Developing an efficient parameter-aware search space heuristic to optimize deep learning models via selective quantization

(with hardware-aware state expansion)

Alosh Denny Anandu Babu Abhinand D Manoj Ameena Jamal

Guide: Dr. Pramod Pavithran

Class Coordinator: Ms. Thasnim KM

CS-A

Abstract:

We introduce a search space heuristic aimed at addressing the computational needs of low-end hardware accelerators for training and inferring deep learning models. Our objectives include: a) developing an algorithm that effectively identifies parameters crucial for model performance and quantizes them without sacrificing key performance metrics like accuracy and perplexity; b) creating a new data type (ideally in floating point) that optimally balances the magnitude of the mantissa and the precision of the exponent, drawing inspiration from existing formats such as float32, float16, brainfloat16, float8, and int8, while also lowering computational costs; c) designing a hardware schematic and simulating it using Vivado. We refer to this approach as Weight-aware Selective Quantization (WASQ).

Problem:

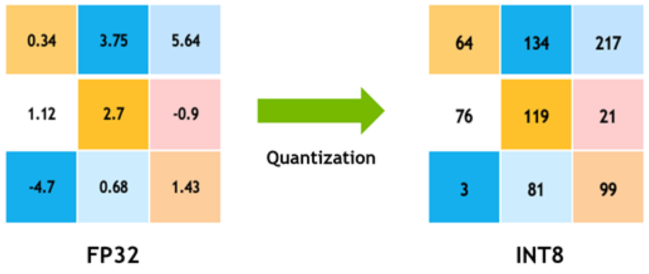
Modern deep models and large language models contain billions of parameters, which are stored in their native format - **float32 (fp32)**. This leads to several challenges:

1. Memory constraints: Storing billions of fp32 parameters requires significant memory, limiting deployment on memory-constrained devices.
2. Computational overhead: Operations on fp32 numbers are computationally expensive, especially on low-end hardware accelerators.
3. Energy consumption: Higher precision computations consume more energy, which is a concern for edge devices and mobile applications.

While existing quantization techniques (e.g., int8, float16) can address some of these issues, they often lead to a degradation in model performance, especially for large and complex models. This is a good place to start on tensor datatypes.

Existing Solutions:

Existing methods such as LLM.int8() offer cheaper compute costs by cutting down the precision from 32 bits to 8 bits via two quantization methods – symmetric and asymmetric quantization. At the smaller end of the spectrum, up to 7-billion parameter int8 models achieve comparable levels of perplexity as that of its fp32 counterparts. But scaling upwards, the authors noticed a drastic discharge in performance.



Another technique introduced was Tensorfloat32 (tf32). tf32 uses the same 10-bit mantissa as the half-precision (fp16) math, shown to have more than sufficient margin for the precision requirements of AI workloads. And tf32 adopts the same 8-bit exponent as fp32 so it can support the same numeric range.

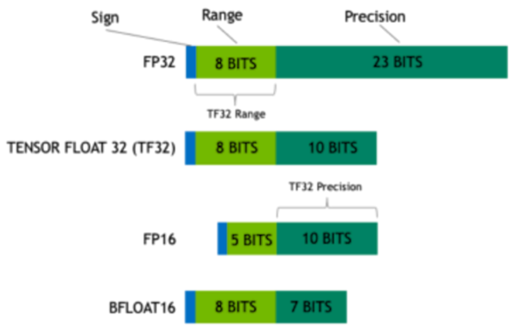


Image from Nvidia blog post

A survey of quantization methods can be found here

Ours:

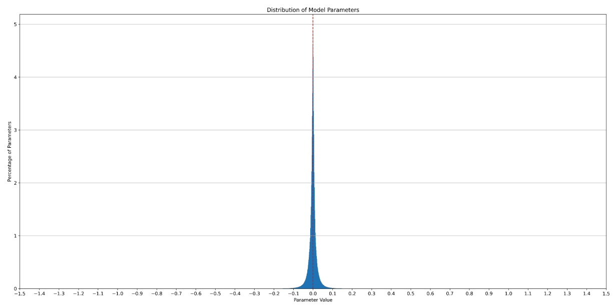
We propose a search space heuristic to selectively quantize parameters that significantly impact the model. This involves:

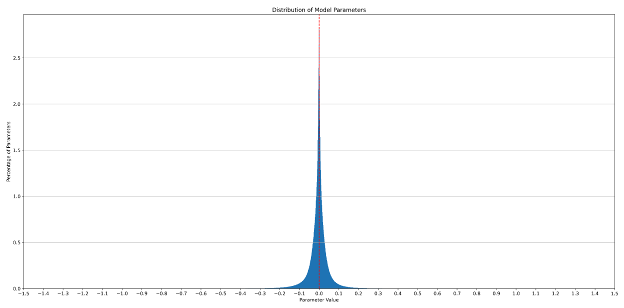
1. Identifying parameters that contribute highly to model performance – for example, categorizing parameters in terms of percentile of magnitude (99th, 95th, 90th, etc..) and quantizing only those parameters below a specified percentile threshold.
2. Sparse quantization – explained very well in this blog post. Research shows that a major chunk of computations can be skipped by forcing some weights to be zero, with little impact on the final accuracy.
3. **Dynamic Precision with SuperFloat16**: We introduce a novel datatype, SuperFloat16 (sf16), which allows for flexible bit allocation between mantissa and exponent.
4. n the parameters being processed. For instance, when multiplying parameters of different sf16 configurations (e.g., 1_4_11 and 1_9_6), the result can exceed the original bit allocation and be rounded to

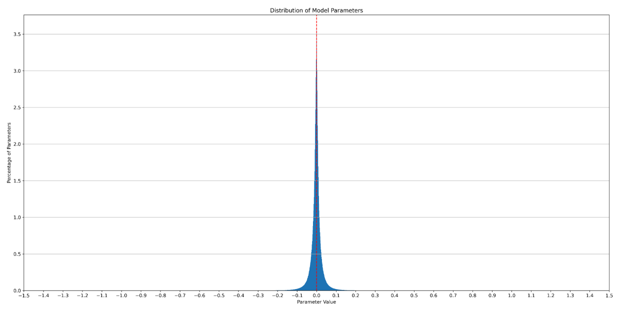
FP32 or FP16 as needed. This approach enables mixed precision training, optimizing both memory usage and computational efficiency while maintaining model accuracy.

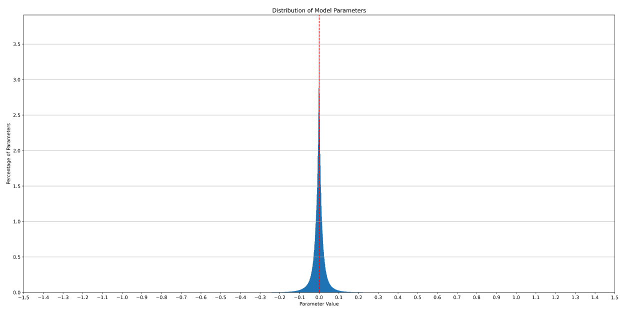
5. **Hardware Implementation using Verilog and Vivado**

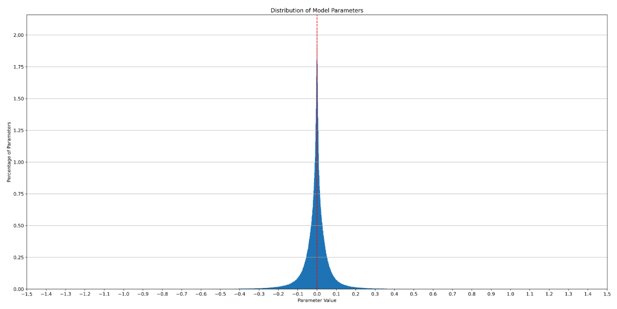
<u>STUDIES</u>

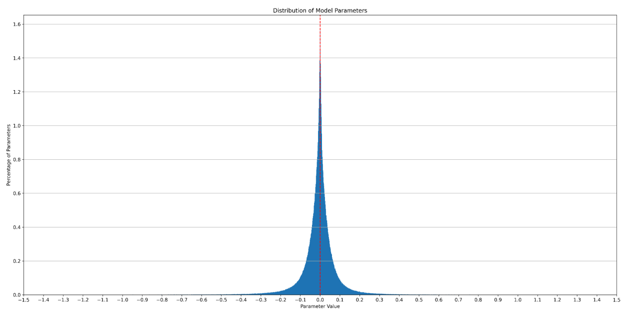












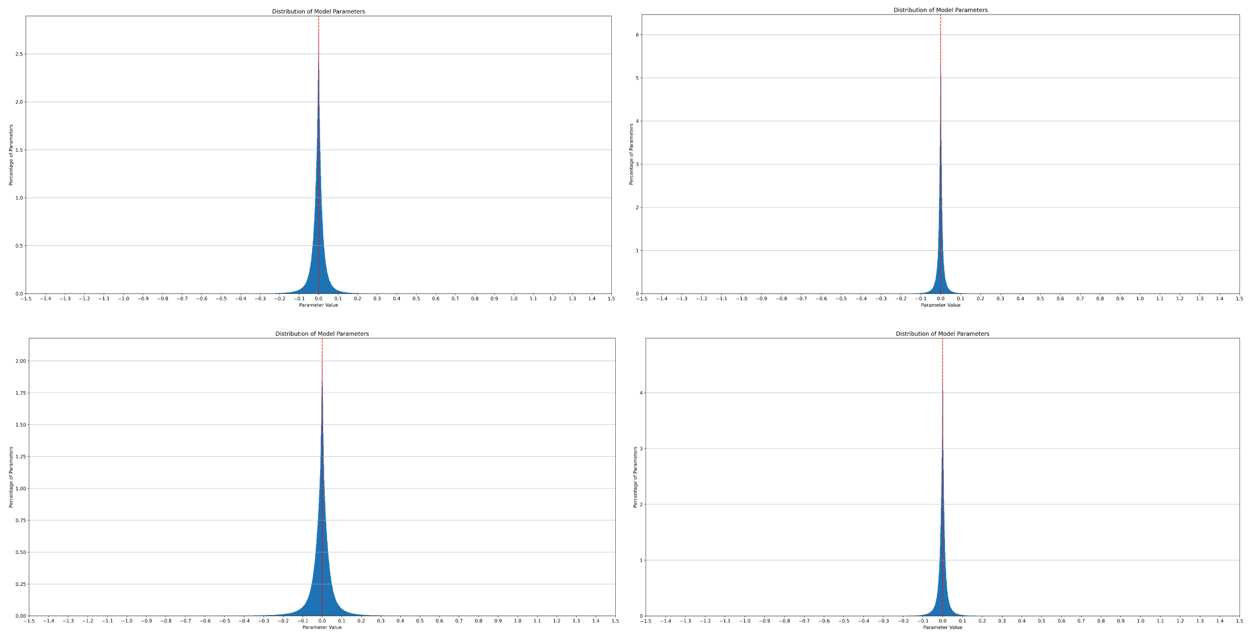
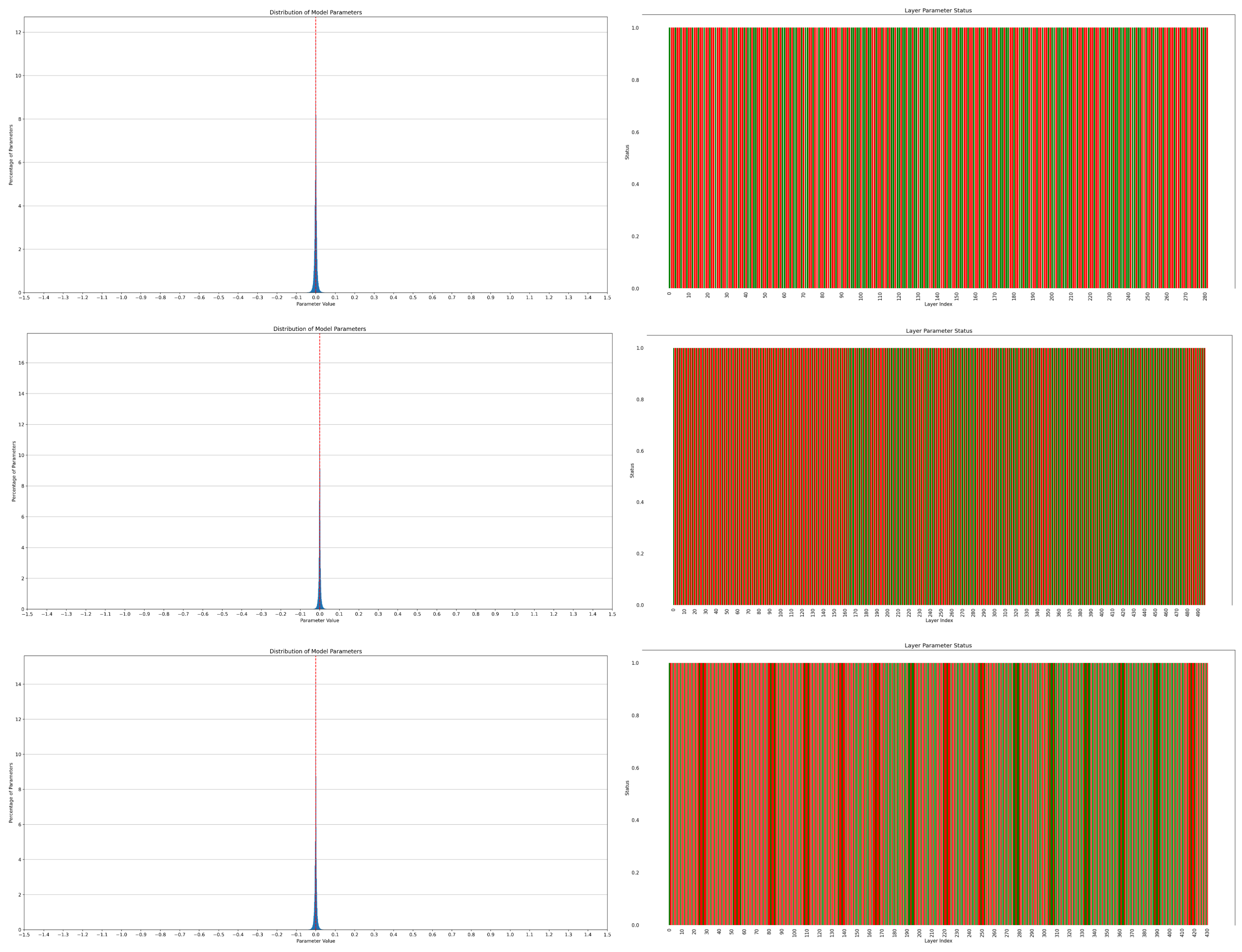


Fig 3. Model parameter distribution histograms for YOLOv5 models - L6, 5S, 5X6, 5M, 5N, 5N6, 5M6, 5L, 5S6, 5X



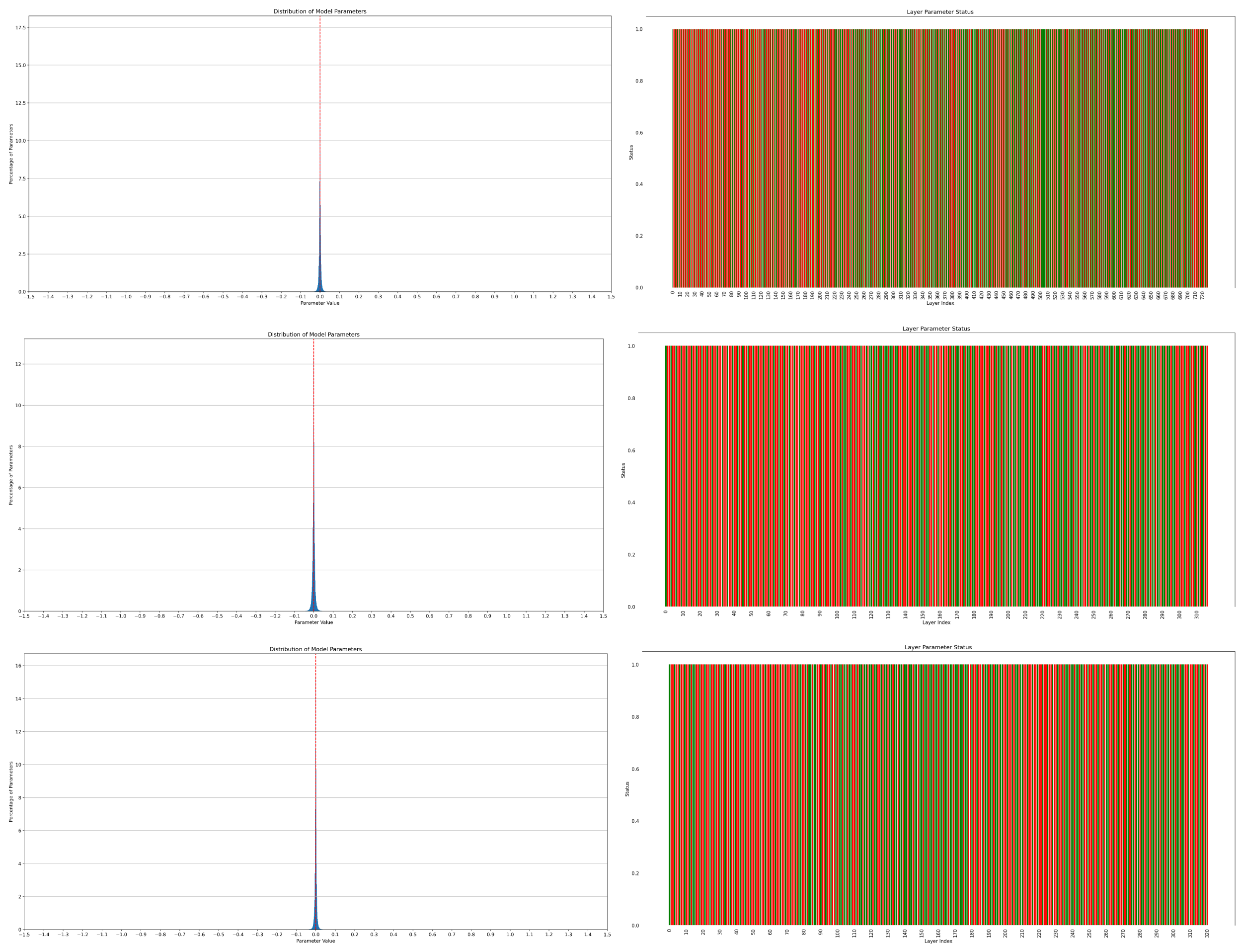
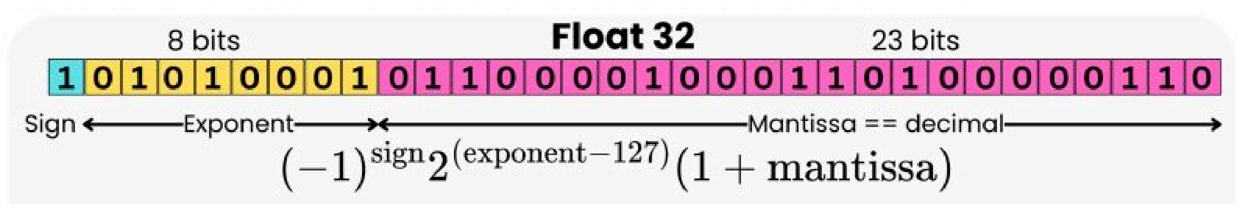


Fig 4. Model parameter distribution histograms for YOLOv7 models - Base, D6, E6, E6E, W6, X

Notice that ~99% of parameters in all cases lie between -1.0 and 1.0. All plots are quite similar, with a surprising spike around 0. This gives us an edge.

The weights between [-1, 1] do not require a whole number to represent it. For example, a weight **0.124351501464844** does not require any bits to represent its whole number part, that is 0. On the other hand, a weight such as **1.1243515501464844** requires bits to store the 1 before decimal point. This feature allows us to shave off the 8 exponent bits. The intuition lies as follows:



In the example, **0.124351501464844** can be converted to float32 and broken down as follows:

0	**01111011**	**11111101010110000000000**
sign	**biased exponent**	**significand/mantissa**

Applying the above formula,

exponent = 123

$(-1)^{sign} * 2^{(exponent-127)} = (-1)^{0} * 2^{(123-127)} \Rightarrow 1 * 2^{-4}$

Now, $2^{-1} * (1+mantissa) \Rightarrow 2^{-4} *$ **1.11111101010110000000000 => 0.00011111101010110000000000** *(shifted to right)* which translates back into **0.12435150146484375**; there is a little error but it's negligible (link).

What concerns us is the number of bits it takes to represent this fat nigga: **0.00011111101010110000000000**

~~Honestly I could not give a fuck about the last 8 or 9 bits lol~~

SuperFloat16 to the rescue

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].

Example:

0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Decimal: 0.999969482421875 (16 scale and 16 precision)

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

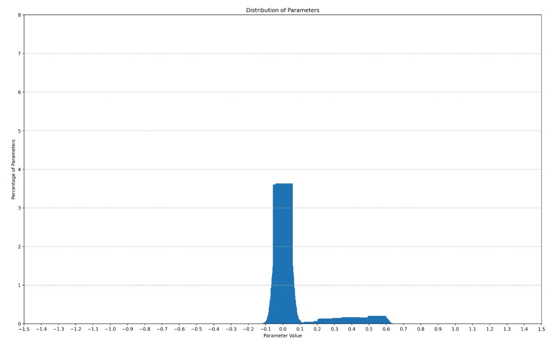
Decimal: -0.999969482421875 (16 scale and 16 precision)

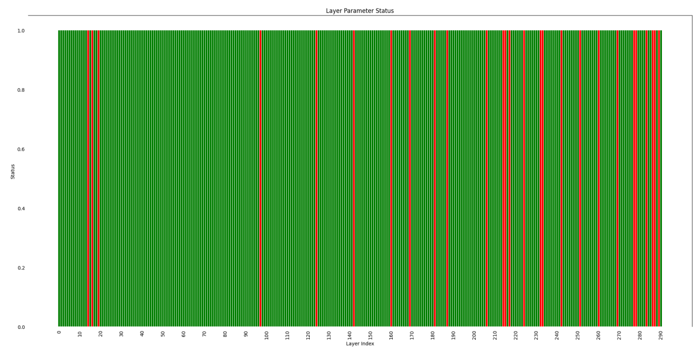
Thus the maximum possible decimal value lies in the range [-0.999969482421875, 0.999969482421875].

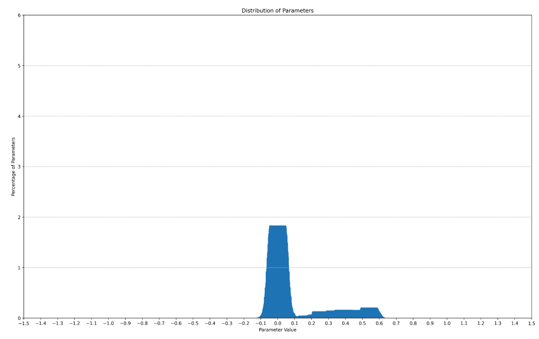
Vision Model	Percentage Outside Range (%)	Percentage Within Range (%)	Percentage Of Unity (%)	Percentage Of Sparse (%)
YOLOv7-d6	0.0290	99.9710	0.00	1.28
YOLOv7-w6	0.0331	99.9669	0.00	0.86
YOLOv7-e6	0.0327	99.9673	0.00	1.05
YOLOv7x	0.0334	99.9666	0.00	1.08
YOLOv7-e6e	0.0320	99.9680	0.00	1.41
YOLOv5n	0.4210	99.5790	0.00	0.11
YOLOv5s	0.1088	99.8912	0.00	0.13
YOLOv5m	0.0330	99.9670	0.00	0.17
YOLOv5l	0.0154	99.9846	0.00	0.22
YOLOv5x	0.0081	99.9919	0.00	0.31
YOLOv5n6	0.2834	99.7166	0.00	0.14

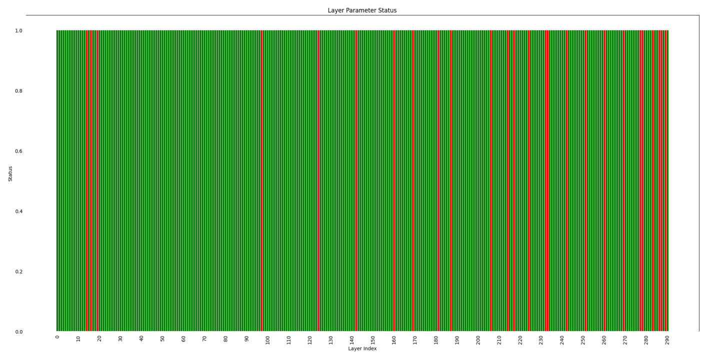
YOLOv5s6	0.0693	99.9307	0.00	0.18
YOLOv5m6	0.0204	99.9796	0.00	0.24
YOLOv5l6	0.0090	99.9910	0.00	0.30
YOLOv5x6	0.0059	99.9941	0.00	0.42

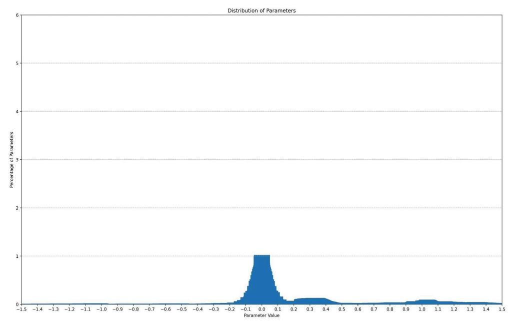
Table 2: Percentages of parameters lying within and outside the range [-0.999969482421875, 0.999969482421875]

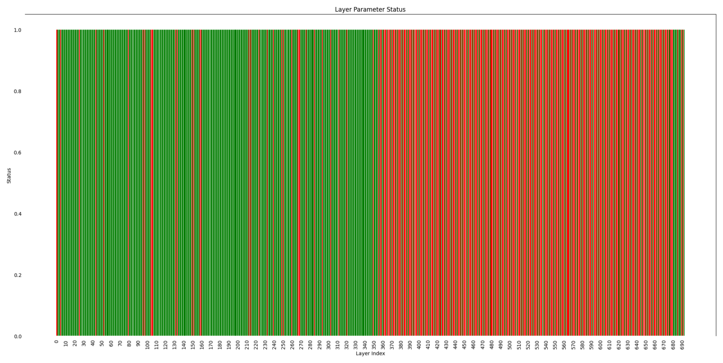












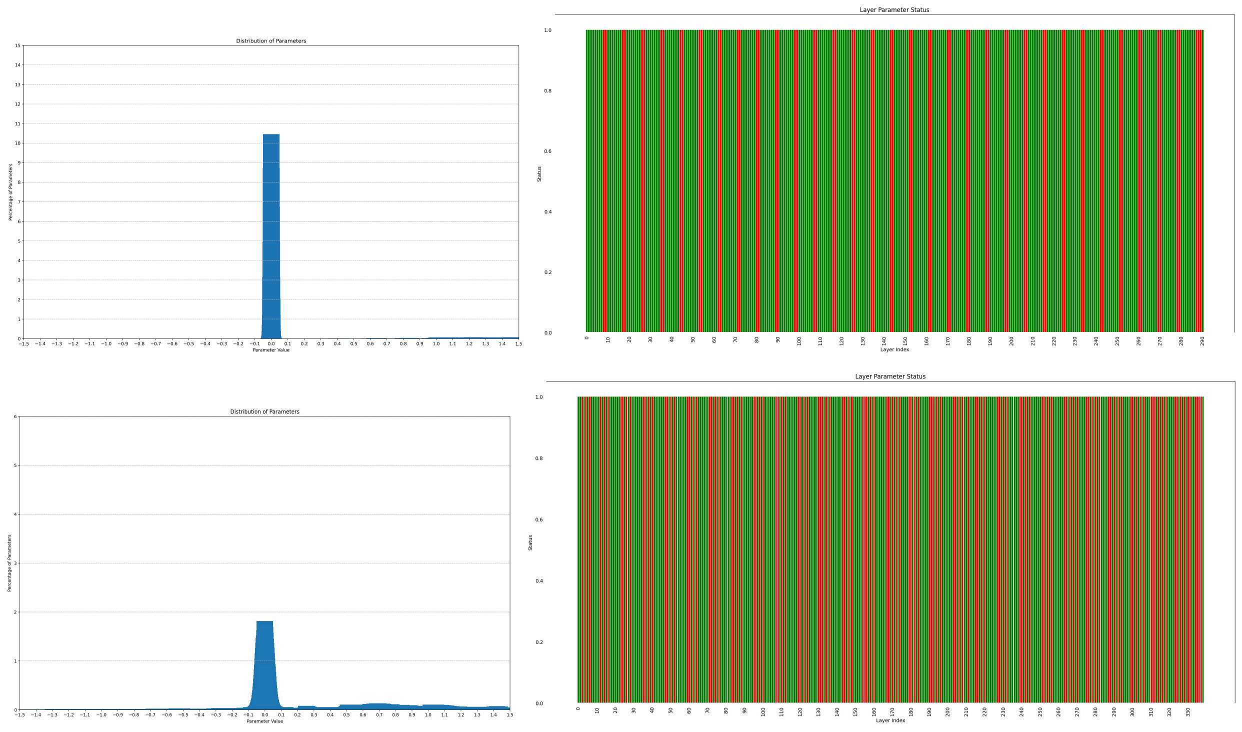


Fig 5. Model parameter distribution histograms (left) and corresponding layer distribution histograms (right):

(i) Japanese StableLMLlama-2 (ii) Llama2 (iii) MiniCPM v2.6 (iv) Mistral (v) Qwen-2

Llama has a unique parameter distribution signature around values 0.0 and 0.6, as can be seen in Japanese StableLM (fine tuned variant of Llama encoder + Stable Diffusion decoder).

Large Model (7-Billion Parameters)	**% of Outliers**	**% of Inliers**	**% of sparse parameters**	**Layers with 100% inliers**	**Smallest Parameter**	**Largest Parameter**
Mistral-7B-v0.1 (fp16)	0.0032	99.9968	0.0091	226/291	-0.28515625	13.25
Llama-2-7b-chat-hf (fp16)	0.0001	99.9999	0.0002	265/291	-1.4140625	2.84375
Japanese-stablelm-base-beta-7b (fp16)	0.0001	99.9999	~0%	262/291	-1.421875	2.859375
MiniCPM-V-2_6 (fp16)	0.0034	99.9966	0.0106	437/693	–66.5	212.0
gte-Qwen2-7B-instruct (fp16)	0.0008	99.9992	0.0881	212/339	-52.75	58.0

Table 3: Percentages of parameters lying within and outside 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.

Example:

0	1	1	1	1	1	1	1

Decimal: 0.9921875 (8 scale and 8 precision)

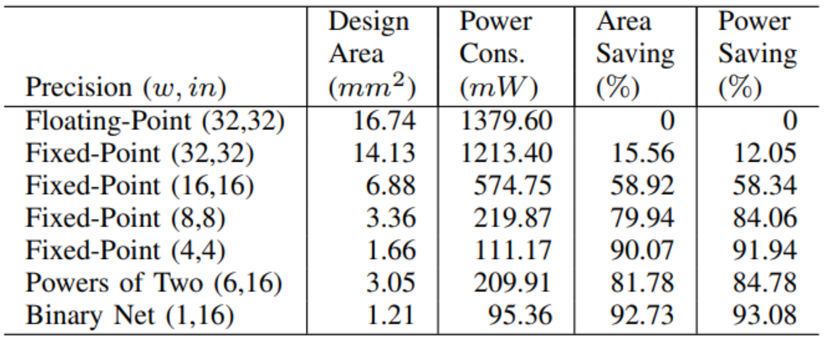
1	1	1	1	1	1	1	1

Decimal: -0.9921875 (8 scale and 8 precision)

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

Limitations of SF16 and SF8

- Works only in a specified range: The need to represent numbers beyond this range in native format arises.
- Compile-time overhead includes: Identifying layers with inliers, pattern of sparse params, continuity of inlier layers
- Run-time overhead includes: Mixed precision inference (type upcasting/downcasting), sparse multiplication, unity multiplication.



Precision (w, in)	Design Area (mm^2)	Power Cons. (mW)	Area Saving (%)	Power Saving (%)
Floating-Point (32,32)	16.74	1379.60	0	0
Fixed-Point (32,32)	14.13	1213.40	15.56	12.05
Fixed-Point (16,16)	6.88	574.75	58.92	58.34
Fixed-Point (8,8)	3.36	219.87	79.94	84.06
Fixed-Point (4,4)	1.66	111.17	90.07	91.94
Powers of Two (6,16)	3.05	209.91	81.78	84.78
Binary Net (1,16)	1.21	95.36	92.73	93.08

Link

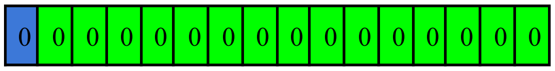
INT4

Sparse Matrix Multiplication

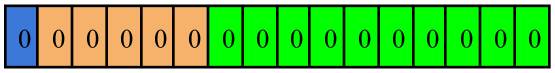
What identifies as sparse in SF16? The answer boils down to how quantization is performed. Let's compare a sf8, sf16, fp16 and fp32 side-by-side:

0	0	0	0	0	0	0	0

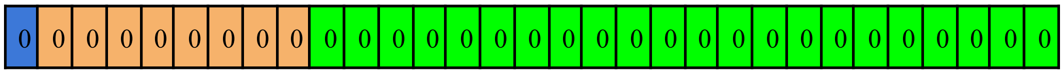
SF8 Decimal: 0



SF16 Decimal: 0



FP16 Decimal: 0

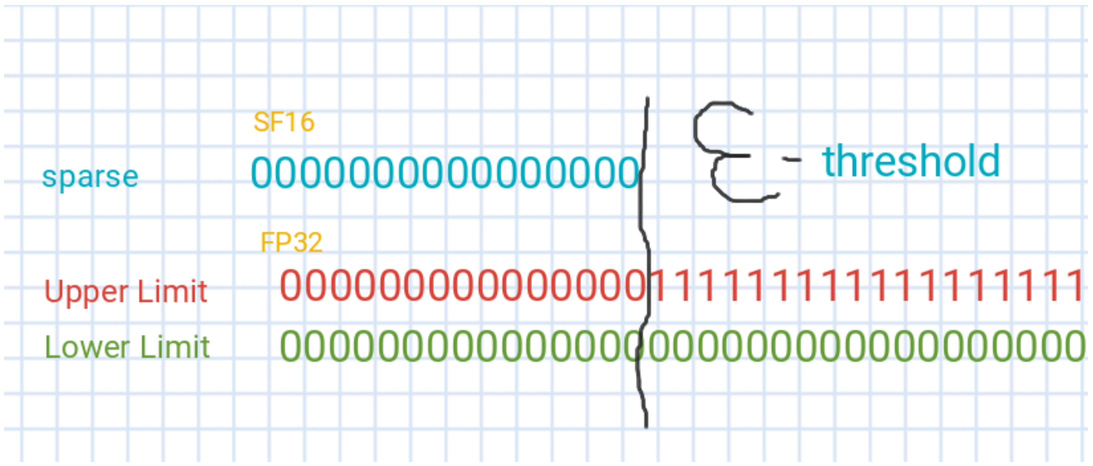


FP32 Decimal: 0

Blue - sign bit, Dark Green - Exponent bits, Light Green - Mantissa bits

As we studied before, SF16 can only represent within a specific range lying between -1 and +1. FP16 can represent floating point exponents between -14 and +15. FP32 can represent floating point exponents between -126 and +127. This means that FP16 can shift bits to a maximum of 15 positions to left or right, and in the case of FP32 the shifting value becomes 127.

Therefore, when the exponent of fp32 is 111 in decimal (or 01101111 in binary), the exponent is calculated as 111 - 127 = -16. This is the **111 Reduction Trick**. This means that the mantissa 1.m must be shifted to the right by 16 bits. The value 16 is considered the **ε-threshold for sparsity**, which indicates that any fp32 exponent value less than or equal to 111 will have the same effect. **This is how we define sparse values in sf16**.



The same makes sense while quantizing fp16 or fp32 to sf8: the value of the exponent has to be -8 or lesser.

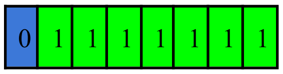
<u>Hardware Algorithm for Sparse Quantization in sf16 (111 Reduction Trick)</u>

1. Input: Receive an fp32 or fp16 number.
2. Extract Components:
   - For fp32:
     - Extract the sign bit (1 bit).
     - Extract the exponent (8 bits).
     - Extract the mantissa (23 bits).
   - For fp16:
     - Extract the sign bit (1 bit).

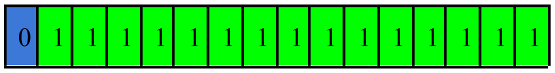
   - Extract the exponent (5 bits).
   - Extract the mantissa (10 bits).
3. Determine sf16 Value:
  - If exponent <= 01101111 (binary) / 111 (decimal), set the mantissa of sf16 to 000000000000000 (quantize to 0).
  - Else, extract the relevant bits from the mantissa after shifting to form the sf16 value:
    - If shift_amount < 0, all bits in sf16 are 0.
    - Otherwise, take the top 15 bits from the shifted mantissa.
4. Set Sign:
  - Combine the sign bit and the mantissa to form the final sf16 value.
5. Output: Return the sf16 value.

Unity Matrix Multiplication

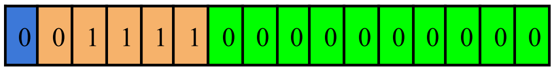
We follow an almost similar method for unity quantization in sf16. Technically any value lying between the highest bound of sf16 and the 1.0 is considered unity i.e. anything between 0.999969482421875 and 1.0 is rounded off to unity.



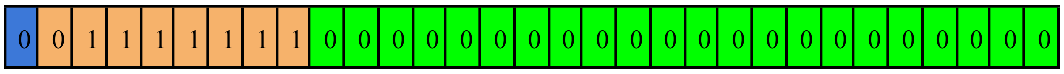
SF8 Decimal: 0.9921875



SF16 Decimal: 0.999969482421875



FP16 Decimal: 1.0



FP32 Decimal: 1.0

Blue - sign bit, Dark Green - Exponent bits, Light Green - Mantissa bits

Hardware Algorithm for Unity Quantization in sf16

1. Input: Receive an fp32 or fp16 number.
2. Extract Components:
  - For fp32:
    - Extract the sign bit (1 bit).
    - Extract the exponent (8 bits).
    - Extract the mantissa (23 bits).
  - For fp16:
    - Extract the sign bit (1 bit).
    - Extract the exponent (5 bits).
    - Extract the mantissa (10 bits).
3. Action:
  - If:

Condition 1: exponent equals 01111111 (binary) / 127 (decimal) **and** mantissa bits are all zero

Condition 2: exponent equals 01111110 (binary) / 126 (decimal) **and** first 14 bits of mantissa are 1 (binary)

-Then:

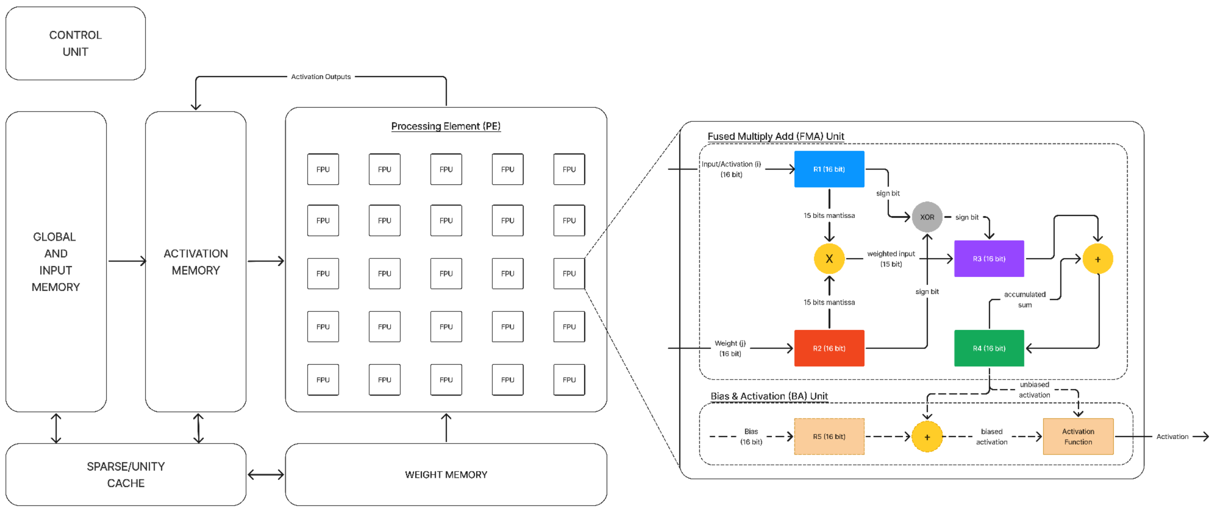
Store the other number in the result matrix (since any number multiplied by one equals that number).

- Else, extract the relevant bits from the mantissa after shifting to form the sf16 value:
  - If shift_amount < 0, all bits in sf16 are 0.
  - Otherwise, take the top 15 bits from the shifted mantissa.

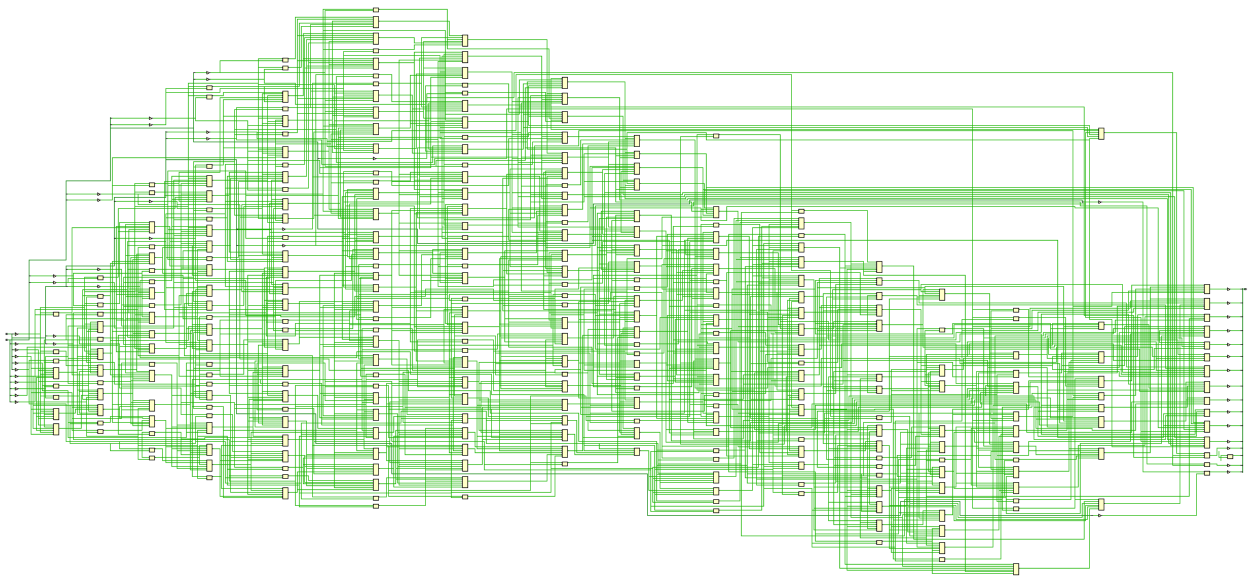
Instead of using specialized data structures (like COO or CSR) to store only non-zero values and their indices, we would work directly with the original dense matrix.

- Load the weights and activations from memory into registers.
- Check if either register value is zero (all bits 0)
- If it's zero, do not upcast; skip the multiplication, result is zero, move to the next pair of values.
- Check if either register value is exactly one.
- If either is exactly unity, do not upcast; skip the multiplication, result is the other weight, move to the next pair of values.

HIGH-LEVEL ARCHITECTURE



1. Modified Array Multiplier for SF16



2. Modified Dadda Multiplier for SF16