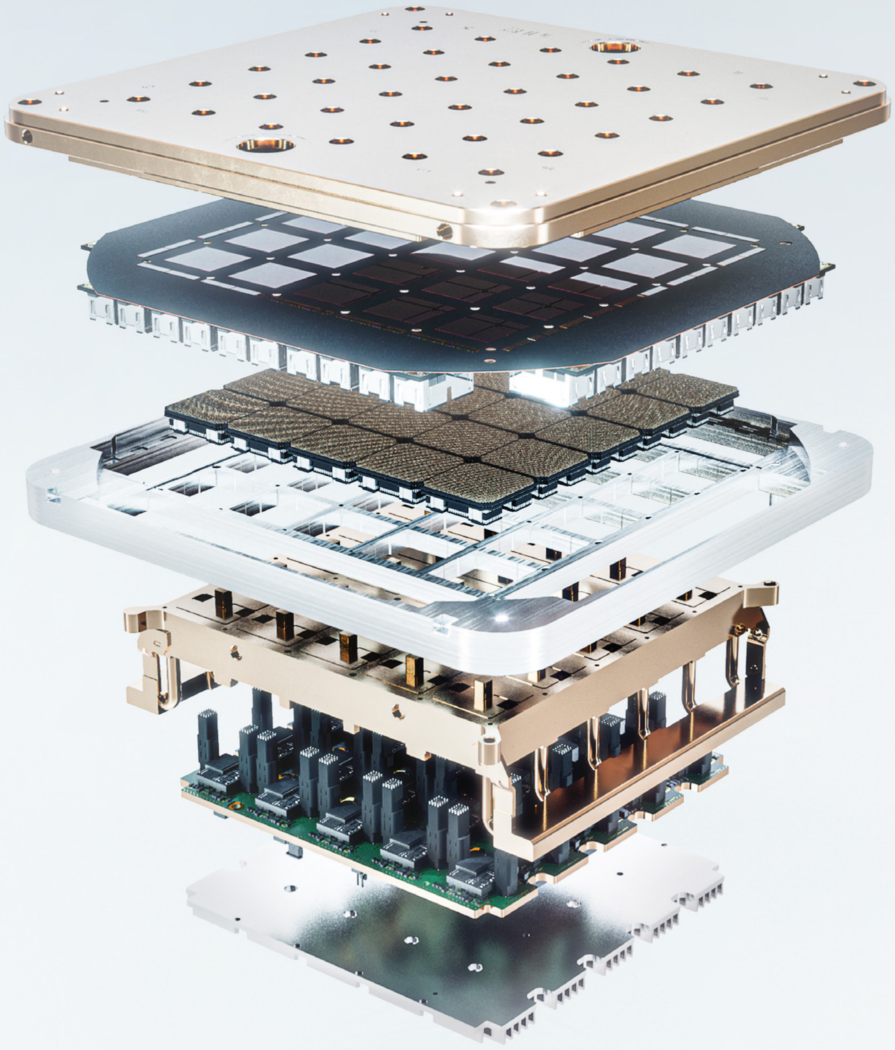
Tesla Dojo Technology

A Guide to Tesla's Configurable Floating Point Formats & Arithmetic



Tesla Configurable Float8 (CFloat8) & Float16 (CFloat16) Formats

Abstract
This standard specifies Tesla arithmetic formats and methods for the new 8-bit and 16-bit binary floating-point arithmetic in computer programming environments for deep learning neural network training. This standard also specifies exception conditions and the status flags thereof. An implementation of a floating-point system conforming to this standard may be realized entirely in software, entirely in hardware, or in any combination of software and hardware.

Keywords
Arithmetic, binary, computer, deep learning, neural networks, training, exponent, floating-point, format, NaN, Infinity, number, mantissa, subnormal, denormal, configurable exponent bias, range, precision, rounding mode, random number generator, stochastic rounding.

Motivation

The original IEEE 754 standard, which was published in 1985 specified formats and methods for floating-point arithmetic in computer systems—standard and extended functions with single (32-bit), double (64-bit) precision. The standard single and double precision formats are shown in Table 1 below.

Table 1: Floating Point Formats defined by the IEEE 754 Standard

Format	Sign bit?	No. of Mantissa bits	No. of Exponent bits	Exponent Bias Value
Single Precision (Float32)	Yes	1 + 23	8	127
Double Precision (Float64)	Yes	1 + 52	11	1023

The purpose of the standard was to provide a method for computation with floating-point numbers that will yield the same result whether the processing is done in hardware, software, or a combination of the two. The results of the computation will be identical, independent of implementation, given the same input data. Errors, and error conditions, in the mathematical processing will be reported in a consistent manner regardless of implementation.

The above formats have been widely adopted in computer systems, both hardware and software, for scientific, numeric, and various other computing. Subsequently, the revised IEEE754 standard in 2008 also included a half precision (16-bit), only as a storage format without specifying the arithmetic operations. However, Nvidia and Microsoft defined this datatype in the Cg language even earlier, in early 2002, and implemented it in silicon in the GeForce FX, released in late 2002.

The IEEE half precision format has been used not just for storage but even for performing arithmetic operations in various computer systems, especially for graphics and machine learning applications. This format is used in several computer graphics environments including MATLAB, OpenEXR, JPEG XR, GIMP, OpenGL, Cg, Direct3D, and D3DX. The advantage over single precision binary format is that it requires half the storage and bandwidth (at the expense of precision and range). Subsequently, the IEEE half precision format has been adopted in machine learning systems in the Nvidia AI processors, especially for training, due to the significantly increased memory storage and bandwidth requirements in such applications.

More recently, Google Brain, an artificial intelligence research group at Google, developed the Brain Floating Point, or BFloat16 (16-bit) format in their TPU architecture for their machine learning training systems. The BFloat16 format is utilized in Intel AI processors, such as Nervana NNP-L1000, Xeon processors (AVX-512 BF16 extensions), and Intel FPGAs, Google Cloud TPUs and TensorFlow. ARMv8.6-A, AMD ROCm, and CUDA also support the BFloat16 format. On these platforms, BFloat16 may also be used in mixed-precision arithmetic, where BFloat16 numbers may be operated on and expanded to wider data types, since it retains the dynamic range of the Float32 format. The BFloat16 format differs from the IEEE Float16 format in the number of bits provisioned for the mantissa and exponent bits. These two formats are shown Table 2 below.

Table 2: IEEE 16-bit Floating Point and Brain 16-bit Floating Point formats

Format	Sign bit?	No. of Mantissa bits	No. of Exponent bits	Exponent Bias Value
IEEE Half Precision(Float16)	Yes	1 + 10	5	15
Brain Floating Point (BFloat16)	Yes	1 + 7	8	127

As deep learning neural networks grow, the memory storage and bandwidth pressure continue to present challenges and create bottlenecks in many systems, even with the Float16 and BFloat16 storage in memory.

Tesla CFloat8 Formats

Tesla extended the reduced precision support further, and introduced the Configurable Float8 (CFloat8), an 8-bit floating point format, to further reduce the enormous pressure on memory storage and bandwidth in storing the weights, activations, and gradient values necessary for training the increasingly larger networks. Unlike the IEEE 754R standard, the purpose of this standard is mostly to standardize the formats and not necessarily to provide for portability of code to guarantee identical numerical result across all platforms.

The IEEE Float16 and Bfloat16 formats described above have a fixed number of bits allocated to the mantissa and exponent fields and have a fixed exponent bias. However, eight bits can only accommodate a small number of mantissa and exponent bits, so some configurability is required to ensure high accuracy and convergence of the training models.

One key property enabling this configurability is the fact that different parameters, namely weights, gradients and activations, have different precision and dynamic range requirements to achieve high training accuracy and convergence. The configurability enables different allocations of the number of exponent and mantissa bits, depending on the parameter being represented. Moreover, the numeric range these parameters span is also very different. Weights and gradients typically have much smaller numerical values compared to activations. The latter property allows meeting the dynamic range requirements of the various parameters using a configurable bias, instead of increasing the number of exponent bits.

The range of exponent values follows the principle of locality of space and time during the execution of a training network, and do not change frequently. Thus, only a small number of such exponent biases are used in any given execution step, and the appropriate bias values can be learnt during the training.

The two CFloat8 formats with the fully configurable bias are shown in Table 3 below.

Table 3: Tesla configurable 8-bit Floating Point formats

Format	Sign bit?	No. of Mantissa bits	No. of Exponent bits	Exponent Bias Value
CFloat8_1_4_3	Yes	1 + 3	4	Unsigned 6-bit integer
CFloat8_1_5_2	Yes	1 + 2	5	Unsigned 6-bit integer

Normalized numbers, Subnormal (denormal) numbers and Zeros are supported in both CFloat8_1_4_3 and CFloat8_1_5_2 formats. Due to the limited number of representable exponent values, Infinity and NaN encodings are not supported. So, the maximum exponent value is not reserved for encoding NaN and Infinity and just used to represent normalized floating-point numbers. Any Infinity or NaN operand, or an overflow in an arithmetic operation will clamp the result to the maximum representable number in each of these formats.

Numerical value of Zero is represented by an encoding with all zero Exponent and all zero Mantissa. Encodings with all zero Exponent and non-zero Mantissa represent denormal numbers.

The numerical value of a normalized floating point number is $(-1)^{\text{sign}} \times 2^{\text{exponent}-\text{bias}} \times 1.\text{mantissa}$ while the numerical value of a denormal floating point number is $(-1)^{\text{sign}} \times 2^{-\text{bias}} \times 0.\text{mantissa}$.

With the configurable exponent bias set to the minimum possible value ($000000_2 = 0$), the numerical values that can be represented in the normalized CFloat_1_4_3 format are shown below.

$E_{min} = 0001_2 - 000000_2 = 1$; $E_{max} = 1111_2 - 000000_2 = 15$

The range of numerical values thus represented is
+/- [$1.000_2 \times 2^1$, $1.111_2 \times 2^{15}$].

Similarly, with the exponent bias set to the maximum possible value ($111111_2 = 63$),

$E_{min} = 0001_2 - 111111_2 = -62$; $E_{max} = 1111_2 - 111111_2 = -48$

The range of numerical values thus represented is
+/- [$1.000_2 \times 2^{-62}$, $1.111_2 \times 2^{-48}$]

For normalized floating point numbers in the CFloat8_1_4_3 format, the numerical values for exponent bias values of 0, 1, 2, 3,..., 62, 63 are shown in Table 4 below. The entire exponent range [-62, 15] can be spanned by this format by reconfiguring the exponent bias appropriately. Please note that the exponent range with a 4-bit exponent with a fixed bias only spans 15 consecutive exponent values. For example, for bias = 31, only the exponent range [-30, -16] can be spanned.

Table 4: Range of numerical values of normalized floating-point numbers in CFloat8_1_4_3 format for various exponent bias values

Exponent Bias	Range of Numerical Values
0	+/-[1.000×2^1, 1.111×2^{15}]
1	+/-[1.000×2^0, 1.111×2^{14}]
2	+/-[1.000×2^{-1}, 1.111×2^{13}]
3	+/-[1.000×2^{-2}, 1.111×2^{12}]
...	
31	+/-[1.000×2^{-30}, 1.111×2^{-16}]
...	
60	+/-[1.000×2^{-59}, 1.111×2^{-45}]
61	+/-[1.000×2^{-60}, 1.111×2^{-46}]
62	+/-[1.000×2^{-61}, 1.111×2^{-47}]
63	+/-[1.000×2^{-62}, 1.111×2^{-48}]

Similarly, for normalized floating point numbers in the CFloat8_1_5_2 format, the numerical values for exponent bias values of 0, 1, 2, 3,..., 62, 63 are shown in Table 5 below. Thus, the entire exponent range [-62, 31] can be spanned by this format by reconfiguring the exponent bias appropriately. Please note that the exponent range with a 5-bit exponent with a fixed bias only spans 31 consecutive exponent values. For example, for bias = 31, only the exponent range [-30, 0] can be spanned.

Table 5: Range of numerical values of normalized floating-point numbers in CFloat8_1_5_2 format for various exponent bias values

Exponent Bias	Range of Numerical Values
0	+/-[1.00 x 2^{1}, 1.11 x 2^{31}]
1	+/-[1.00 x 2^{0}, 1.11 x 2^{30}]
2	+/-[1.00 x 2^{-1}, 1.11 x 2^{29}]
3	+/-[1.00 x 2^{-2}, 1.11 x 2^{28}]
...	
31	+/-[1.00 x 2^{-30}, 1.11 x 2^{0}]
...	
60	+/-[1.00 x 2^{-59}, 1.11 x 2^{-29}]
61	+/-[1.00 x 2^{-60}, 1.11 x 2^{-30}]
62	+/-[1.00 x 2^{-61}, 1.11 x 2^{-31}]
63	+/-[1.00 x 2^{-62}, 1.11 x 2^{-32}]

In the CFloat8_1_4_3 format, an Exponent = 0000_2 and Mantissa = 000_2 represents numerical value of Zero, while Exponent = 0000_2 and Mantissa = 001_2, 010_2, 011_2, 100_2, 101_2, 110_2, and 111_2 represent the denormal numbers. The corresponding numerical values are +/- {0.001_2, 0.010_2, 0.011_2, 0.100_2, 0.101_2, 0.110_2, 0.111_2} x 2^{-bias} respectively, where bias is the 6-bit exponent bias from 0 to 63. Similarly, in the CFloat8_1_5_2 format, an Exponent = 00000_2 and Mantissa = 00_2 represents numerical value of Zero, while Exponent = 00000_2 and Mantissa =01_2, 10_2, and 11_2 represent the denormal numbers. The corresponding numerical values are +/- {0.01_2, 0.10_2, 0.11_2} x 2^{-bias} respectively, where bias is the 6-bit exponent bias from 0 to 63.

Gradual underflow with denormal handling is supported in both CFloat8_1_4_3 and CFloat8_1_5_2 formats. These formats have limited exponent range, and the denormal number support helps increase the representable numeric range. The 6-bit bias is chosen as an unsigned integer to skew the range of representable values more on the smaller numeric values at the expense of larger numerical values, as the parameters in the deep learning neural networks are normalized within some [-N,N] bound where N is an integer, and thus tend to span very small numerical values.

Arithmetic Operations

When used as a storage format only, the two CFLoat8 formats, CFloat8_1_4_3 and CFloat8_1_5_2, shall support convert operations to and from the BFloat16 and IEEE Float32 formats. Two modes of rounding should be supported to convert from BFloat16 and IEEE Float32 formats to the two CFLoat8 formats—round-to-nearest and stochastic rounding. Stochastic rounding is implemented with a Random Number Generator (RNG). The arithmetic performed with stochastic rounding should be consistent and reproducible when the same seed is used in the RNG. Stochastic rounding enables probabilistic rounding with a uniform random number generator and enables statistical parameter updates, such as computing stochastic gradient descent (SGD) during back propagation in training. Very small values are accumulated into larger values, and such updates would not happen with IEEE rounding modes. Stochastic rounding is also useful in various training computes to add random noise to prevent regularization

The arithmetic operations that the CFloat8 formats should provide are implementation dependent. Typically, the CFloat8 formats are used in mixed precision arithmetic, where the operands stored in CFloat8 format in memory may be operated on and expanded to wider data types, such as Float32.

Tesla CFloat16 Formats

Two other formats are also specified for 16-bit floating point numbers, Signed Half Precision (SHP) and Unsigned Half Precision (UHP) as shown below. These formats are used to store parameters such as gradients, in cases where the precision of the CFloat8 formats may be too small to guarantee the convergence of training networks. The BFloat16 and Float16 formats both have some limitations for quantization. The BFloat16 format has sufficient range but the precision is very limited to quantize many parameters, such as gradients, requiring storing in FP32 format. The Float16 format has more precision, but the range is too small to quantize in many cases, also requiring storing in FP32 format. The SHP and UHP formats offer the precision of Float16 format while increasing the range of the Float16 format substantially with the configurable bias. The SHP and UHP formats obviate the storage requirement in FP32 format in most cases, reducing the memory storage pressure and improving the memory bandwidth further. The SHP and UHP formats are shown in Table 6 below.

Table 6: Tesla configurable 16-bit Floating Point formats

Format	Sign bit?	No. of Mantissa bits	No. of Exponent bits	Exponent Bias Value
Signed Half Precision (SHP)	Yes	1 + 10	5	Unsigned 6-bit integer
Unsigned half Precision (UHP)	No	1 + 10	6	31

Normalized numbers, Subnormal numbers (denormal) and Zeros are supported in the SHP format. Infinity and NaN encodings are not supported, and any overflow, or an Infinity or NaN operand during an arithmetic operation will clamp the result to the maximum representable number in the destination SHP format. Infinity and NaN are not supported in the SHP format since only a small number of exponents can be represented. So, the maximum exponent value is not reserved for the NaN and Infinity encodings, and just used to represent normalized floating-point numbers.

Normalized numbers and Zeros are supported in the UHP format. Denormal encodings are supported but denormal operands and results are flushed to zero. Infinity and NaN encodings are also supported in the UHP format as there are more exponent bits than the SHP format to allow these two encoding with the maximum exponent value. Infinity is encoded as with all ones Exponent and all zero Mantissa, and NaNs are encoded as with all ones Exponent and non-zero Mantissa. The NaN propagation and Infinity result generation for any arithmetic operation with destination in the UHP format follow the specifications in the IEEE 754R standard. In the UHP format, Infinity is encoded Exponent =111111_2 and Mantissa = 0000000000_2, while NaN is encoded as Exponent = 111111_2 and Mantissa ≠ 0000000000_2. Any operation with a NaN in the destination UHP format is encoded as a canonical NaN which is encoded as Exponent = 111111_2 and Mantissa = 1000000000_2.

When used as a storage format, the SHP and UHP formats shall support convert operations to and from the Float32 format. Two modes of rounding should be supported to convert from IEEE Float32 formats to the SHP and UHP formats—round-to-nearest and stochastic rounding. The arithmetic operations that the SHP and UHP formats should provide are implementation dependent. Typically, these formats are used in mixed-precision arithmetic, where the operands stored in SHP or UHP format in memory may be operated on and expanded to wider data types, such as Float32.

Exception Status Flags

The following exception status flags are supported in operations with CFloat8, SHP, and UHP operands and results: Invalid, Denormal, Overflow and Underflow. An arithmetic operation with any denormal operand will set the denormal exception flag, while an arithmetic operation with any NaN operand or no useful definable result, as specified by the IEEE 754R standard, will set the invalid exception flag. Any arithmetic operation with CFloat8, SHP, and UHP destination that overflows or underflows, will set the overflow and underflow exception flags respectively. The response to the setting of the exception flags is implementation dependent.