - 3s) + f) \times 2^{(1-2s)\times(4r+e+s)}$$

3 Related Work

Although there have been many related works studying the accuracy and energy benefits of using posits compared to IEEE 754 floats or fixed-point representation, there has not been any prior work looking into the compressibility of posits. Below, we summarize related works exploring the accuracy benefits in general and the (simulated) energy savings in machine learning, high-performance computing, and other scientific computing domains.

Ciocirlan et al. design and implement a posit arithmetic unit, called POSAR, and analyze the accuracy and efficiency of their implementation using a series of machine learning benchmarks [10]. The authors found that 32-bit posits outperform 32-bit floats in

terms of accuracy and execution speed but require 30% more FPGA resources than 32-bit floats.

Romanov et al. evaluate several popular test suites and programs used in machine learning (such as BLAS, GEMM, etc.) by comparing floats, posits, and bfloats [28]. The results show that software implementations of posit16 and posit32 have high accuracy but are slow. Furthermore, bfloat16 is about the same as float32 in accuracy but much better for performance. Therefore, bfloat and posit types each have a specific use case depending on the application.

Lu et al. explore reduced-precision posits in Deep Neural Networks (DNN) to address the enormous memory requirements and computational complexity [26]. The authors use multiple datasets (MNIST, CIFAR-10, ImageNet, and Penn Treebank) and compare their 8-bit posit DNN framework to a 32-bit float baseline. The posit framework performs on par with the baseline while requiring less memory. Furthermore, the authors provide a hardware posit implementation and run the training on a hardware simulator. Compared to the float hardware, the posit hardware achieves a reduction in terms of chip area, power usage, and memory requirement.

Carmichael et al. propose Deep Positron, a DNN framework supporting 8-bit posits [8]. Moreover, they propose an FPGA soft core supporting fixed-point, floating-point (IEEE 754), and posit arithmetic to run their experiments. Posits provide better accuracy than both 8-bit fixed-point and 8-bit floating-point values for 3D datasets, comparable to 32-bit float accuracy.

Langroudi et al. study the use of 8-bit posits in DNNs compared to 16-bit fixed-point representation using multiple datasets [23]. The results show that posits achieve better accuracy while also requiring a smaller memory footprint than a fixed-point format.

Nakasota et al. implement 32-bit posit arithmetic on FPGAs and GPUs to accelerate linear algebra operations [27]. For the FPGA implementation, the authors use Flo-Posit, an existing open-source project providing VHDL units supporting posits. For the GPU implementation, they port the necessary operations from the SoftPosit library to CUDA and OpenCL kernels. The posit arithmetic was found to be approximately 0.5–1.0 digits more accurate than IEEE 754 single-precision float arithmetic.

Leong and Gustafson argue that 16-bit fixed-point and IEEE 754 float types lack the accuracy required for 1024- and 4096-point Fast Fourier Transform computations, but 16-bit posits can be used instead [24]. Similarly, 32-bit posit FFTs are able to replace 64-bit IEEE 754 float FFTs for many high-performance computing tasks, improving speed, energy efficiency, and storage costs by about a factor of 2 for these workloads.

Chien et al. provide a 32-bit posit NAS parallel benchmark to explore the feasibility of posit encoding in high-performance applications [9]. The authors show that posits yield higher accuracy for all tested kernels and proxy-applications. The calculated overhead of using a software implementation of posits are 4 to 19 times slower than an IEEE 754 hardware implementation, highlighting the need for hardware units that support posits.

Fernandez-Hart et al. look at replacing IEEE 754 floats with posits when conducting spiking neuron simulations [16]. The authors compare the accuracy of the computation, the spike count, and the spike timing when using various-width posits and floats compared against a 64-bit float standard. They found that there was no difference in accuracy between 32-bit posits and 32-bit floats compared to the 64-bit reference. However, the 16-bit posits provided higher accuracy than the 16-bit floats when comparing to the standard, so much so that the 64-bit float reference could be replaced with the 16-bit posit version without significantly impacting the computation.

Klower et al. study the effects of using posit arithmetic as an alternative to float arithmetic for weather and climate models [22]. 16-bit posits with 1 or 2 exponent bits are more accurate than 16-bit floats, and the authors argue that reduced precision posit arithmetic in computational fluid dynamics will provide a benefit.

Esmaeel et al. implement the second-order infinite impulse response notch filter with two hardware posit versions, a standard posit system and a posit system with a faster multiplier [12]. These implementations are compared against an IEEE 754 implementation. With the standard posit implementation, posits provide 3.9 times higher accuracy than floats. With the improved multiplier version, posits perform better than floats on all metrics—chip area, speed, power, and energy.

Hou et al. explore a hardware implementation of posit arithmetic, using Xilinx Vivado Design Suite for the hardware design and Verilog HDL for the logic, and run the implementation through a simulator [21]. They found that, due to the variable exponent and fraction bits, posits require more logic for decoding but provide better accuracy compared to IEEE 754 floats.

Wu et al. present a design for a posit vector arithmetic unit in the Chisel language [32]. It supports vector operations such as addition, subtraction, multiplication, division, and dot product. The experiments show that the division operations are 95.85% accurate, while the other operations are 100% accurate.

Gohil et al. propose a fixed-posit representation to alleviate the complexity when handling a varying number of exponent and fraction bits. They evaluate the design of their fixed-posit multiplier on the error-resilient AxBench and OpenBLAS benchmarks. Compared to standard posit multipliers and IEEE 754 floating-point multipliers, the fixed-posit implementation provides savings for power, chip area, and delay. Furthermore, they observe a minimal quality loss when using fixed-posit.

Alouani et al. study the robustness of 32-bit posits and 32-bit floats to bit flips [5]. First, the authors present a theoretical analysis for IEEE 754 compliant numbers and posit numbers for single and double bit flips. Then, they perform an exhaustive fault injection experiment that shows that, in 95% of the tests, the posit representation is less impacted by faults than the float representation.

Lindstrom proposes an alternative block-based floating-point representation that utilizes a variable-length encoding of the exponent, rather than a fixed-length exponent like other block-based approaches, borrowing from universal number representations, such as posits [25]. Using numerical experiments with real data, the author shows that his posit-based approach may yield as much as six orders of magnitude increase in accuracy over conventional posits for the same amount of storage, and even more accuracy gains over IEEE floats.

4 Experimental Methodology

To evaluate the compressibility of posits, we took several IEEE 754 single-precision inputs from the SDRBench suite [33] and created

Table 1: Information about the 5 evaluated compressors

Name	Version	Source
bzip2	1.1.0	[30]
gzip	1.13	[1]
lz4	1.04	[2]
XZ	5.4.1	[3]
Zstd	1.5.1	[4]

copies with posit encoding using the cppposit library [29]. Although standard 32-bit posits use 2 maximum exponent bits ($es = 2$), we opted for a 32-bit posit configuration with a maximum of 3 exponent bits ($es = 3$) as this allows for the posit's dynamic range to capture more of the normal values from the IEEE 754 standard (see Section 4.2 for more detail).

We ran all evaluated compressors on both the IEEE and the posit version of the inputs. For each evaluated compressor, we report the geometric mean compression ratio across all tested inputs. We use the geometric mean rather than the arithmetic mean to help dampen inputs that significantly outperform the general case [17]. Since the compression ratio depends only on the algorithm and is independent of the hardware, we do not list the system specifications as the ratios will be the same on any system.

The rest of this section provides details on the selected inputs, the evaluated compressors, and on the LC framework.

4.1 Evaluated Compressors

Table 1 lists information on the 5 studied compressors. These compressors are designed for general-purpose data and are not specialized for floating-point data. We focus on general-purpose compressors since no special-purpose posit compressors exist yet. We ran each compressor with the `-best` flag (if there is an option to do so). Furthermore, we generated 2 compressors with LC, one for the IEEE inputs and one for the posit inputs, as outlined in Section 4.3.

4.2 Datasets

Table 2 provides information on the 7 datasets we use in our tests. The data is encoded in single-precision IEEE floats and encompasses a wide range of scientific domains. We choose 2 inputs from each dataset at random, excluding any small files from the pool of choices for each dataset. By choosing 2 inputs from each dataset, we avoid weighing datasets with more files differently. Table 3 lists the 14 selected inputs and their sizes.

Posits are more accurate for values close to 1.0 due to their variable fields and tapered accuracy, meaning some IEEE values that are far away from 1.0 may not be representable with a given posit configuration. Hence, when converting IEEE data to posit data, very large and very small values may incur a loss. To test how lossy this conversion is, we converted each posit file back to IEEE encoding and measured how many values differed from the original input.

When using a maximum of 3 exponent bits, the geometric mean over all 14 inputs yields 97.1% precise values. Of the 14 inputs we use, 8 incur no loss in the posit conversion and 4 have a very small amount of error introduced. The AEROD input yields 90% precise values, and the QRAIN input yields 73% precise values. These two files have very large and very small values, respectively, explaining the increased percentage of inexact values.

When using a maximum of 2 exponent bits, the geometric mean over all 14 inputs yields only 85.6% precise values. Due to this substantially lower ratio, we use $es = 3$ for our experiments rather than the standard 2 exponent bits for 32-bit posits.

Table 2: Information about the datasets

Name	Description
CESM	Climate simulation
EXAALT	Molecular dynamics simulation
HACC	Cosmology particle simulation
ISABEL	Weather simulation
NYX	Cosmology N-body simulation
QMC	Many-body ab initio Quantum Monte Carlo
SCALE	Climate simulation

Table 3: Information about the 14 single-precision inputs

Name	Dataset	Size
AEROD_v_1_1800_3600.f32	CESM	25 MB
ICEFRAC_1_1800_3600.f32	CESM	25 MB
dataset1.y.f32.dat	EXAALT	65 MB
dataset2.x.f32.dat	EXAALT	342 MB
vx.f32	HACC	1.1 GB
xx.f32	HACC	1.1 GB
CLOUDf48.bin.f32	ISABEL	96 MB
QRAINf48.bin.f32	ISABEL	96 MB
baryon_density.f32	NYX	512 MB
velocity_x.f32	NYX	512 MB
einspline.f32	QMC	602 MB
einspline.pre.f32	QMC	602 MB
PRES-98x1200x1200.f32	SCALE	539 MB
RH-98x1200x1200.f32	SCALE	539 MB

4.3 LC Framework

The LC framework is a tool that can automatically synthesize compression pipelines from a library of data transformations. Each stage in a pipeline is a single data transformation. LC has produced some state-of-the-art compressors for floating-point data [6, 14], highlighting that LC is capable of producing pipelines that are on par with the other evaluated compressors. We opted to search for 3-stage pipelines, resulting in a total of 236,196 lossless pipelines. We ran all 236,196 pipelines on both encodings of each file. To choose a single LC algorithm that works best for all IEEE inputs and another single LC algorithm that works best for all posit inputs, we selected the algorithm with the highest geometric-mean compression ratio across all tested inputs.

The resulting LC pipeline for float data uses the following 3 stages, all of which interpret the bits of each float word as an integer word: DIFFMS, RARE, and RAZE [6]. The DIFFMS component computes the difference sequence (also called "delta modulation") by subtracting the binary representation of the previous value

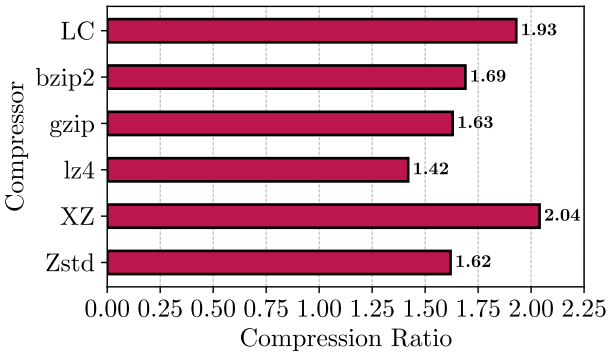

Figure 3: Geometric-mean compression ratios on IEEE float data

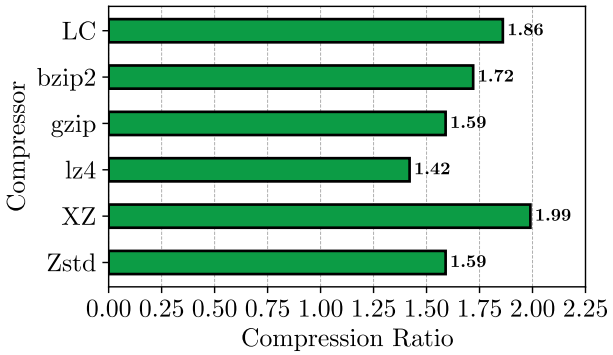

Figure 4: Geometric-mean compression ratios on posit data

from the current value and outputting the resulting difference in magnitude-sign format, which is often more compressible because it tends to produce values with many leading zero bits. If neighboring values correlate with each other, this tends to yield a more compressible sequence. The RARE component creates a bitmap in which each bit specifies whether the top k bits of a word match the top k bits of the prior word or not, where k is automatically picked to be optimal. It outputs the non-repeating k-bit words, the $32 - k$ remaining bits from all values, and a compressed version of the bitmap that is repeatedly compressed with the same algorithm. Lastly, the RAZE component works in the same way, except it checks whether the top k bits in each word are zero or not.

The LC pipeline for posit data uses the following 3 stages, all of which interpret the bits of each posit word as an integer word: DIFFNB, BIT, and RZE [15]. The DIFFNB component is similar to DIFFMS but outputs the resulting difference in negabinary rather than magnitude-sign format. The BIT component is often referred to as "bit shuffle" or "bit transpose". It takes the most significant bit of each value in the input and outputs them together, then it takes the second most significant bit of each value and outputs them, and so on down to the least significant bit. This improves compressibility if nearby values tend to have the same bits in certain positions. Lastly, the RZE component is similar to RAZE, except it operates on all bits of each word rather than only the top k bits.

5 Results

In this section, we study the compression ratio of the evaluated compressors on the selected IEEE and posit encoded inputs. First, we compare the compression ratios provided by the evaluated compressors on the float data and on the posit data. Then, we explore how the compression ratio improves when allowing a unique per-file LC pipeline for each input.

5.1 Posit vs. Float Compression

Figures 3 and 4 show the compression ratio achieved by each compressor on the float encoded data and the posit encoded data, respectively. The compression ratio runs along the x-axis and the compressors run along the y-axis. Higher ratios are better.

On the float data, XZ achieves the highest compression. LC provides the next highest compression, followed by bzip2. gzip and Zstd provide similar compression, and lz4 yields the lowest compression. XZ uses LZMA, a dictionary-based compression algorithm derived from LZ77. bzip2, gzip, and Zstd also use a dictionary approach, but XZ takes advantage of the large main memory available on modern systems, allowing for larger dictionaries providing better compression, explaining why bzip2, gzip, and Zstd achieve similar compression ratios whereas XZ provides the highest compression of the evaluated compressors. Lz4 also uses the LZ77 dictionary approach but does not combine it with an entropy stage (e.g., Huffman coding) like the other compressors do, resulting in the lowest compression ratio. However, lz4 is designed for fast compression, and removing this additional stage hurts the compression ratio but improves compression speed.

On the posit data, the ranking of the evaluated compressors remain the same. XZ provides the highest compression, followed by LC and bzip2. gzip and Zstd yield the same compression ratio, with lz4 achieving the lowest compression. All compressors except bzip2 and lz4 yield lower compression ratios on the posit inputs than on the float inputs. XZ, LC, gzip, and Zstd exhibit a small 2.45, 3.62, 2.45, and 1.85% reduction in compression ratio on posit data compared to float data, respectively. lz4 performs the same on both sets of inputs, and bzip2 yields a 1.74% increase in compression ratio on posit data.

Figure 5 shows the percentage of IEEE floats within each input that have a given exponent value. The biased exponent runs along the x-axis, and the percentage of input values runs along the y-axis. In general, most of the values have a biased exponent of around 128, meaning that most of the absolute values are close to 1.0. The QRAIN input, and especially the CLOUD and ICEFRAC inputs, contain many zero or subnormal values. The QRAIN input further contains quite a few small non-zero values. The AEROD input contains many extremely large absolute values. As mentioned, posits are particularly accurate for values close to 1.0, and the accuracy diminishes for values that are further away. Due to this, the QRAIN, ICEFRAC, CLOUD, and AEROD inputs are some of the only inputs that do not have 100.0% precise values (see Section 4.2).

When looking at each individual file's compressibility, QRAIN is the only one of the 14 tested inputs that, when encoded in posit

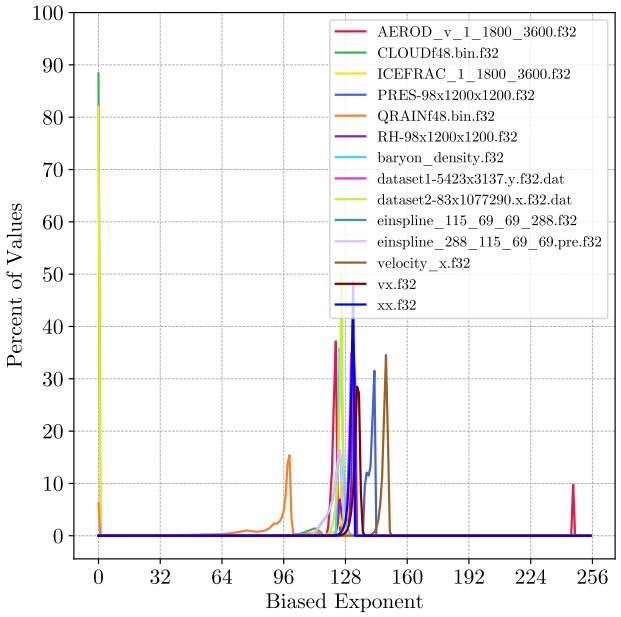

Figure 5: Percentage of IEEE floats with a given exponent for all 14 inputs

format, achieves higher compression ratios than when encoded in float format. Since this input has the lowest percentage of precise values, its increase in compressibility is likely a result of the loss in accuracy, potentially causing posit values to contain more zero bits (or one bits) in a row. On the other inputs, the variable-length exponents may be the cause of the reduced compression ratios.

Although the compression ratios are, in general, a little lower with posit data than with float data, we found that converting float to posit data does not affect the compressibility much. Dinechin et al. argue that 8- and 16-bit posits can be cast without error to 32-bit floats, and 32-bit posits can be cast without error to 64-bit floats [11], due to smaller-width posits having a larger dynamic range than the larger-width floats when using an appropriate value for *es*. Hence, storage requirements can be reduced by simply saving float data in a smaller-width posit. However, this approach will not work for certain float values that posits are unable to represent due to not having enough bits for the fraction after allocating the regime bits, thus truncating the float mantissa when converting to posit.

5.2 Per-File LC Pipeline

Figure 6 shows the compression provided by LC on both float and posit data when allowing each file its own unique compression pipeline. Out of the 236,196 possible pipelines, we took the highest compressing pipeline for each file and then computed the geometric mean over the resulting compression ratios, as opposed to using a single pipeline as was done in the previous subsection. When using a distinct pipeline on each file, LC delivers a 4.92% and 6.06% increase in compression ratio on float and posit data respectively, providing compression on par with XZ. However, XZ only runs

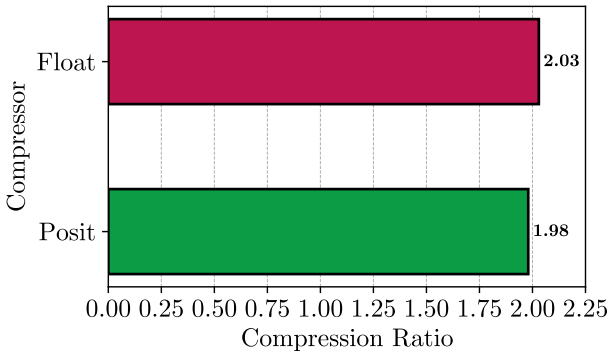

Figure 6: Geometric-mean compression ratios with a unique per-file LC pipeline

on the CPU, while LC compressors run on both the CPU and GPU, producing bit-for-bit the same result on both types of devices.

6 Conclusion

This paper presents the first study on the compressibility of posit data. We evaluate 5 general-purpose lossless compressors, and several new compressors generated by our LC framework, on 14 single-precision inputs from 7 datasets in the SDRBench suite. We create copies of these inputs and encode them in posit format to compare how well the compressors perform on the two formats.

We find that the XZ compressor provides the highest compression ratios on both formats. When allowing a customized LC pipeline for each input, LC yields ratios on par with XZ. Furthermore, the compression ratios on posit data are generally very close to and only slightly lower than on float data. Hence, storing scientific data in posit format is a viable option for applications that require the accuracy benefits of posits.

In future work, other metrics such as compression/decompression throughput and the compressors' memory overhead could be studied. Moreover, our work could be extended to cover other values for *es* and double-precision data as well. Once lossless and lossy special-purpose compressors for posits have been developed, these new compressors should be compared to special-purpose compressors designed for IEEE formatted data.

Acknowledgments

This work has been supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Research (ASCR), under contract DE-SC0022223.

References

[1] [n. d.]. Gzip: The GNU data compression utility. https://www.gnu.org/software/gzip/. Accessed: 2025-8-12.
[2] [n. d.]. LZ4 - Extremely fast compression. https://github.com/lz4/lz4. Accessed: 2025-8-12.
[3] [n. d.]. XZ Utils. https://tukaani.org/xz/. Accessed: 2025-8-12.
[4] [n. d.]. Zstandard. https://github.com/facebook/zstd. Accessed: 2025-8-12.
[5] Ihsen Alouani, Anouar Ben Khalifa, Farhad Merchant, and Rainer Leupers. 2021. An investigation on inherent robustness of posit data representation. In *2021 34th International Conference on VLSI Design and 2021 20th International Conference on Embedded Systems (VLSID)*. IEEE, 276–281.

[6] Noushin Azami, Alex Fallin, and Martin Burtscher. 2025. Efficient Lossless Compression of Scientific Floating-Point Data on CPUs and GPUs. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 395–409. doi:10.1145/3669940.3707280

[7] Ian Bird. 2011. Computing for the large hadron collider. *Annual Review of Nuclear and Particle Science* 61, 1 (2011), 99–118.

[8] Zachariah Carmichael, Hamed F Langroudi, Char Khazanov, Jeffrey Lillie, John L Gustafson, and Dhireesha Kudithipudi. 2019. Deep positron: A deep neural network using the posit number system. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1421–1426.

[9] Steven WD Chien, Ivy B Peng, and Stefano Markidis. 2019. Posit NPB: Assessing the precision improvement in HPC scientific applications. In *International Conference on Parallel Processing and Applied Mathematics*. Springer, 301–310.

[10] Stefan Dan Ciocirlan, Dumitrel Loghin, Lavanya Ramapantulu, Nicolae Ţăpuş, and Yong Meng Teo. 2021. The accuracy and efficiency of posit arithmetic. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*. IEEE, 83–87.

[11] Florent De Dinechin, Luc Forget, Jean-Michel Muller, and Yohann Uguen. 2019. Posits: the good, the bad and the ugly. In *Proceedings of the Conference for Next Generation Arithmetic 2019*. 1–10.

[12] Anwar A Esmaeel, Sa'ed Abed, Bassam J Mohd, and Abbas A Fairouz. 2022. Posit vs. floating point in implementing IIR notch filter by enhancing radix-4 modified booth multiplier. *Electronics* 11, 1 (2022), 163.

[13] Burtscher et al. 2025. LC-framework. https://github.com/burtscher/LC-framework/. Accessed: 2025-8-10.

[14] Alex Fallin, Noushin Azami, Sheng Di, Franck Cappello, and Martin Burtscher. 2025. Fast and Effective Lossy Compression on GPUs and CPUs with Guaranteed Error Bounds. In *2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 874–887.

[15] Alex Fallin, Noushin Azami, Sheng Di, Franck Cappello, and Martin Burtscher. 2025. Fast and Effective Lossy Compression on GPUs and CPUs with Guaranteed Error Bounds. In *2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 874–887. doi:10.1109/IPDPS64566.2025.00083

[16] T Fernandez-Hart, James C Knight, and Tatiana Kalganova. 2024. Posit and floating-point based Izhikevich neuron: A Comparison of arithmetic. *Neurocomputing* 597 (2024), 127903.

[17] Philip J Fleming and John J Wallace. 1986. How not to lie with statistics: the correct way to summarize benchmark results. *Commun. ACM* 29, 3 (1986), 218–221.

[18] John L Gustafson. 2017. *The end of error: Unum computing*. Chapman and Hall/CRC.

[19] John L Gustafson and Isaac T Yonemoto. 2017. Beating floating point at its own game: Posit arithmetic. *Supercomputing frontiers and innovations* 4, 2 (2017), 71–86.

[20] Salman Habib, Vitali Morozov, Nicholas Frontiere, Hal Finkel, Adrian Pope, and Katrin Heitmann. 2013. HACC: Extreme scaling and performance across diverse architectures. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–10.

[21] Junjie Hou, Yongxin Zhu, Sen Du, and Shijin Song. 2019. Enhancing accuracy and dynamic range of scientific data analytics by implementing posit arithmetic on FPGA. *Journal of Signal Processing Systems* 91, 10 (2019), 1137–1148.

[22] Milan Klöwer, Peter D Düben, and Tim N Palmer. 2019. Posits as an alternative to floats for weather and climate models. In *Proceedings of the conference for next generation arithmetic 2019*. 1–8.

[23] Seyed Hamed Fatemi Langroudi, Tej Pandit, and Dhireesha Kudithipudi. 2018. Deep learning inference on embedded devices: Fixed-point vs posit. In *2018 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*. IEEE, 19–23.

[24] Siew Hoon Leong and John L Gustafson. 2023. Lossless FFTs using posit arithmetic. In *Conference on Next Generation Arithmetic*. Springer, 1–18.

[25] Peter Lindstrom. 2022. MULTIPOSITS: Universal coding of R n. In *Conference on Next Generation Arithmetic*. Springer, 66–83.

[26] Jinming Lu, Chao Fang, Mingyang Xu, Jun Lin, and Zhongfeng Wang. 2020. Evaluations on deep neural networks training using posit number system. *IEEE Trans. Comput.* 70, 2 (2020), 174–187.

[27] Naohito Nakasato, Yuki Murakami, Fumiya Kono, and Maho Nakata. 2024. Evaluation of posit arithmetic with accelerators. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. 62–72.

[28] Aleksandr Yu Romanov, Alexander L Stempkovsky, Ilia V Lariushkin, Georgy E Novoselov, Roman A Solovyev, Vladimir A Starykh, Irina I Romanova, Dmitry V Telpukhov, and Ilya A Mkrtchan. 2021. Analysis of posit and bfloat arithmetic of real numbers for machine learning. *IEEE Access* 9 (2021), 82318–82324.

[29] Emanuele Ruffaldi and Federico Rossi. 2025. cppposit. https://github.com/federicoops/cppposit. Accessed: 2025-8-10.

[30] Julian Seward. 1996. Bzip2 and libbzip2. http://www.bzip.org.

[31] Nathan Whitehead and Alex Fit-Florea. 2011. Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs. *rn (A+ B)* 21, 1 (2011), 18749–19424.

[32] Xinyu Wu, Yaobin Wang, Tianyi Zhao, Jiawei Qin, Zhu Liang, and Jie Fu. 2025. PVU: Design and Implementation of a Posit Vector Arithmetic Unit (PVU) for Enhanced Floating-Point Computing in Edge and AI Applications. *arXiv preprint arXiv:2503.01313* (2025).

[33] Kai Zhao, Sheng Di, Xin Lian, Sihuan Li, Dingwen Tao, Julie Bessac, Zizhong Chen, and Franck Cappello. 2020. SDRBench: Scientific data reduction benchmark for lossy compressors. In *2020 IEEE international conference on big data (Big Data)*. IEEE, 2716–2724.