Scaling Laws for Superfloat: Shifting to Exponent-less Precision

Alosh Denny[1] Anandu Babu[1] Ameena Jamal[1] Abhinand Manoj[1]

[1]Department of Computer Science, School of Engineering, Cochin University of Science and Technology, Kochi, Kerala, India, 682022

ABSTRACT

We introduce SuperFloat (SF), a novel data type for efficient parameter-aware deep learning model optimization. SuperFloat leverages the observation that the majority of deep learning model parameters lie within a bounded range, often centered around zero. By reserving all bits for the significand and reducing the exponent range, SuperFloat enables a more compact representation without compromising model accuracy.

We present multiple variants of the SuperFloat data type that strike a balance between representational precision and computational efficiency. These customized data types allow for selective quantization of less critical model parameters, reducing memory footprint and computation costs without degrading performance.

Through extensive experiments on a variety of deep learning models, including large language models and computer vision architectures, we demonstrate the effectiveness of our SuperFloat-based approach. We show that a significant portion of model parameters can be represented using the more efficient SuperFloat formats with negligible impact on accuracy. Furthermore, we introduce hardware-aware techniques, such as sparse matrix multiplication and unity-based computations, to further optimize the deployment of SuperFloat-quantized models on resource-constrained hardware.

Our work provides a principled and flexible framework for parameter-aware deep learning model optimization, paving the way for more efficient deployment of advanced models on edge devices and embedded systems.

INTRODUCTION

Deep learning models have achieved remarkable success across a wide range of domains, from computer vision and natural language processing to robotics and healthcare. However, the increasing complexity and scale of these models has led to significant challenges in terms of memory usage, computational costs, and energy consumption. Modern deep neural networks and large language models often contain billions of parameters, which are typically stored and processed using the standard 32-bit floating-point (FP32) representation.

While existing quantization techniques, such as 8-bit integer (INT8), 16-bit floating-point (FP16) and 32-bit floating-point (FP32), have shown promise in reducing the memory and computational requirements of deep learning models, they often come at the cost of decreased model performance, especially for large and complex architectures. This trade-off between model efficiency and accuracy has become a critical bottleneck in the deployment of advanced deep learning models on resource-constrained hardware, such as edge devices and mobile platforms.

In this work, we introduce a novel data type called SuperFloat (SF) that addresses the limitations of existing quantization methods. The key insight behind SuperFloat is the observation that the majority of deep learning model parameters lie within a bounded range, often centered around zero. By leveraging this property, we can design customized data representations that prioritize the efficient encoding of these common parameter values, without sacrificing model performance. Specifically, we present three variants of the SuperFloat data type:

1. **SuperFloat16 (SF16)**: A 16-bit data type that allocates all 15 bits for the significand, with 1 bit reserved for the sign. This allows for a more precise representation of parameters within the [-1, 1] range, compared to traditional 16-bit floating-point formats.

2. **SuperFloat8 (SF8)**: An 8-bit data type that follows a similar principle, using 7 bits for the significand and 1 bit for the sign. While offering a more compact representation, SF8 is suitable for a narrower range of parameter values.

3. **SuperFloat4 (SF4)**: A 4-bit data type that allocates 3 bits for the significand and 1 bit for the sign. It is suitable for compute on low-end edge devices.

Through extensive experiments on a diverse set of deep learning models, including large language models (Llama, Gemma, Qwen) and computer vision architectures (ViT, SWIN, YOLO), we demonstrate the effectiveness of our SuperFloat-based approach. Our results show that a substantial portion of model parameters can be represented using the more efficient SuperFloat formats, with negligible impact on accuracy.

The contributions of this work are as follows:

1. We introduce the SuperFloat data type, a customized number representation that leverages the bounded range of deep learning model parameters for more efficient encoding.
2. We present two variants of SuperFloat, SF16 and SF8, that strike a balance between representational precision and computational efficiency.
3. We develop hardware-aware techniques, such as sparse matrix multiplication and unity-based computations, to further optimize the deployment of SuperFloat-quantized models on resource-constrained hardware.
4. We extensively evaluate the effectiveness of our SuperFloat-based approach on a wide range of deep learning models, showcasing its ability to significantly reduce memory and computational requirements without compromising model performance.

The remainder of this paper is organized as follows: Section 2 provides background on existing quantization techniques and their limitations. Section 3 introduces the SuperFloat data type and three sample variants, SF16, SF8 and SF4. Section 4 presents our hardware-aware optimization techniques for SuperFloat-quantized models. Section 5 details our experimental setup and results. Section 6 discusses the implications and limitations of our work. Finally, Section 7 concludes the paper and outlines future research directions.

RELATED WORKS

(to be filled)

SUPERFLOAT METHODOLOGY

SuperFloat (SF; SFx) is a customizable data type that is void of any exponent bits and solely relies on a sign bit and mantissa bits. In SFx, x represents the total number of bits used to represent the value. We reserve 1 bit for the sign and the remaining (x-1) bits for the significand. This allows us to represent parameter values in the range *[-2^(x-1), 2^(x-1) - 1]*. As we will see in the three SF variants: SF16, SF8 and SF4, each variant is only backward compatible. That is, any floating point type that is to be casted to SuperFloat must have mantissa bitwidth >= x.

Mathematically, the value of an SFx number can be expressed as:

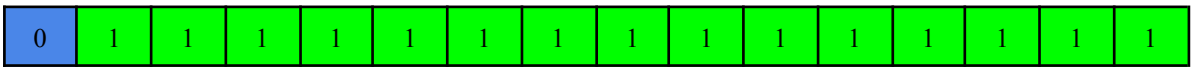
$$SFx = (-1)^{\text{sign}} \cdot 2^0 \cdot \left(1 + \sum_{i=1}^{x-1} \text{bit}_i \cdot 2^{-i}\right)$$

Where:

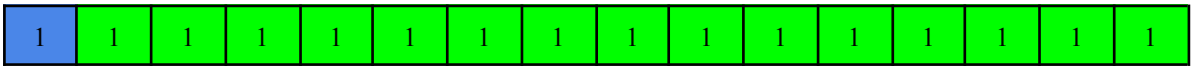
- *sign* is the sign bit (0 for positive, 1 for negative)
- *bit_i* is the i-th bit of the significand, starting from the least significant bit

SuperFloat16

SuperFloat16 (or SF16 / sf16) is a significand-only mixed-point precision data type that reserves all its 15 bits for the significand and one bit for sign. It simplifies the logic by fixing the location of the radix point, but still generates SOTA results in accuracy. It offers even better precision than the mantissa portion of TensorFloat32, Float16 and BFloat16. The limitation here is that it can only be applied to floating points lying in the range [-1,1].



Decimal: 0.999969482421875 (16 scale and 16 precision)

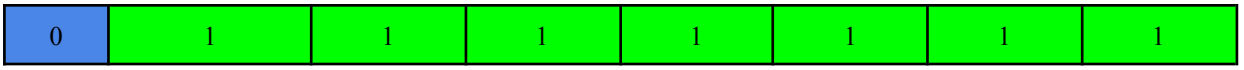


Decimal: -0.999969482421875 (16 scale and 16 precision)

Thus the maximum possible decimal value lies in the range [-0.999969482421875, 0.999969482421875].

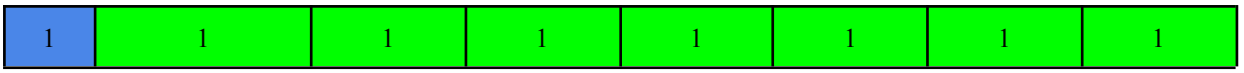
SuperFloat8

SuperFloat8(or SF8/ sf8) is a smaller variant of SuperFloat16 that reserves all its 7 bits for the significand and one bit for sign. It offers better precision than the mantissa portion of BFloat16 and Float 8.



Decimal: 0.9921875 (8 scale and 8 precision)

Anything above 0.9928175 => 0.9921875



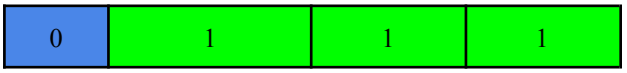
Decimal: -0.9921875 (8 scale and 8 precision)

Anything below -0.9921875 => -0.9921875

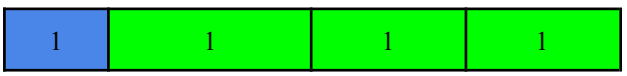
Thus the maximum possible decimal value lies in the range [-0.9921875, 0.9921875].

SuperFloat4

Similar to its siblings, SF4 is at the lower end of the SF quantization spectrum. It reserves 3 bits for mantissa and one bit for sign.



Decimal: 0.875



Decimal: -0.875

Casting Table

A casting table maps the theoretical floating point type of an SFx parameter to its corresponding real-world hardware type, such as FP32, FP16, BF16, etc.

SF[x]	Real world data type
SF16	FP32
SF15	FP32
SF14	FP32
SF13	FP32
SF12	FP32
SF11	FP16
SF10	FP16
SF9	FP16
SF8	FP16 / BF16
SF7	FP16 / BF16
SF6	FP16 / BF16
SF5	FP16 / BF16
SF4	BF16 / FP8 (e4m3)

Vanilla SFx Quantization Algorithm

Input: A neural network model (LLM, CNN) with parameters in any precision (FP64, FP32, TF32, FP16, BF16, INT8, FP8, INT4, FP4, INT2).

Output: SFx-quantized parameter model.

Step 0 (optional): Select typecasting

From the casting table, select the hardware floating point type after casting to superfloat.

Step 1: Layer-wise Parameter Range Scanning

- Scan each layer's parameters.
- Identify layers where all parameters are within the range of *[-1, 1]*. These are non-critical parameters

Step 2: Precision Conversion for Layers within Range

For each layer satisfying Step 1:

- Step 2.1: Convert the parameters from their native precision (FP64, FP32, etc.) to SFx:
    - Clamp the mantissa of the parameters within *(x-1)* bits, with 1 bit reserved for the sign bit.
    - This ensures that the parameters are represented in the range *[-(x-1), +(x-1)]*.
- Step 2.2: Assign the most significant bit (MSB) as the sign bit.
- Step 2.3: Append the quantized weights to the new model.

Step 3: Retain Native Precision for Out-of-Range Layers

For layers that do not satisfy the range condition, append the parameters in their original precision to the new quantized model.

Step 4 (Optional): Sparse Matrix Representation

Construct sparse matrices to optimize matrix multiplication operations and improve inference performance.

Step 5: Load Model in Automatic Mixed Precision (AMP) Mode

Load the model in AMP mode to further optimize the performance during inference.

Step 6: Run Inference

Perform inference with the SFx-quantized model, and compute the activations and logits.

Notes:

1. SFx Precision: SFx represents parameters with *(x-1)* bits for the mantissa, and 1 bit for the sign, effectively constraining the parameter range to *[-(x-1), +(x-1)]*.
2. The algorithm works for a variety of input precisions (FP64, FP32, TF32, FP16, BF16, INT8, FP8, INT4, FP4, INT2), and performs quantization based on the specified bit-width *x*.

SUPERFLOAT QUANTIZATION METHODS

SuperFloat offers three different methods for quantizing deep learning model parameters into any of the SuperFloat bit-width types (SF4 to SF16):

1. Vanilla SuperFloat Quantization (abbr. *van*):

- This is the basic implementation of the SuperFloat quantization algorithm.
- All parameters are cast to the target SFx data type after clamping the mantissa within the (x-1) bit range.
- **No additional calibration or retraining is involved.**
- This is the **fastest** implementation, but may result in higher perplexity (worse performance) compared to the other methods.
- The model can be operated in either Automatic Mixed Precision (AMP) mode or pure SFx mode.

2. Full Parameter SuperFloat Quantization (abbr. *fpm*)*:*

- In this method, all parameters are cast to the target SFx data type after clamping.
- The entire model is then recalibrated using a smaller calibration dataset.
- This approach offers the lowest perplexity (best performance) but is the slowest to implement, as it requires the full model retraining.
- The model operates in pure SFx mode during inference.

3. Optimized SuperFloat Quantization (abbr. *opt*)*:*

- This is a middle ground between the Vanilla and Full Parameter methods.
- All parameters are cast to the target SFx data type after clamping.
- Only the critical parameters (e.g., those with the highest contribution to the model's performance) are recalibrated using a smaller dataset.
- This approach is around 50% faster than the Full Parameter method, while achieving perplexity results that are almost as good.
- The model operates in pure SFx mode during inference.

The Full Parameter and Optimized versions are simply modified versions of the Vanilla Superfloat algorithm, with the added step of **calibration**. Calibration refers to the retraining of a model's parameters on a smaller dataset.

Compare on:
1. Memory usage - theoretical (base, vanilla, fp, opt)

2. Perplexity (base, vanilla, fp, opt)

3. Time to inference (base, vanilla, fp, opt)

4. Time to train per epoch (fp, opt)

4. BLEU score (base, vanilla, fp, opt)

5. Loss over calibration epochs (fp, opt)

6. Perplexity performance of over different epochs (fp, opt)

(Note: If **Perplexity** = 1, your LLM is perfectly good at prediction. Higher values = worse prediction)

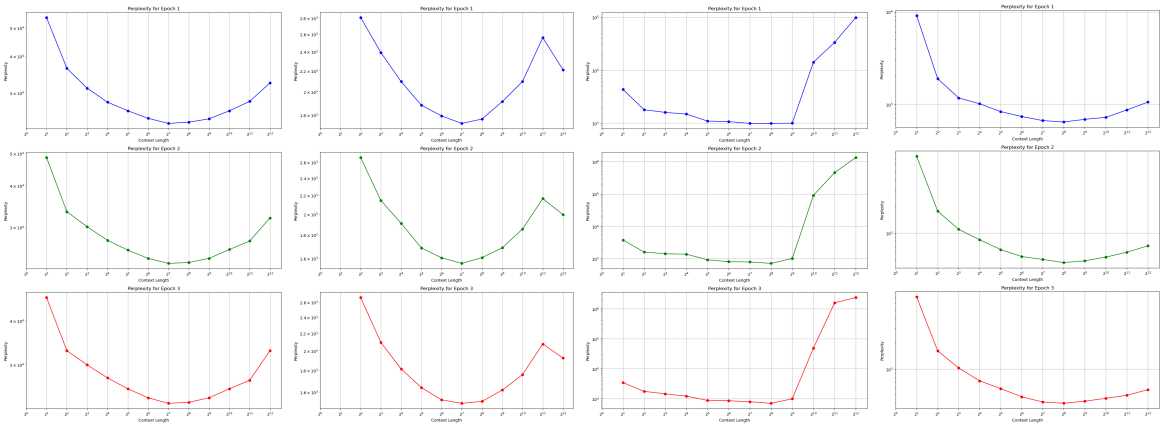
SCALING LAWS

1. Maximum Context Length Barrier

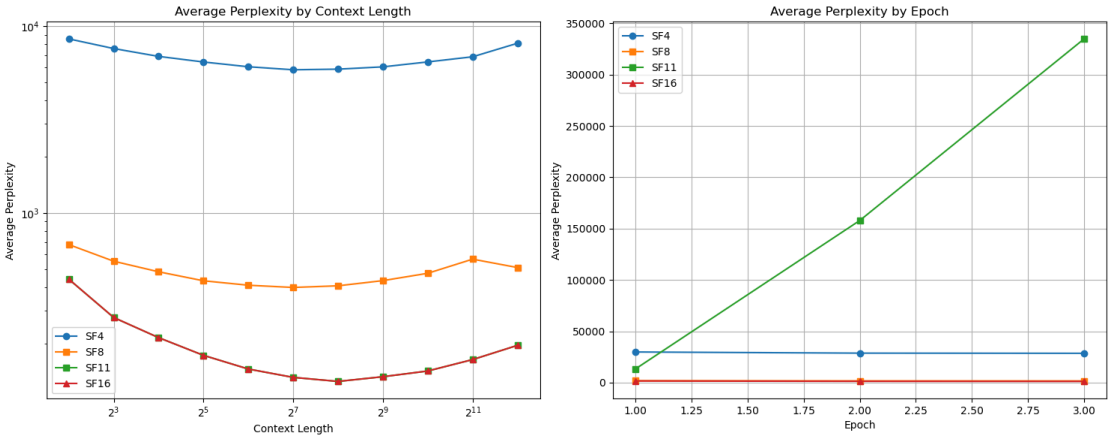
For a model with n parameters, a calibration dataset of maximum input length c, three-shot quantization fine-tuning, and Superfloat precision bit x (where $4 \leq x \leq 16$):

$P = f(n, c, 3, x)$

Lower P indicates better model understanding and calibration performance.



From left to right: SF4, SF8, SF11, SF16; three-shot quantized variants of the Qwen2-0.5B models across input lengths ranging from 2^1 to 2^{12}. In all cases, the lowest perplexity was achieved between an input length of 2^7 and 2^8.



Left: Average perplexity by context length; **Right**: Average perplexity by epoch; **SF16** and **SF11** hold the lowest perplexity levels for **three-shot** quantization.

2. Maximum Neuron Spread Factor

This scaling law uses the Lottery Ticket Hypothesis for WASQ quantization to stabilize activations:

1. Perform a forward pass using the original model and record the average magnitudes of activations across all layers.
2. Perform the same for the vanilla quantized model to observe how quantization impacts activation magnitudes.
3. Rank layers based on the difference in activation magnitudes between the original and quantized models.
4. Identify and cluster layers with significant deviations to address issues like exploding/vanishing activations.
5. Fine-tune or analyze these clusters to ensure stable activations and minimal performance degradation.

The law establishes that the maximum neuron spread (region targeted for fine-tuning/updating) is a function of:

Activation magnitude

Activation fracture (spread of how a weight affects neighboring weights during backpropagation)

GPTQ:

The GPTQ algorithm begins with a Cholesky decomposition of the Hessian inverse (a matrix that helps decide how to adjust the weights)

It then runs in loops, handling batches of columns at a time.

For each column in a batch, it quantizes the weights, calculates the error, and updates the weights in the block accordingly.

After processing the batch, it updates all remaining weights based on the block's errors.