Beyond FP16: The Need For FP8 Training

Qiulin Shang (商秋林) Kun Yuan

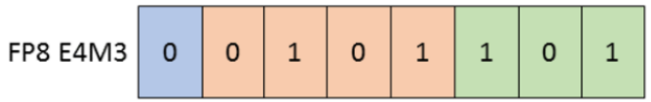
Beyond FP16: The Need For FP8 Training

- **Recap**: We have learned how FP16 mixed precision accelerates training and saves memory. The core idea is maintaining an FP32 master weight copy while performing computations in FP16.

- **Challenge**: As model sizes continue to explode, the memory and compute advantages of FP16 are becoming insufficient.

- **Question**: Can we push the compute precision even further down to 8-bit (FP8) to unlock greater performance gains?

Float 8 (E4M3)



= 0.40625

More precision

- **Sign**: 1 bit; 0 for positive and 1 for negative

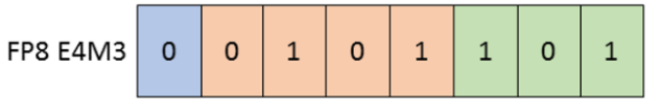
- **Exponent**: 4bit; range: 0001(1) ~ 1110(14); value range:$2^{-6} \sim 2^{7}$; offset:-7

  Example: 0101 $\rightarrow 2^{5-7} = 2^{-2}$

- **Fraction**: 3 bits;

  Example: 101 $\rightarrow$ 5 $\rightarrow$ 1 + 5/8 = 1.625

Float 8 (E4M3)



= 0.40625

More precision

- **Translation Law:**

$$(-1)^{\text{sign}} \times 2^{\text{exponent- } 7} \times \left(1 + \frac{\text{fraction}}{8}\right)$$

- **Largest Positive Number:**

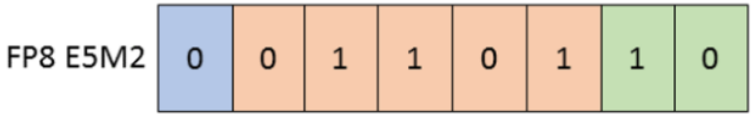
$$(-1)^0 \times 2^7 \times \left(1 + \frac{7}{8}\right) = 128 \times 1.875 = 240$$

- **Smallest Positive Number (Denormalized)**:

$$(-1)^0 \times 2^{-6} \times \left(0 + \frac{1}{8}\right) = 2^{-9} \approx 1.953 \times 10^{-3}$$

- **Range**: [−240,+240]

Float 8 (E5M2)



= 0.375

More range

- **Sign**: 1 bit; 0 for positive and 1 for negative

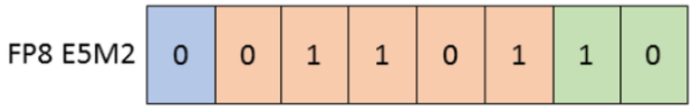
- **Exponent**: 5bit; range: 00001(1) ~ 11110(30); value range:2^{-14}~2^{15}; offset:-15

  Example: 01101 $\rightarrow 2^{13-15} = 2^{-2}$

- **Fraction**: 2 bits;

  Example: 10 $\rightarrow$ 4 $\rightarrow$ 1 + 2/4 = 1.5

Float 8 (E5M2)



= 0.375

More range

- **Translation Law:**

$$(-1)^{\text{sign}} \times 2^{\text{exponent-15}} \times \left(1 + \frac{\text{fraction}}{4}\right)$$

- **Largest Positive Number:**

$$(-1)^0 \times 2^{15} \times \left(1 + \frac{3}{4}\right) = 32768 \times 1.75 = 57344$$

- **Smallest Positive Number (Denormalized)**:

$$(-1)^0 \times 2^{-14} \times \left(0 + \frac{1}{4}\right) = 2^{-16} \approx 1.5259 \times 10^{-5}$$

- **Range**:[−57344,+57344]

FP16 vs FP8

Feature	FP16	BF16	FP8 (E4M3)	FP8 (E5M2)
Exponent Bits	5	8	4	5
Mantissa Bits	10	7	3	2
Dynamic Range	Wide	Wider	Very Narrow	Narrow
Precision	Higher	Moderate	Low	Very Low
Key Advantage	Good Balance	Good Balance	Better Precision	Wider Range
Risk	Overflow/Underflow	Moderate	High Underflow Risk	High Quantization Error

FP16 vs FP8

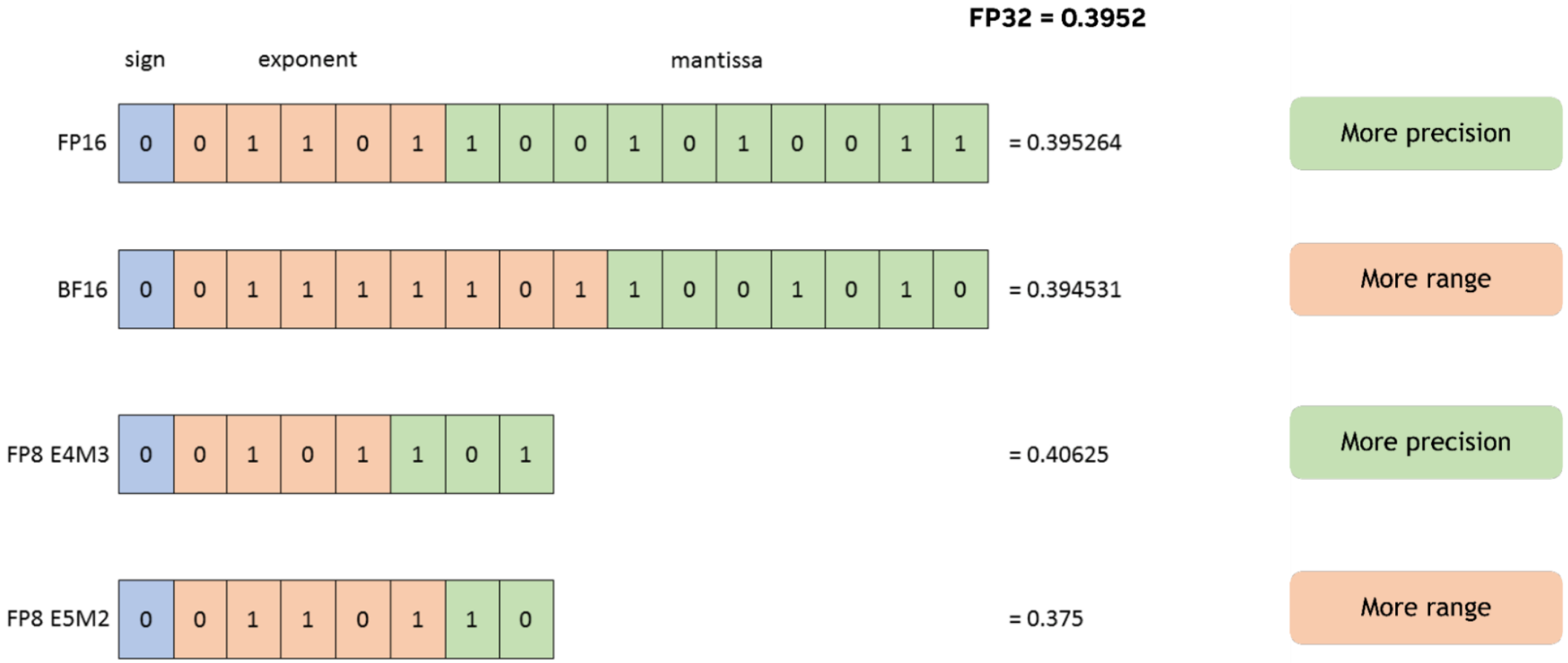


Figure 1: FP8 formats (E4M3, E5M2).

Source: NVIDIA Developer Blog [2].

E4M3 vs E5M2

- Exponent bits ↑ ⟶ wider dynamic range
- Mantissa bits ↑ ⟶ finer precision around each value
- Consider the interval [1.0, 2.0):
  - E4M3:
    - 3 mantissa bits ⇒ 2^3 = 8 distinct values per exponent
    - Step size ≈ 1/8 ≈ 0.125
    - Representable values (example): 1.00, 1.125, 1.25, 1.375, 1.50, 1.625, 1.75, 1.875
  - E5M2:
    - 2 mantissa bits ⇒ 2^2 = 4 distinct values per exponent
    - Step size ≈ 1/4 = 0.25
    - Representable values (example): 1.00, 1.25, 1.50, 1.75

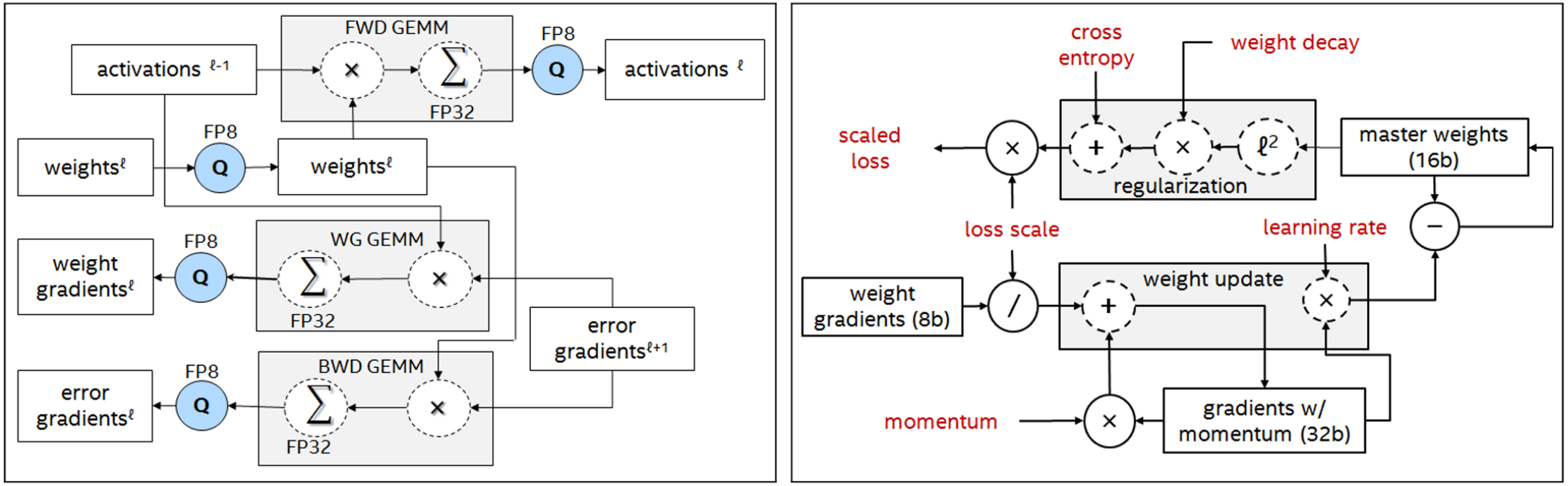
E4M3 vs E5M2

- Exponent bits ↑ ⟶ wider dynamic range
- Mantissa bits ↑ ⟶ finer precision around each value
- Quantizing x = 1.3:
  - E4M3:
    - Nearest values: 1.25 and 1.375
    - Relative error ≈ min(|1.3 – 1.25|, |1.3 – 1.375|) / 1.3 ≈ 3.8%
  - E5M2:
    - Nearest values: 1.25 and 1.50
    - Relative error ≈ min(|1.3 – 1.25|, |1.3 – 1.50|) / 1.3 ≈ 7.7%

FP8 Mixed Precision Training

- To successfully train with FP8, simply replacing all operations is **NOT** feasible.
- A mixed-precision strategy relies on three key techniques to maintain numerical stability.
  - **FP16 Master Weights**: Maintains a higher-precision (FP16) copy of the weights for accurate accumulation of small gradient updates.
  - **Enhanced Loss Scaling**: Aggressively scales the loss to prevent the extremely small gradients from vanishing (underflowing) in the limited FP8 range.
  - **Stochastic Rounding**: A probabilistic rounding method that mitigates the large quantization errors and systematic bias introduced when converting from FP32 to FP8.

The FP8 Training Loop



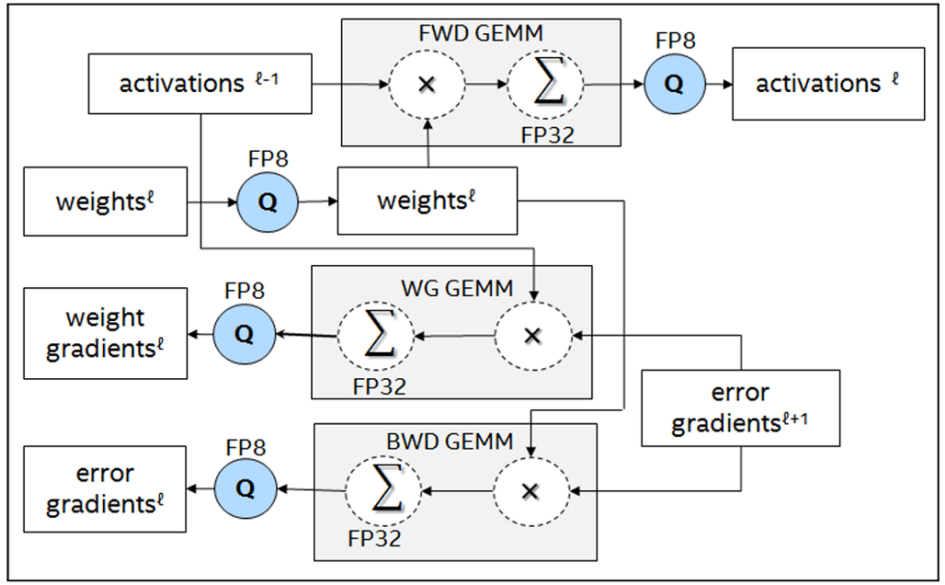
(a) (b)

Figure 1: Mixed precision data flow for FP8 training. (left) precision settings for key compute kernels in Forward, Backward, and Weight Update passes, (right) flow diagram of the weight update rule.

Source: Mixed precision training with 8-bit floating point [1].

The FP8 Training Loop

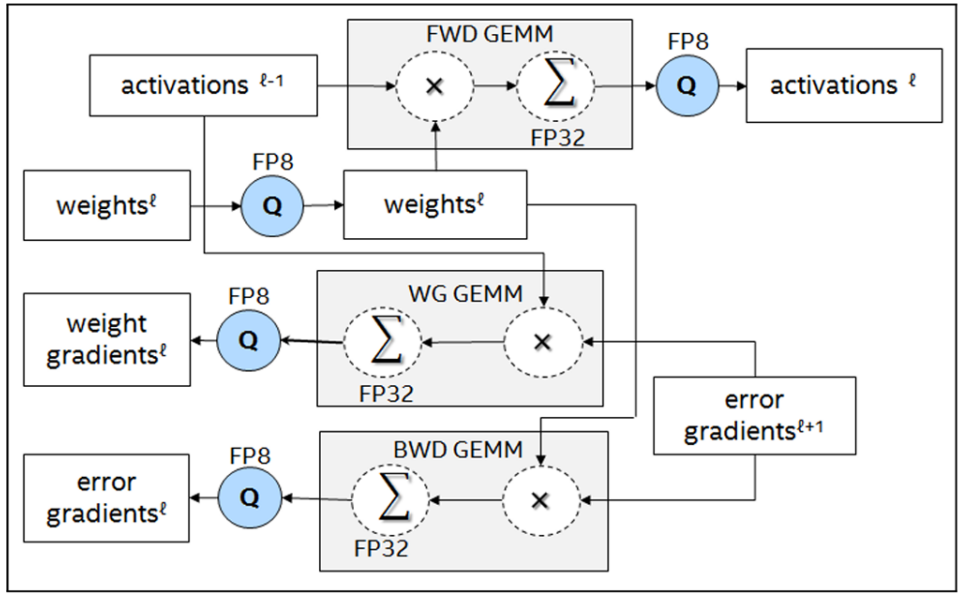
1. Forward Pass (FWD GEMM):
    - Computes: activations(t) = activations(t-1) * weights(t)
    - Process: Takes FP8 activations and FP8 weights, performs matrix multiplication using an FP32 accumulator, and quantizes the result back to FP8.



The FP8 Training Loop

2. Backward Pass (BWD GEMM):

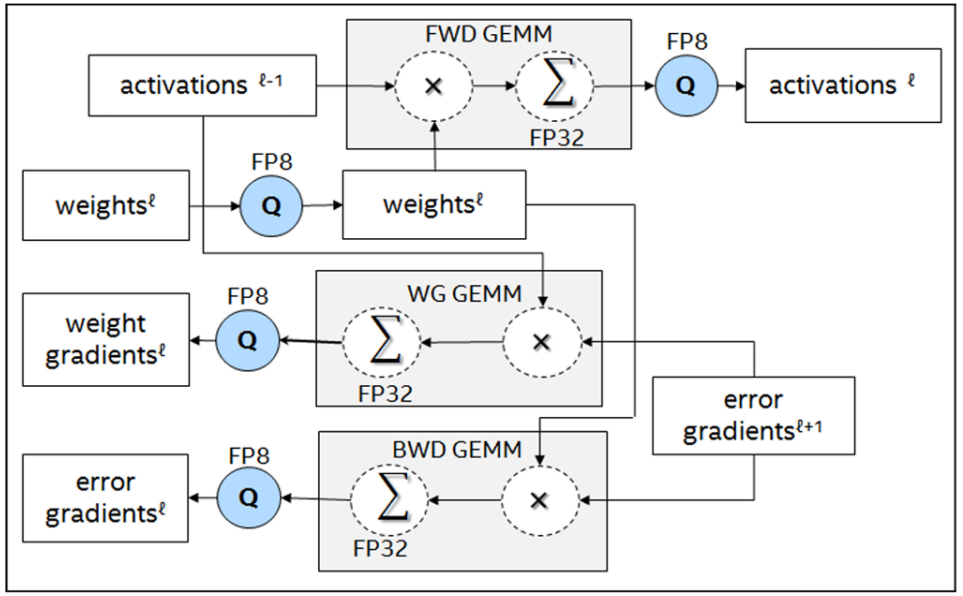
   - Computes: error_gradients(t-1) = error_gradients(t) * weights(t)
   - Process: Takes FP8 error gradients and FP8 weights, computes gradients for the previous layer in an FP32 accumulator, and quantizes the result back to FP8.



The FP8 Training Loop

3. Weight Gradient (WG GEMM):

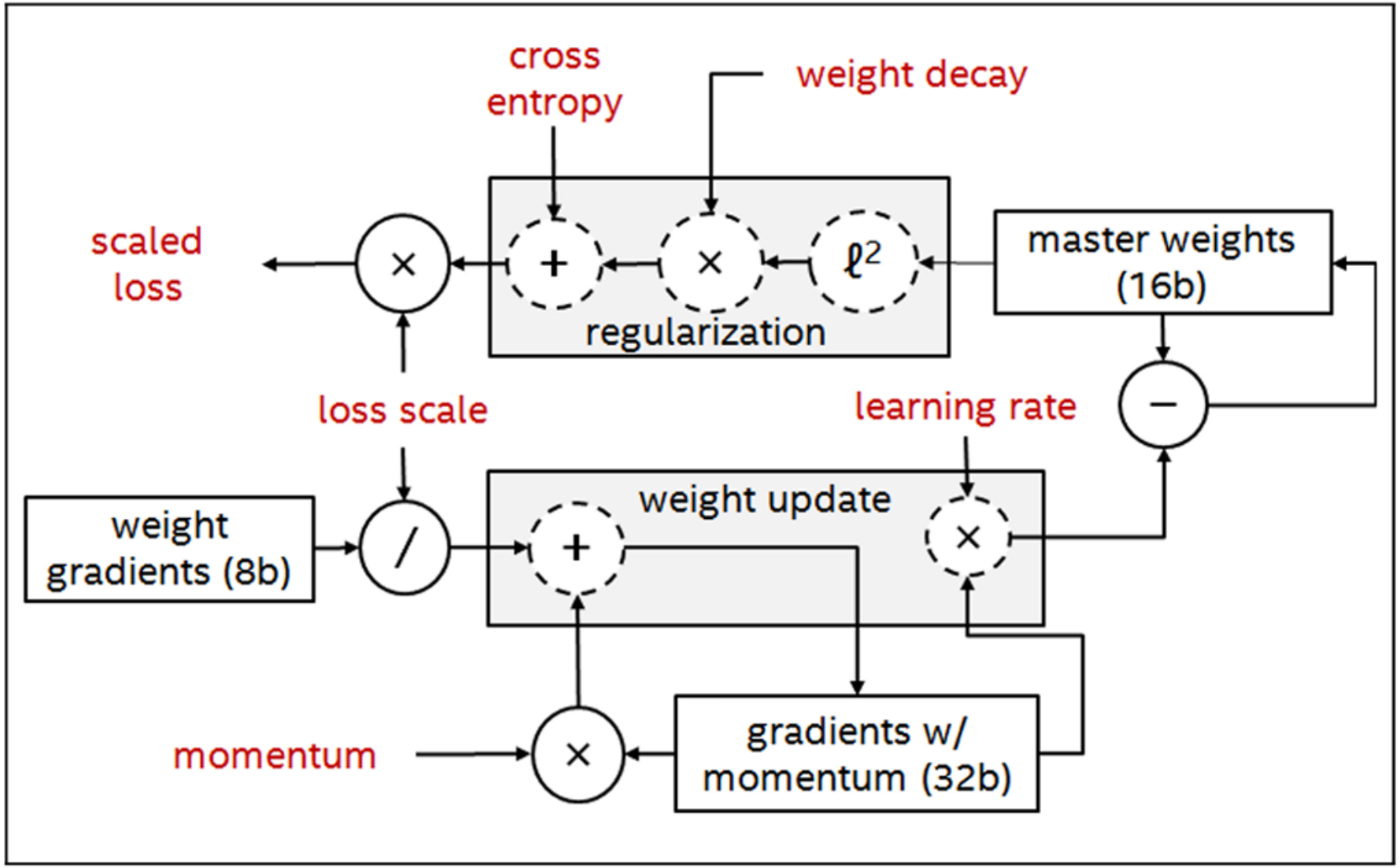
- Computes: weight_gradients(t) = error_gradients(t) * activations(t-1)
- Process: Takes FP8 error gradients and FP8 activations, computes weight gradients in an FP32 accumulator, and outputs the result in FP32 for the update step.



The Full Update Path

- Step 1: Unscale Gradients
    - The FP8 weight gradients are first converted to FP32.
    - They are then divided by the loss scale factor S. This operation is done in FP32 to prevent the original small gradient values from underflowing.
- Step 2: Optimizer Step
    - The unscaled FP32 gradients are fed into the optimizer (e.g., Adam).
    - All optimizer states (e.g., momentum buffers) are kept in FP32 to accurately track gradient history.
- Step 3: Master Weight Update
    - The FP16 master weights are loaded and up-converted to FP32.
    - The final update from the optimizer is applied to the FP32 master weights.
    - The updated FP32 weights are converted back to FP16 for storage.

The Full Update Path



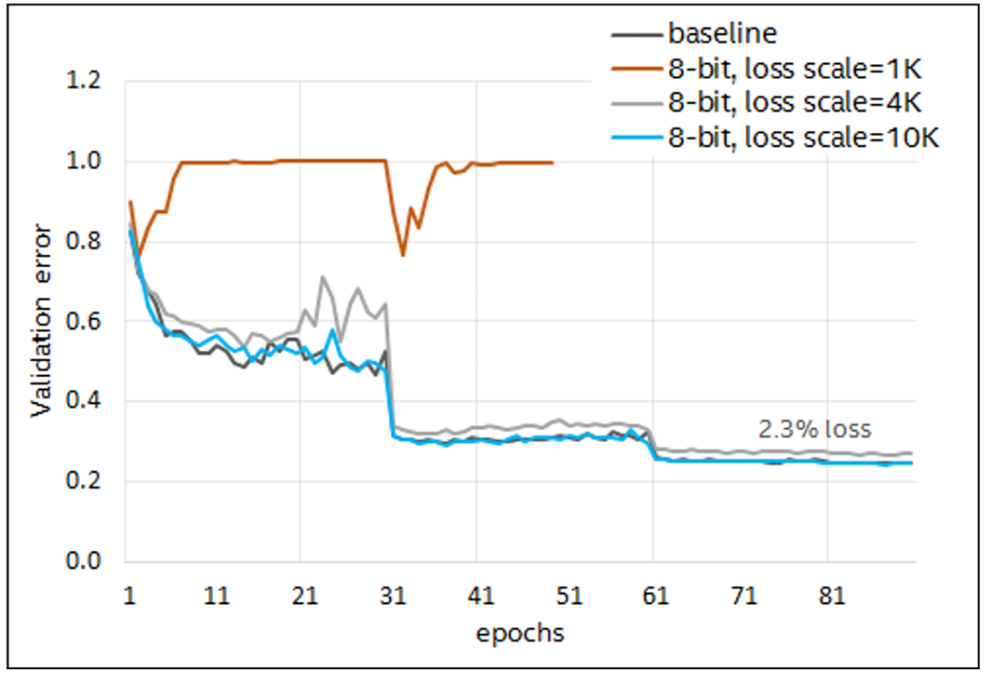
Challenge 1: Gradient Underflow

- **Problem**: FP8 has an extremely small representable range near zero. During backpropagation, gradients often become tiny and fall below this range, effectively becoming zero. This is known as underflow, and it **STOPS** learning.

Data Type	Bit Format (s, e, m)	Max Normal	Min Normal	Min Subnormal
IEEE-754 float	1, 8, 23	3.40e38	1.17e−38	1.40e−45
IEEE-754 half-float	1, 5, 10	65 535	6.10e−5	5.96e−8
FP8 (proposed)	1, 5, 2	57 344	6.10e−5	1.52e−5

Dynamic Loss Scaling

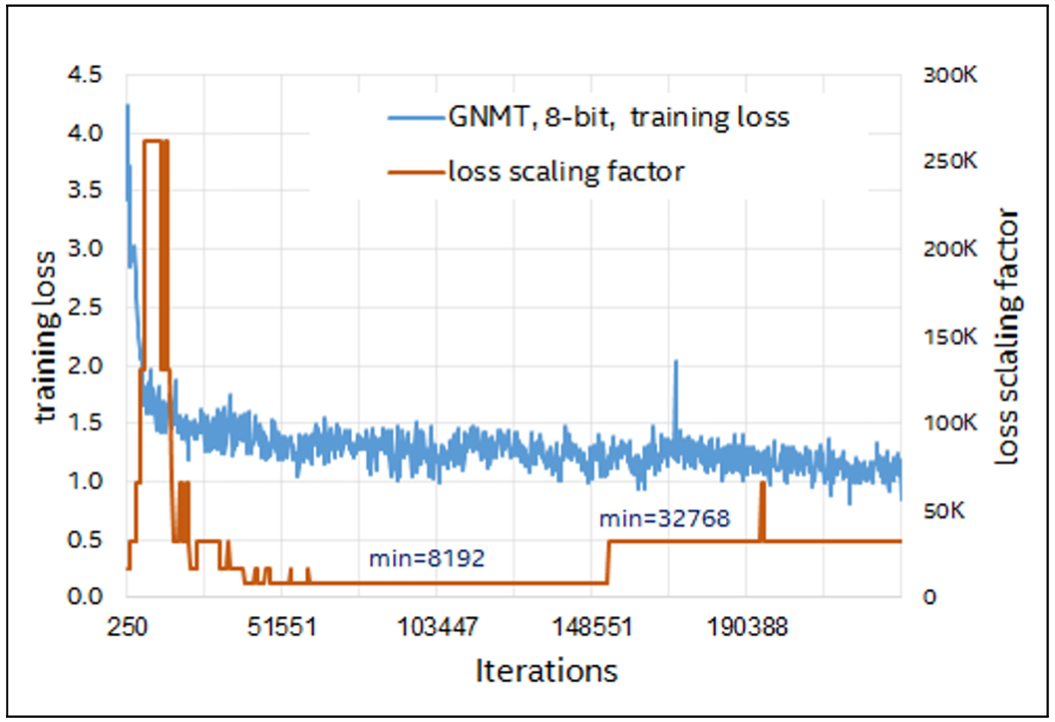
- **Amplify**: Scale the initial loss by a very large factor S (e.g., 10,000 or more). This amplifies all gradients by S, pushing them into the representable FP8 range.



Source: Mixed precision training with 8-bit floating point [1].

Dynamic Loss Scaling

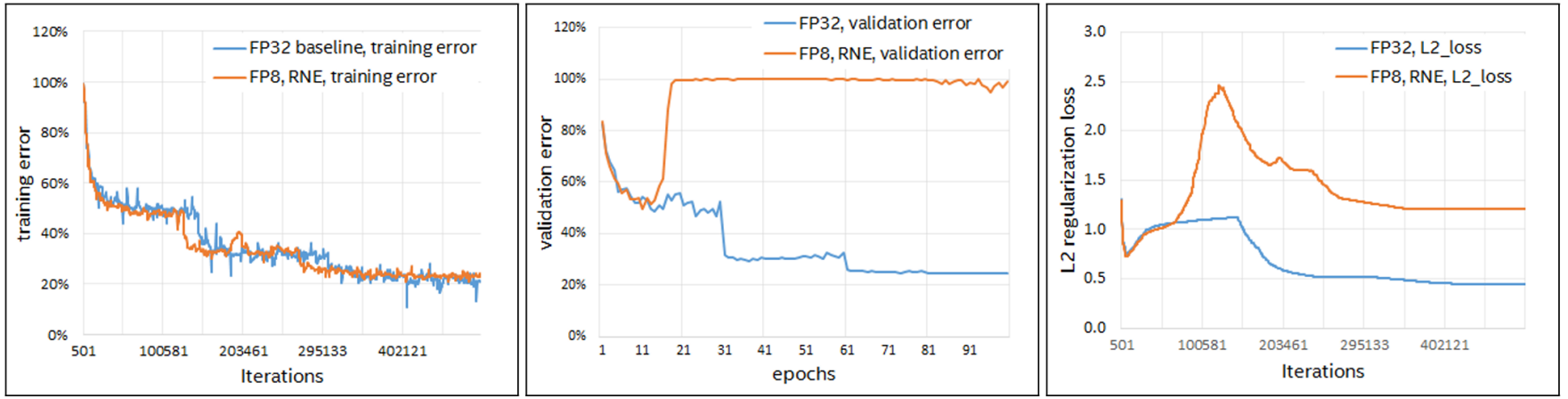
- **Adapt**: Dynamically adjust the scaling factor S during training.
  - If an overflow (inf/NaN) is detected in the gradients, decrease S (e.g., S /= 2).
  - If training is stable for N steps without overflow, increase S (e.g., S *= 2) to better resolve small gradients.



Source: Mixed precision training with 8-bit floating point [1].

Challenge 2: Quantization Error

- **Problem**: FP8 has only 2 or 3 mantissa bits, leading to massive precision loss when converting from FP32.



(a) (b) (c)

Figure 3: Impact of quantization (with RNE rounding) noise on model convergence with Resnet-50 (a) comparison of training error, (b) validation error, and (c) L2 regularization loss with FP32 baseline.

Source: Mixed precision training with 8-bit floating point [1].

Challenge 2: Quantization Error

- **Problem**: Standard rounding (Round-to-Nearest-Even, RNE) is deterministic and can introduce systematic bias, harming model accuracy.

- **Example Bias**: If data is skewed, RNE might consistently round in one direction, accumulating error.

Stochastic Rounding (SR)

$$round(x,k) = \begin{cases} \lfloor x \rfloor_k + \epsilon, & \text{with probability } P = \frac{(x - \lfloor x \rfloor_k) + r}{\epsilon} \\ \lfloor x \rfloor_k, & \text{with probability } 1 - P \end{cases}$$

- **Notation:**
  - Let $\lfloor x \rfloor_k$ be the lower grid point (round-down to nearest representable value)
  - Let $\lfloor x \rfloor_k + \epsilon$ be the upper grid point
  - ε: quantization step (distance between two adjacent representable values)
  - δ = $x - \lfloor x \rfloor_k$: distance from x to the lower grid point ($0 \leq \delta \leq \varepsilon$)
  - r is a tiny random offset, which can be set to 0
- **Key property:**
  - E[round(x, k)] = x (unbiased in expectation)

Stochastic Rounding (SR)

$$round(x,k) = \begin{cases} \lfloor x \rfloor_k + \epsilon, & \text{with probability } P = \frac{(x - \lfloor x \rfloor_k) + r}{\epsilon} \\ \lfloor x \rfloor_k, & \text{with probability } 1 - P \end{cases}$$

- **Assume quantization step ε = 0.1**
  - Representable values: …, 0.2, 0.3, 0.4, 0.5, …
- **Example 1: x = 0.37**
  - Lower grid point: $\lfloor x \rfloor_k$ = 0.3
  - Upper grid point: $\lfloor x \rfloor_k + \epsilon$ = 0.4
  - Distance to lower: δ = x – 0.3 = 0.07
  - Probability of rounding up: P = δ / ε = 0.07 / 0.1 = 0.7
    - With probability 0.7: round(x) = 0.4
    - With probability 0.3: round(x) = 0.3

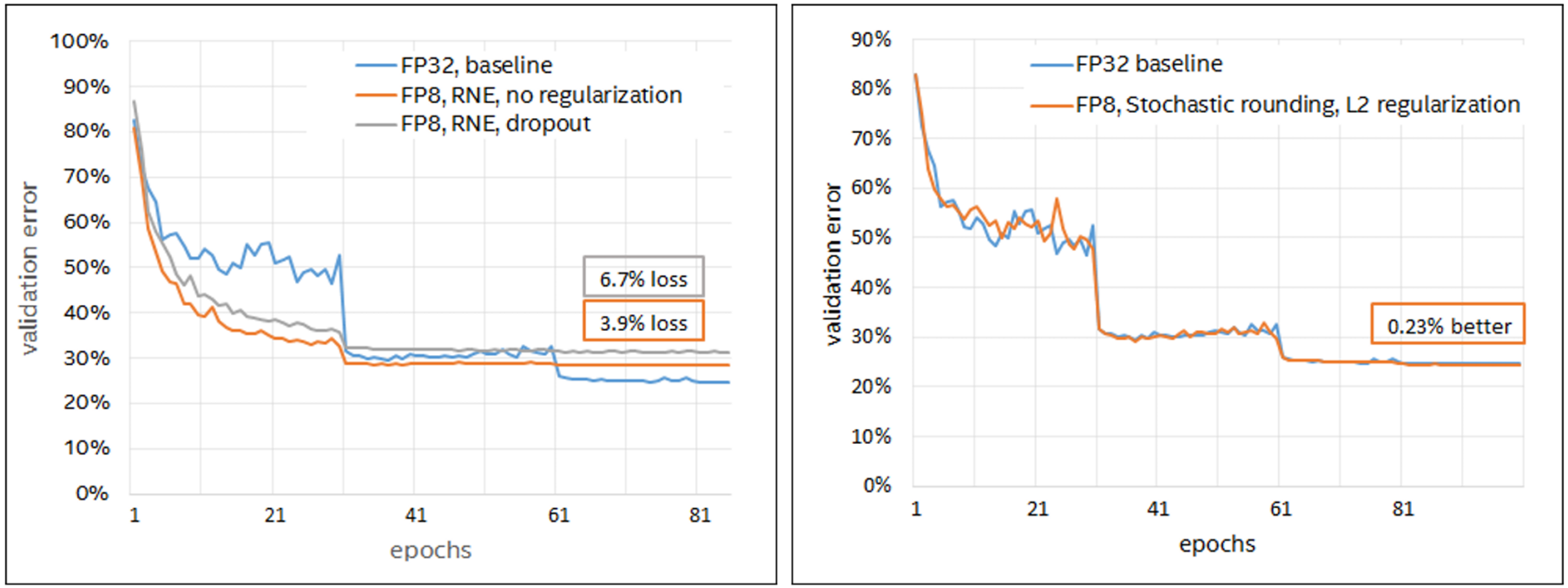
Stochastic Rounding (SR)

$$round(x,k) = \begin{cases} \lfloor x \rfloor_k + \epsilon, & \text{with probability } P = \frac{(x - \lfloor x \rfloor_k) + r}{\epsilon} \\ \lfloor x \rfloor_k, & \text{with probability } 1 - P \end{cases}$$

- **Probabilistic Approach**: Instead of always rounding to the nearest value, SR rounds up or down based on a probability determined by the value's position between two representable numbers.

- **Key Benefit**: SR is unbiased in expectation (E[round(x)] = x). The introduced randomness also acts as a form of regularization, which can help prevent overfitting.

Stochastic Rounding (SR)

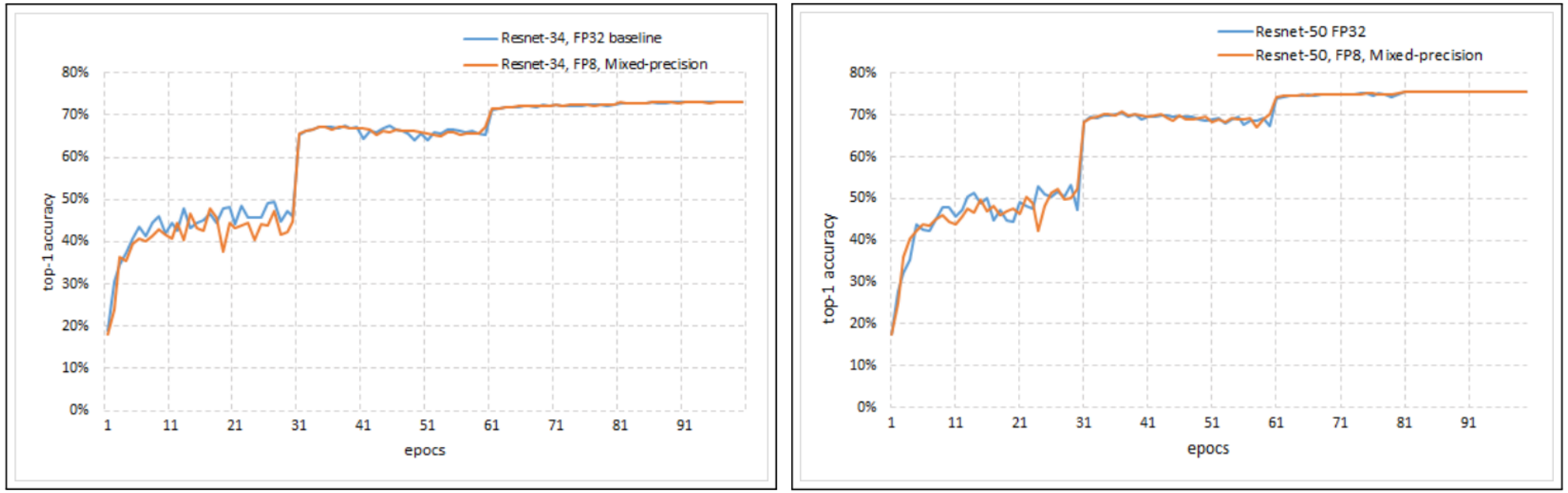


(a) (b)

Figure 4: (a) Comparing validation performance with 'dropout' and noise-based implicit regularization techniques using RNE(round to nearest even) (b) model performance with stochastic rounding with L2 regularization.

Source: Mixed precision training with 8-bit floating point [1].

Numerical Studies



(a) (b)

Figure 5: Convergence plots showing Top-1 validation accuracy for. (a) Resnet-34[12] (b) Resnet-50[12] on imagenet-1K[7] dataset.

Source: Mixed precision training with 8-bit floating point [1].

Summary

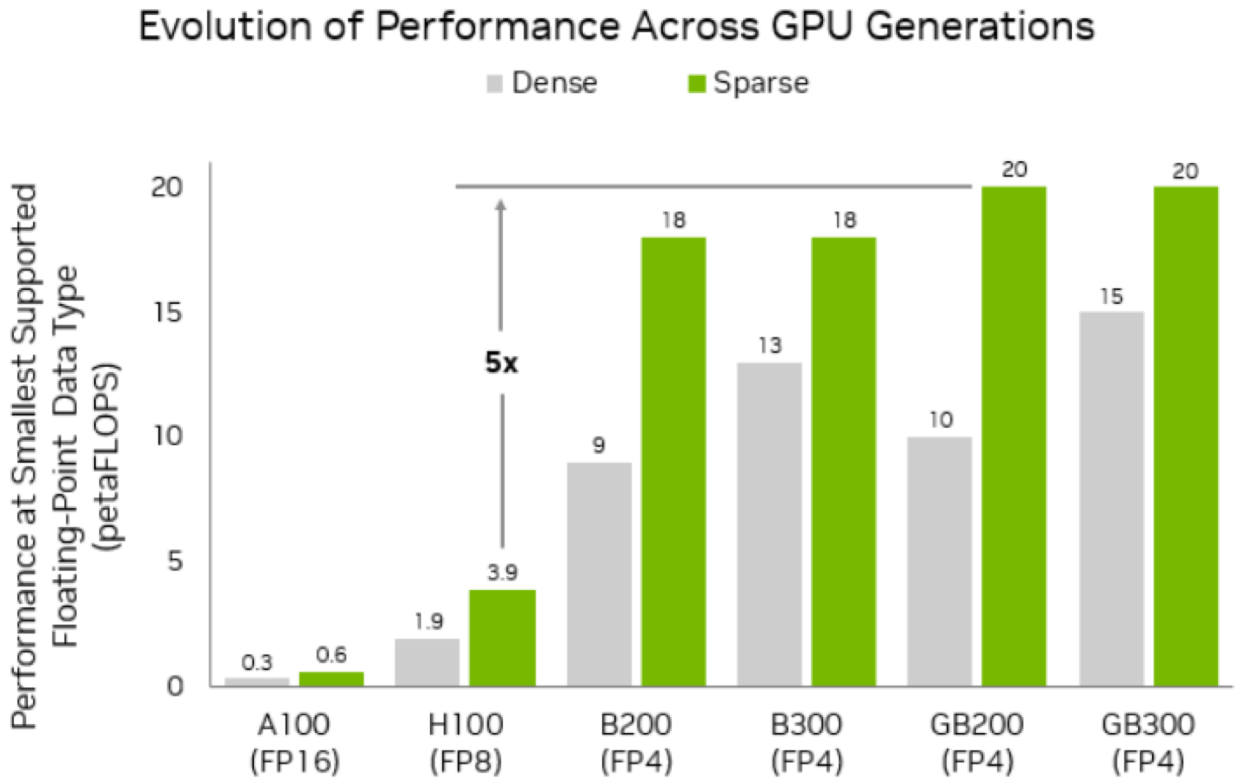
- **FAST Compute, ACCURATE Updates**
    - Compute-intensive operations (GEMMs) use FP8 for maximum speed.
    - Update-critical operations (optimizer, weight updates) use FP32 for maximum accuracy.
- **Memory Efficiency**
    - FP16 master weights offer a good balance between memory savings and update precision.
- **Two CRITICAL Enablers**
    - Dynamic Loss Scaling is essential to prevent gradient underflow.
    - Stochastic Rounding is essential to mitigate catastrophic quantization error.

NVFP4 : Ultra-Low Precision Training for LLMs

Guohong Li (李国鸿) Kun Yuan

Why do we need NVFP4?

1. Training frontier LLMs requires massive compute
2. FP8 is widely adopted but efficiency limits still remain
3. FP4 enables 2–3× faster compute and ~50% memory savings



NVIDIA Blackwell GPU to support NVFP4

https://developer.nvidia.cn/blog/introducing-nvfp4-for-efficient-and-accurate-low-precision-inference/

What is NVFP4?

- NVIDIA's enhanced 4-bit FP format
- Uses microscaling to preserve dynamic range
- Better outlier representation vs MXFP4
- Supported on **Blackwell** Tensor Cores

Feature	FP4 (E2M1)	MXFP4	NVFP4
Format Structure	4 bits (1 sign, 2 exponent, 1 mantissa) plus software scaling factor	4 bits (1 sign, 2 exponent, 1 mantissa) plus 1 shared power-of-two scale per 32 value block	4 bits (1 sign, 2 exponent, 1 mantissa) plus 1 shared FP8 scale per 16 value block
Accelerated Hardware Scaling	No	Yes	Yes
Memory	Up to 4x less memory than FP16		
Accuracy	Risk of noticeable accuracy drop compared to FP8	Risk of noticeable accuracy drop compared to FP8	Lower risk of noticeable accuracy drop particularly for larger models

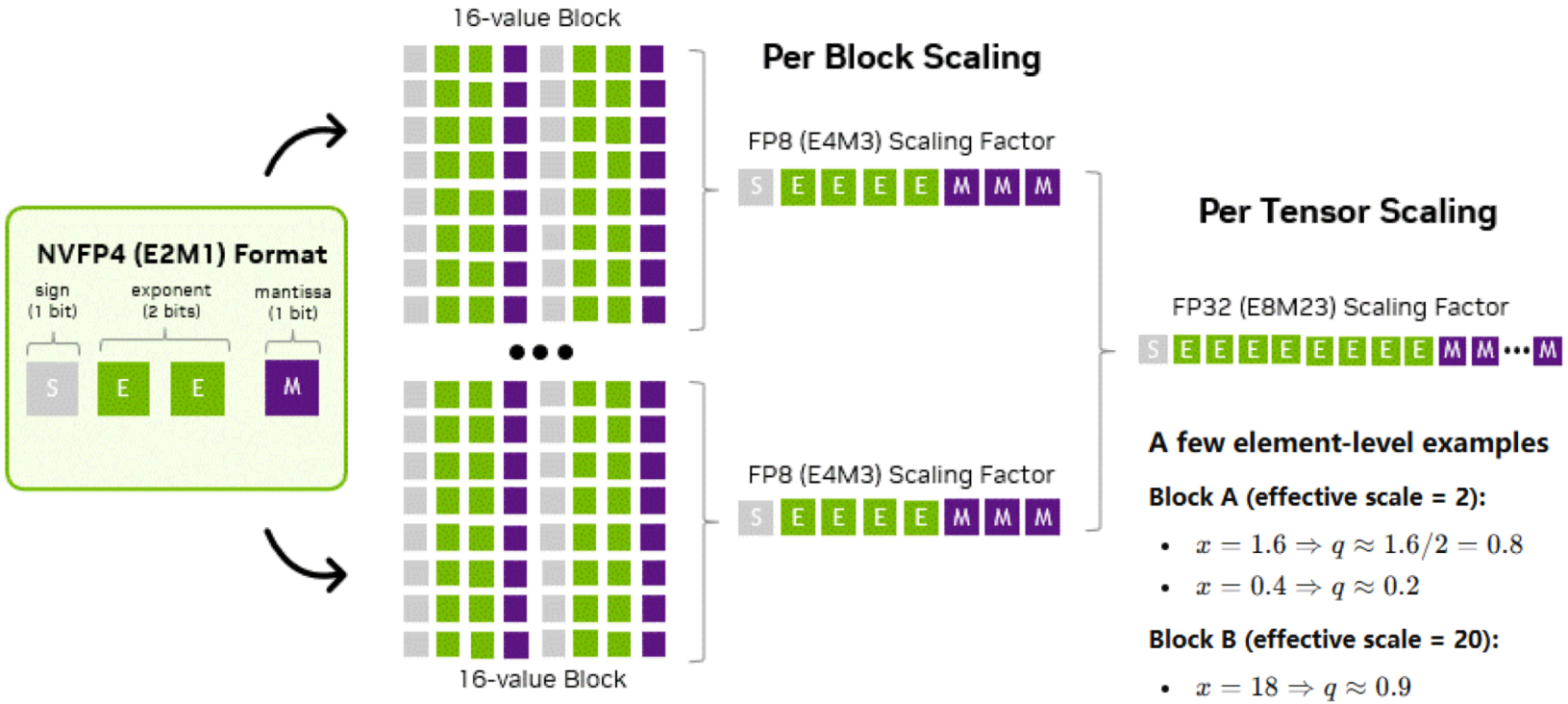
Table 1. Comparison of Blackwell-supported 4-bit floating point formats

https://developer.nvidia.cn/blog/introducing-nvfp4-for-efficient-and-accurate-low-precision-inference/

Why is NVFP4 better?

We reduce numerical errors through **scaling**.
NVFP4 uses two architectural innovations to make it more precise:

1. High-precision scale encoding
2. A two-level micro-block scaling strategy (like follow)



A few element-level examples

Block A (effective scale = 2):

- $x = 1.6 \Rightarrow q \approx 1.6/2 = 0.8$
- $x = 0.4 \Rightarrow q \approx 0.2$

Block B (effective scale = 20):

- $x = 18 \Rightarrow q \approx 0.9$
- $x = 6 \Rightarrow q \approx 0.3$

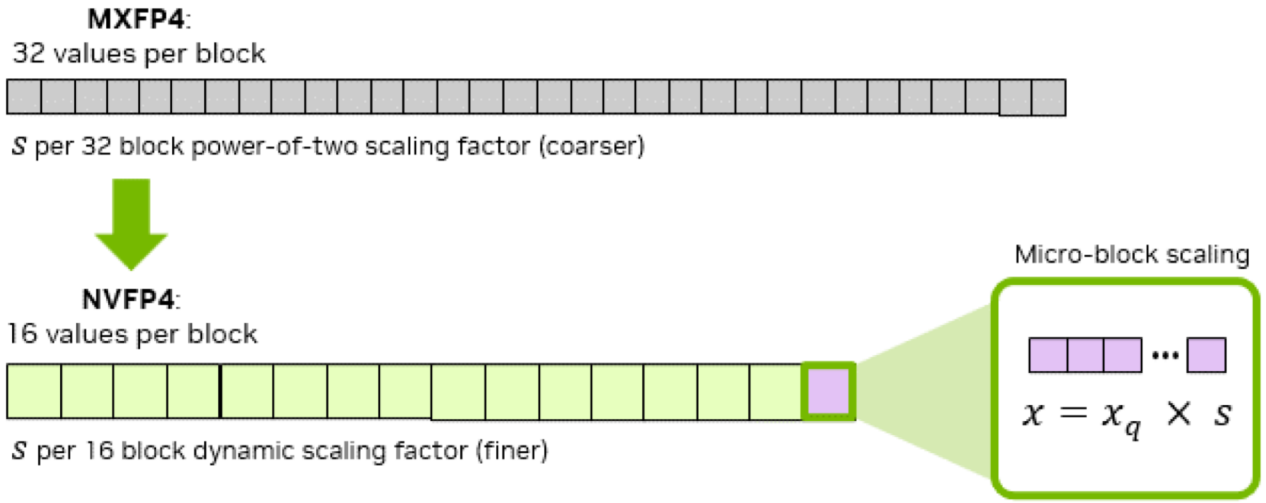
These q values are now in a small, friendly range for **NVFP4**, so the 4-bit format doesn't have to cover the entire tensor's dynamic range by itself.

https://developer.nvidia.cn/blog/introducing-nvfp4-for-efficient-and-accurate-low-precision-inference/

How shared scaling works, for one block?

Micro-block scaling for efficient model compression

Large tensors in AI models often mix large and small numbers, and a single “umbrella” scaling can lead to significant quantization errors that degrade model performance.
The tighter grouping in NVFP4 offers twice as many opportunities to match the local dynamic range of the data, significantly reducing those errors.

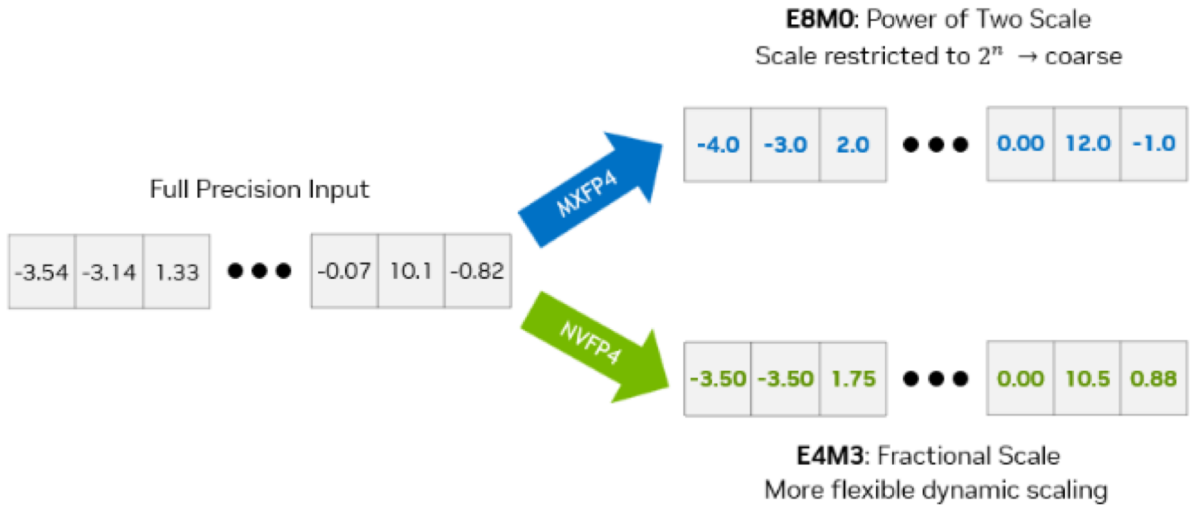


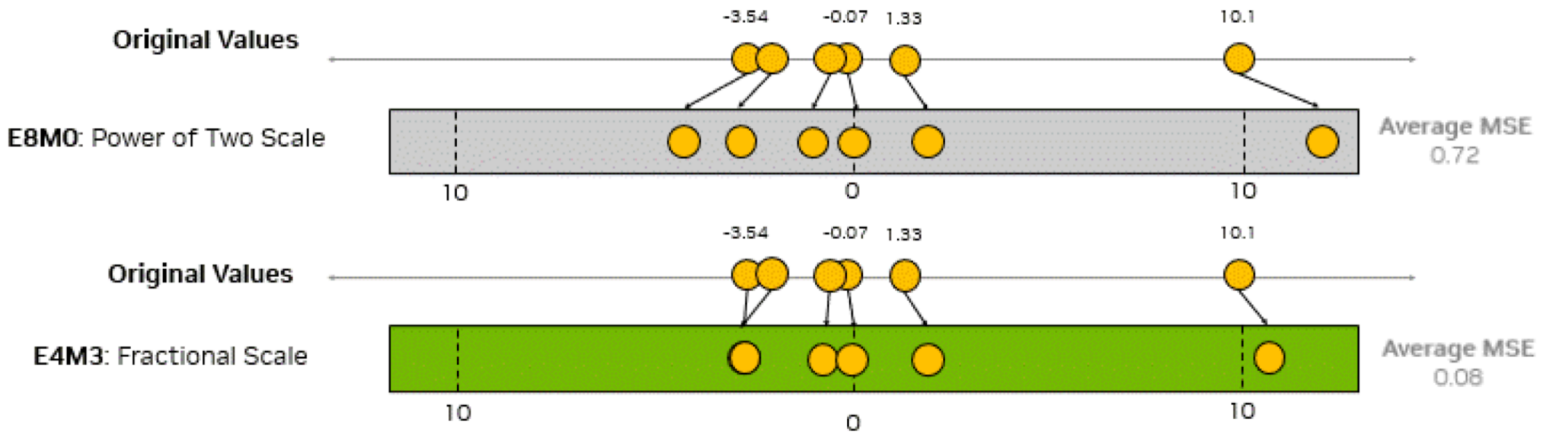
Here x_q is 4-bit encoded value (between the range of -6 to +6)

https://developer.nvidia.cn/blog/introducing-nvfp4-for-efficient-and-accurate-low-precision-inference/

Which format for scaling? (FP8, for example)

High-precision scaling: Encoding more signal, less error

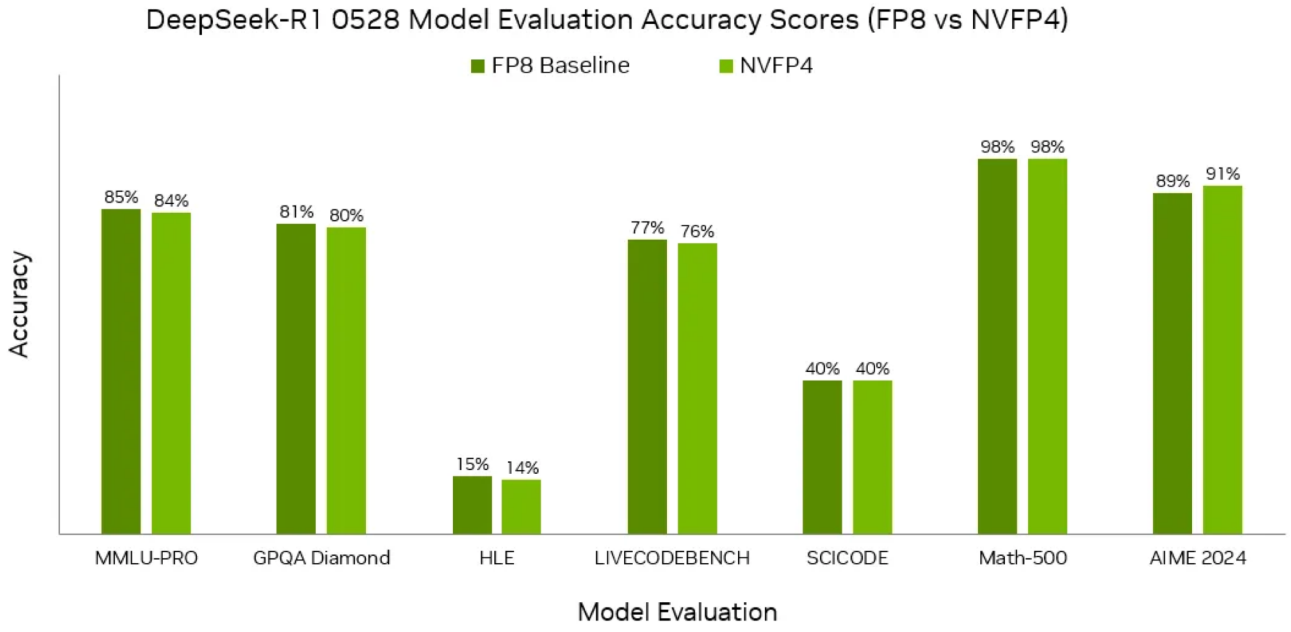


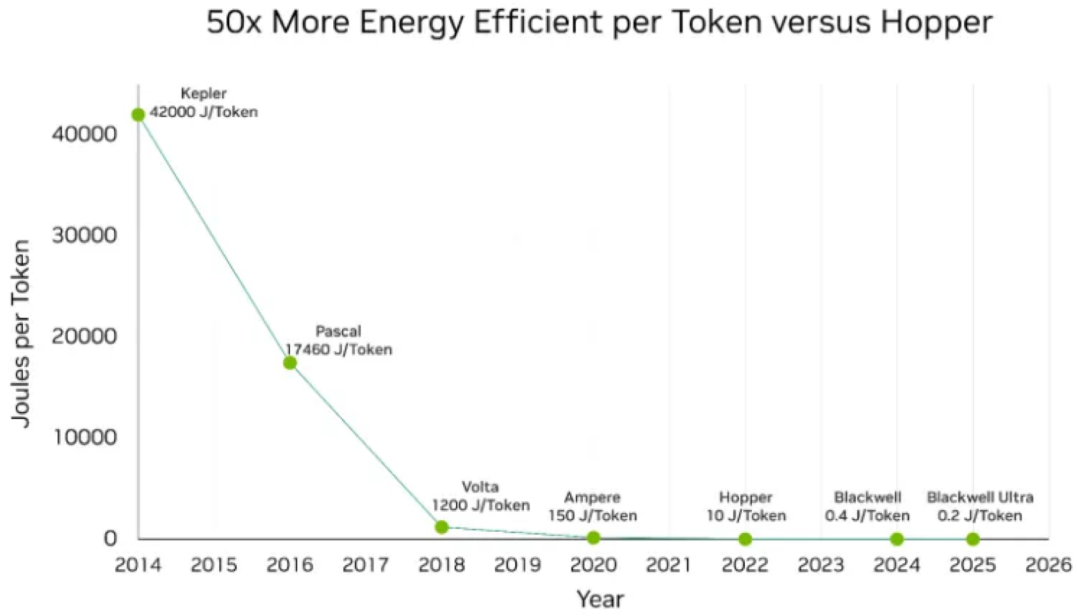


https://developer.nvidia.cn/blog/introducing-nvfp4-for-efficient-and-accurate-low-precision-inference/

NVFP4 versus FP8: Model performance and memory efficiency

Quantization benefits are driven by two factors: reduced **memory** burden and simplified **compute** operations.
These two factors reduce pressure on memory bandwidth which can improve output token throughput.
It can also improve overall end-to-end latency performance as a result of simplified attention layer computations which yield direct benefits during prefill.

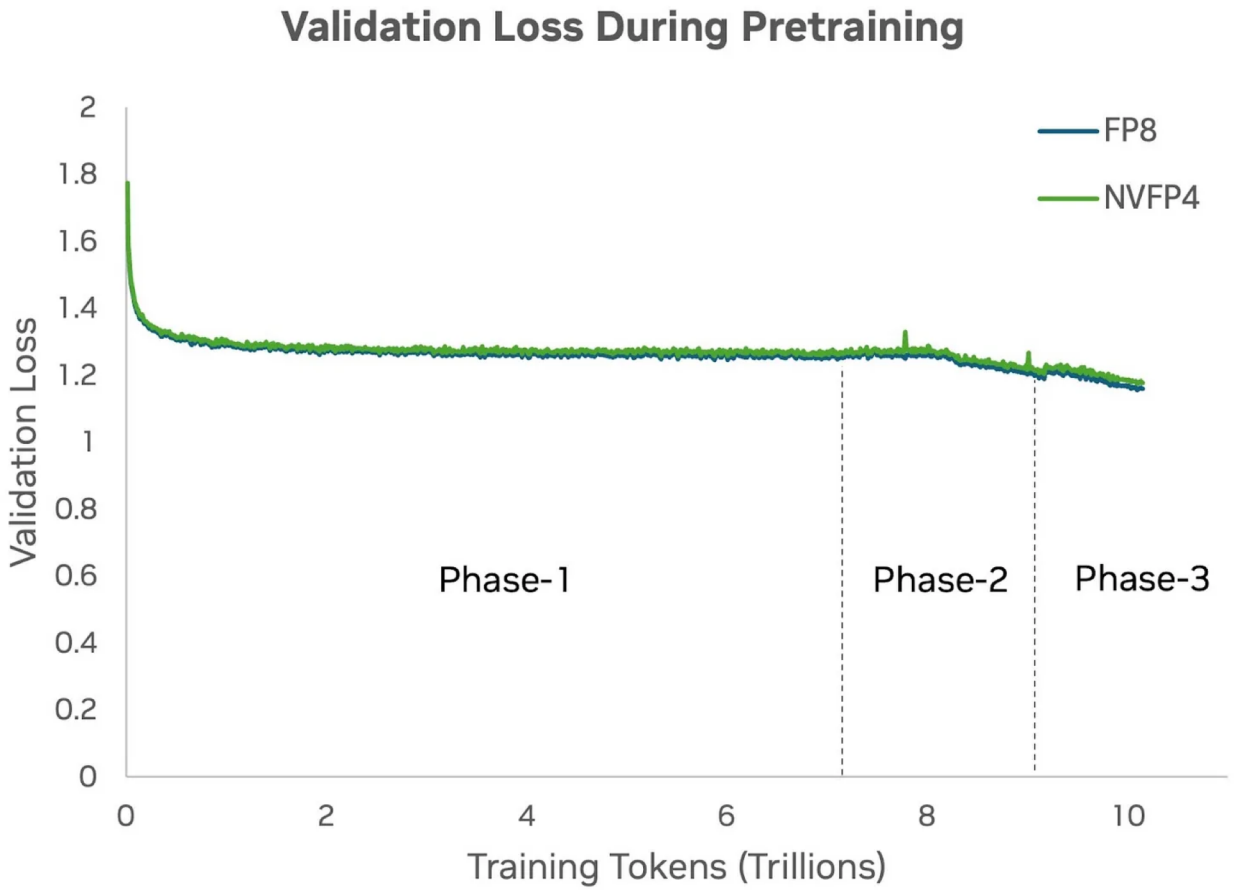




https://developer.nvidia.cn/blog/introducing-nvfp4-for-efficient-and-accurate-low-precision-inference/

NVFP4 versus FP8: Training loss

A comparison of validation losses was performed on a **12B Hybrid Mamba-Transformer** model pre-trained using FP8 and NVFP4 accuracy on a 10T token.
The results show that the loss curve of NVFP4 is **highly consistent** with the loss curve of FP8 (baseline) throughout the training process.



https://developer.nvidia.cn/blog/nvfp4-trains-with-precision-of-16-bit-and-speed-and-efficiency-of-4-bit/

Example Usage: transformer engine, mxfp8 & nvfp4

Sometimes it is desirable to use multiple recipes in the same training run. An example of this is the NVFP4 training, where a few layers at the end of the training should be run in higher precision.

import transformer_engine.pytorch as te

```
my_linear1 = te.Linear(768, 768).bfloat16()  # The first linear - we want to run it in FP4
my_linear2 = te.Linear(768, 768).bfloat16()  # The second linear - we want to run it in MXFP8

inp = inp.bfloat16()

with te.autocast(recipe=nvfp4_recipe):
    y = my_linear1(inp)
    with te.autocast(recipe=mxfp8_recipe):
        out = my_linear2(y)

print(out)

out.mean().backward()
```

https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/examples/fp8_primer.html

Hardware & Software Support for Low-Precision Training

Weichen Jia (贾维宸) Kun Yuan

Hardware Support for Low-Precision Training

Different GPUs have varying computing power to support different precisions.

GPU Precision Performance Comparison Summary (Plain Text Version)

Precision Type	NVIDIA A100 80GB (Ampere)	NVIDIA H20 96GB (Hopper China Edition)	NVIDIA RTX PRO 6000 (Blackwell)
FP64 (Double Precision)	9.7 TFLOPS / (19.5 TFLOPS, Tensor Core)	~1 TFLOPS (Does not support FP64 TC)	~2 TFLOPS / (1/64 FP32, CUDA Core only)
FP32 (Single Precision)	19.5 TFLOPS	~44 TFLOPS	126.0 TFLOPS
TF32 (Tensor Core)	156 / 312* TFLOPS	~74 / 148* TFLOPS	251.9 / 503.8* TFLOPS
BF16 (Tensor Core)	312 / 624* TFLOPS	~148 / 296* TFLOPS (Speculative)	503.8 / 1007.6* TFLOPS
FP16 (Tensor Core)	312 / 624* TFLOPS	148 / 296* TFLOPS	503.8 / 1007.6* TFLOPS
FP8 (E4M3 / E5M2)	Not Supported	≈300 / 600* TFLOPS (Estimated)	1007.6 / 2015.2* TFLOPS
INT8 (Tensor Core)	624 / 1248* TOPS	296 / 592* TOPS	1007.6 / 2015.2* TOPS
INT4 (Tensor Core)	1248 / 2496* TOPS	≈592 / 1184* TOPS (Estimated)	2015.2 / 4030.4* TOPS
FP4 (Tensor Core)	Not Supported	Not Supported	2015.2 / 4030.4* TFLOPS

NVIDIA's China-Compliant H20 GPU Has 41% Fewer Cores & 28% Lower Performance Versus Top Hopper H100 Config

NVIDIA A100 | Tensor Core GPU

(79 封私信) 大型语言模型的低精度训练：方法、挑战与机遇 - 知乎

Hardware Support for Low-Precision Training

Ampere (like A100) and Volta (like V100) CANNOT deal with FP8, but Hopper can.

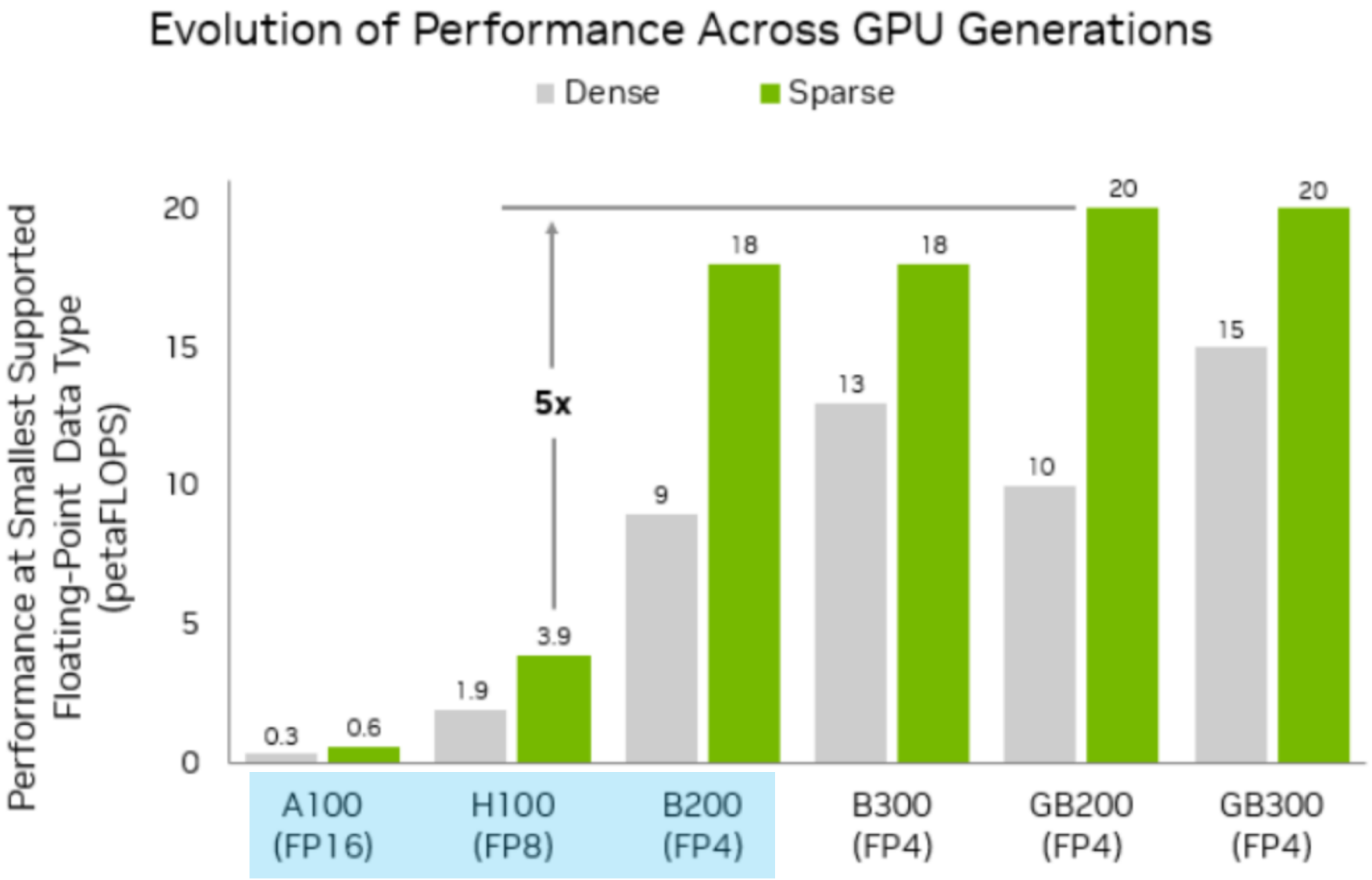
Hopper (like H20) CANNOT deal with FP4, but Blackwell can.

	Hopper	Ampere	Turing	Volta
Supported Tensor Core precisions	FP64, TF32, bfloat16, FP16, FP8, INT8	FP64, TF32, bfloat16, FP16, INT8, INT4, INT1	FP16, INT8, INT4, INT1	FP16
Supported CUDA® Core precisions	FP64, FP32, FP16, bfloat16, INT8	FP64, FP32, FP16, bfloat16, INT8	FP64, FP32, FP16, INT8	FP64, FP32, FP16, INT8

RTX 4090 and 5090 CAN not only deal with Genshin (原神) impact, but also deal with FP8 !

(79 封私信) 英伟达不同架构显卡支持的精度 - 知乎　《原神》官方网站-米哈游开放世界冒险RPG

Hardware Support for Low-Precision Training



隆重推出 NVFP4，实现高效准确的低精度推理 - NVIDIA 技术博客

Hardware Support for Low-Precision Training

- **New-gen GPUs lead in low precision:**
  - RTX PRO 6000 (Blackwell) outperforms A100/H20 in FP8/FP4 for low-precision tasks.

- **GPUs have distinct strengths:**
  - A100 for general training, H20 (large memory) for high-low precision deployment, RTX PRO 6000 for ultra-low precision inference.

- **Choose GPUs based on precision needs:**
  - A100 CANNOT perform FP8 training, and H20 CANNOT perform FP4 training.

Hardware Support for Low-Precision Training

- **New-gen GPUs lead in low precision:**
  - RTX PRO 6000 (Blackwell) outperforms A100/H20 in FP8/FP4 for low-precision tasks.

- **GPUs have distinct strengths:**
  - A100 for general training, H20 (large memory) for high-low precision deployment, RTX PRO 6000 for ultra-low precision inference.

- **Choose GPUs based on precision needs:**
  - A100 CANNOT perform FP8 training, and H20 CANNOT perform FP4 training.

Hardware Support for Low-Precision Training

- **New-gen GPUs lead in low precision:**
  - RTX PRO 6000 (Blackwell) outperforms A100/H20 in FP8/FP4 for low-precision tasks.

- **GPUs have distinct strengths:**
  - A100 for general training, H20 (large memory) for high-low precision deployment, RTX PRO 6000 for ultra-low precision inference.

- **Choose GPUs based on precision needs:**
  - A100 CANNOT perform FP8 training, and H20 CANNOT perform FP4 training.

Software Support for Low-Precision Training

- **Cuda Support**

After version 2.2, PyTorch's native FP8 support is **"limited"**. While it is easy to create FP8 tensors, some basic arithmetic operations on FP8 tensors are not supported. Specific functions, such as torch._scaled_mm, are required for matrix multiplication.

```
import torch
from tabulate import tabulate

f32_type = torch.float32
bf16_type = torch.bfloat16
e4m3_type = torch.float8_e4m3fn
e5m2_type = torch.float8_e5m2
```

```
output, output_amax = torch._scaled_mm(
        torch.randn(16,16, device=device).to(e4m3_type),
        torch.randn(16,16, device=device).to(e4m3_type).t(),
        bias=torch.randn(16, device=device).to(bf16_type),
        out_dtype=e4m3_type,
        scale_a=torch.tensor(1.0, device=device),
        scale_b=torch.tensor(1.0, device=device)
    )
```

Software Support for Low-Precision Training

- **Transformer Engine (TE)**

TE is a library for accelerating Transformer models on NVIDIA GPUs, including using 8-bit floating point (FP8) precision on Hopper and Blackwell GPUs, to provide better performance with lower memory utilization in both training and inference.

PyTorch

```python
import torch
import transformer_engine.pytorch as te
from transformer_engine.common import recipe

# Set dimensions.
in_features = 768
out_features = 3072
hidden_size = 2048

# Initialize model and inputs.
model = te.Linear(in_features, out_features, bias=True)
inp = torch.randn(hidden_size, in_features, device="cuda")

# Create an FP8 recipe. Note: All input args are optional.
fp8_recipe = recipe.DelayedScaling(margin=0, fp8_format=recipe.Format.E4M3)

# Enable autocasting for the forward pass
with te.autocast(enabled=True, recipe=fp8_recipe):
    out = model(inp)

loss = out.sum()
loss.backward()
```

- **Easy-to-use** modules for building Transformer layers with FP8 support
- Optimizations (e.g. fused kernels) for Transformer models
- Support for FP8 on NVIDIA Hopper and Blackwell GPUs
- Support for optimizations across all precisions (FP16, BF16) on NVIDIA Ampere GPU architecture generations and later

GitHub - NVIDIA/TransformerEngine: A library for accelerating Transformer models on NVIDIA GPUs, including using 8-bit and 4-bit floating point (FP8 and FP4) precision on Hopper, Ada and Blackwell GPUs, to provide better performance with lower memory utilization in both training and inference.

Software Support for Low-Precision Training

- **Accelerate**

Accelerate provides integrations with the ***TransformersEngine***, ***MS-AMP***, and ***torchao*** packages to enable training with low-precision methods on specified supported hardware.

Start with:

```python
from accelerate import Accelerator
from accelerate.utils import MSAMPRecipeKwargs
kwargs = [MSAMPRecipeKwargs()]
# Or to specify the backend as `TransformersEngine` even if MS-AMP is installed
# kwargs = [TERecipeKwargs()]
# Or to use torchao
# kwargs = [AORecipeKwargs()]
accelerator = Accelerator(mixed_precision="fp8", kwarg_handlers=kwargs)
```

Config:

```yaml
mixed_precision: fp8
fp8_config:
  amax_compute_algo: max
  amax_history_len: 1024
  backend: TE
  fp8_format: HYBRID
  interval: 1
  margin: 0
  override_linear_precision: (false, false, false)
  use_autocast_during_eval: false
```

低精度训练方法 - Hugging Face 文档

Software Support for Low-Precision Training

- **Accelerate** with ***MS-AMP***

  MS-AMP is generally easier to configure as it has only one parameter: **the optimization level**.
  Two optimization levels: "O1" and "O2" :
  - "O1" : weights, gradients, and all_reduce communications to 8-bit, while the rest is done in 16-bit.
  - "O2" : further converts first-order optimizer states to 8-bit, with second-order states in FP16.

Start with:

```
from accelerate import Accelerator
from accelerate.utils import FP8RecipeKwargs
kwargs = [FP8RecipeKwargs(backend="msamp", optimization_level="O2")]
accelerator = Accelerator(mixed_precision="fp8", kwarg_handlers=kwargs)
```

Config:

```
mixed_precision: fp8
fp8_config:
    backend: MSAMP
    opt_level: O2
```

低精度训练方法 - Hugging Face 文档

Software Support for Low-Precision Training

- **Accelerate** with ***TransformersEngine***

As mentioned earlier, **TE** offers numerous options to customize how and which computations are performed in FP8. For convenience, these options are restated as docstrings in FP8KwargsHandler.

Accelerate attempts to set reasonable default values, but exploring and adjusting various parameters yourself may yield better performance.

Start with:

```
from accelerate import Accelerator
from accelerate.utils import FP8RecipeKwargs
kwargs = [FP8RecipeKwargs(backend="te", ...)]
accelerator = Accelerator(mixed_precision="fp8", kwarg_handlers=kwargs)
```

Config:

```
mixed_precision: fp8
fp8_config:
    amax_compute_algo: max
    amax_history_len: 1024
    backend: TE
    fp8_format: HYBRID
    interval: 1
    margin: 0
    override_linear_precision: (false, false, false)
    use_autocast_during_eval: false
```

低精度训练方法 - Hugging Face 文档

Software Support for Low-Precision Training

- **Accelerate** with ***torchao***

torchao is a PyTorch-powered, modifiable FP8 backend designed to be more user-friendly than the previous two engines. A key difference between **ao** and the other two is that for numerical stability, it is generally advisable to **keep the first and last layers of the model in regular precision** (either FP32 or BF16), while quantizing the remaining layers to FP8.

Start with:

```
from accelerate import Accelerator
from accelerate.utils import AORecipeKwargs
kwargs = [AORecipeKwargs()]
accelerator = Accelerator(mixed_precision="fp8", kwarg_handlers=kwargs)
```

See more details in ao/torchao/float8 at main · pytorch/ao · GitHub

Software Support for Low-Precision Training

- **A variety of low-precision integrations/implementations are available, with *TransformersEngine (TE)* and others being commonly used.**

- **Different methods feature distinct implementation details and mixed-precision techniques, requiring careful configuration.**

- **Complementary to hardware, they also reach into CUDA kernels.**

Software Support for Low-Precision Training

- **A variety of low-precision integrations/implementations are available, with *TransformersEngine (TE)* and others being commonly used.**

- **Different methods feature distinct implementation details and mixed-precision techniques, requiring careful configuration.**

- **Complementary to hardware, they also reach into CUDA kernels.**

Software Support for Low-Precision Training

- **A variety of low-precision integrations/implementations are available, with *TransformersEngine (TE)* and others being commonly used.**

- **Different methods feature distinct implementation details and mixed-precision techniques, requiring careful configuration.**

- **Complementary to hardware, they also reach into CUDA kernels.**

Reference

[1] Mellempudi N, Srinivasan S, Das D, et al. Mixed precision training with 8-bit floating point[J]. arXiv preprint arXiv:1905.12334, 2019.

[2] NVIDIA Developer Blog, “Floating Point 8: An Introduction to Efficient Lower Precision AI Training”

[3] Pretraining Large Language Models with NVFP4
https://arxiv.org/abs/2509.25149

[4] NVFP4 实现 4 位预训练：万亿级 Token 的准确性和稳定性
https://developer.nvidia.cn/blog/nvfp4-trains-with-precision-of-16-bit-and-speed-and-efficiency-of-4-bit/

[5] 隆重推出 NVFP4，实现高效准确的低精度推理https://developer.nvidia.cn/blog/introducing-nvfp4-for-efficient-and-accurate-low-precision-inference/

[6] Using FP8 and FP4 with Transformer Engine
https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/examples/fp8_primer.html