PyTorch CONFERENCE EUROPE 2026

fp8 training

from Hopper to Blackwell

Luca Wehrstedt
Meta - Fundamental AI Research

Goal

Give you a lay of the land

A map to find your own way around fp8 techniques and challenges

I will not show a single line of code, and never mention PyTorch

Why fp8?

Reason 1: faster compute

GPUs are getting faster

But they're fastest at lower precision!

This matters for compute-bound applications (training, inference prefill)

Reason-ish 2: bandwidth

Workloads bound on memory bandwidth can also benefit, by loading/storing half the bytes

For example: KV cache in inference decoding

This can be done across all GPU generations

However, not the focus of this talk

Non-reason 3: footprint

Lower precision can also reduce memory footprint, but this is not always the case!

Sometimes we actually end up storing multiple copies of the same tensor in different precisions

Different floats

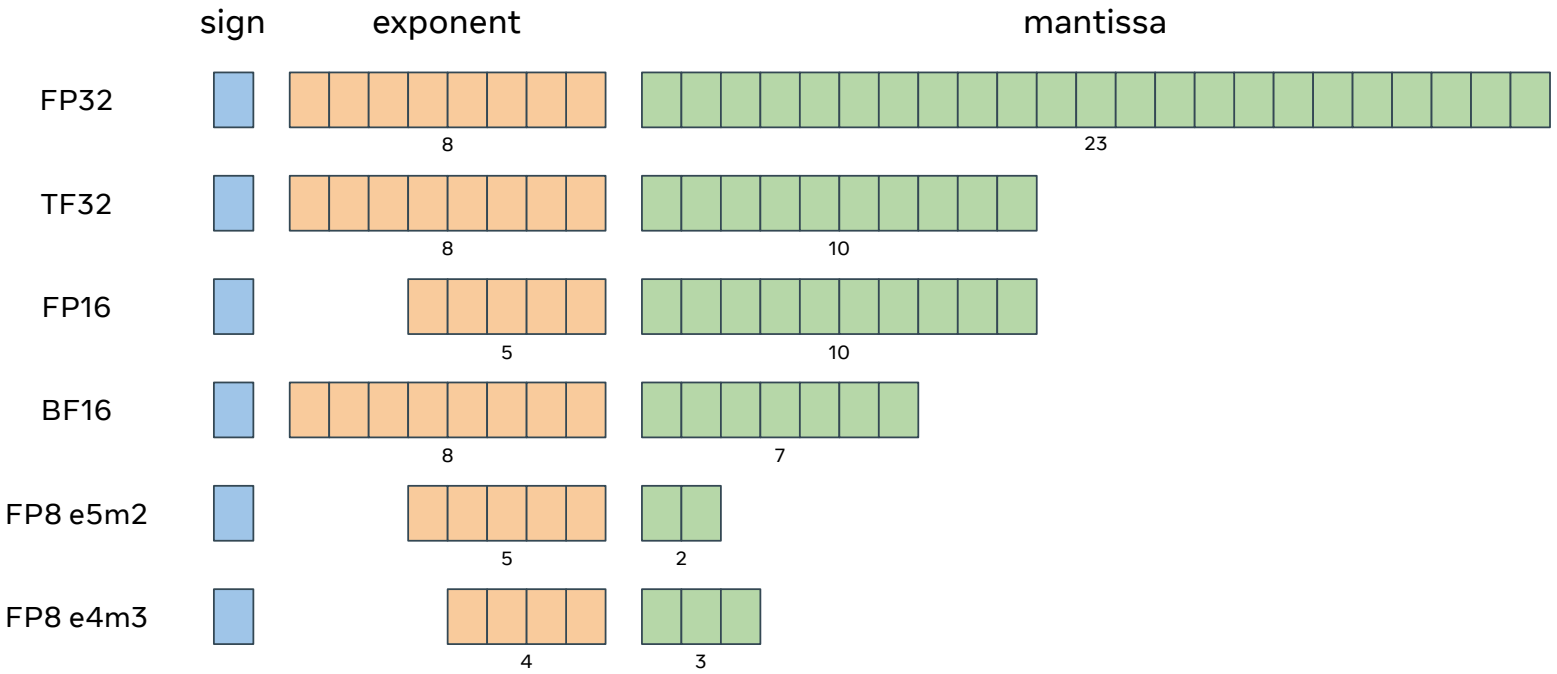
What is a float?

$$\{+1, -1\} \times 1.\text{mantissa} \times 2^{\text{exponent} - \text{offset}}$$

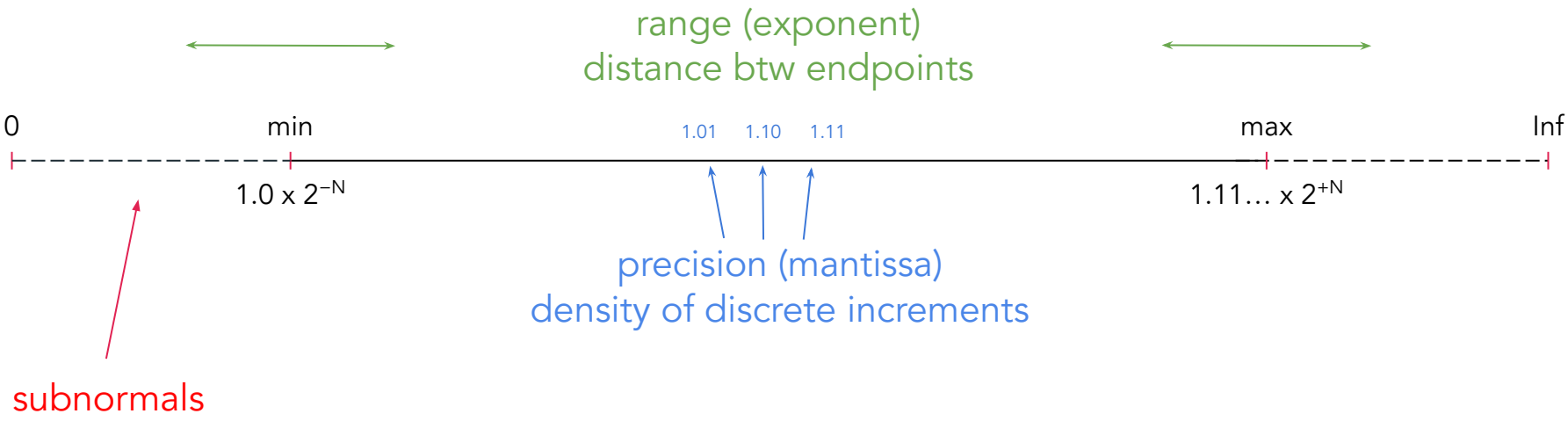
plus some special values:

±0.0 ±Inf NaN subnormals

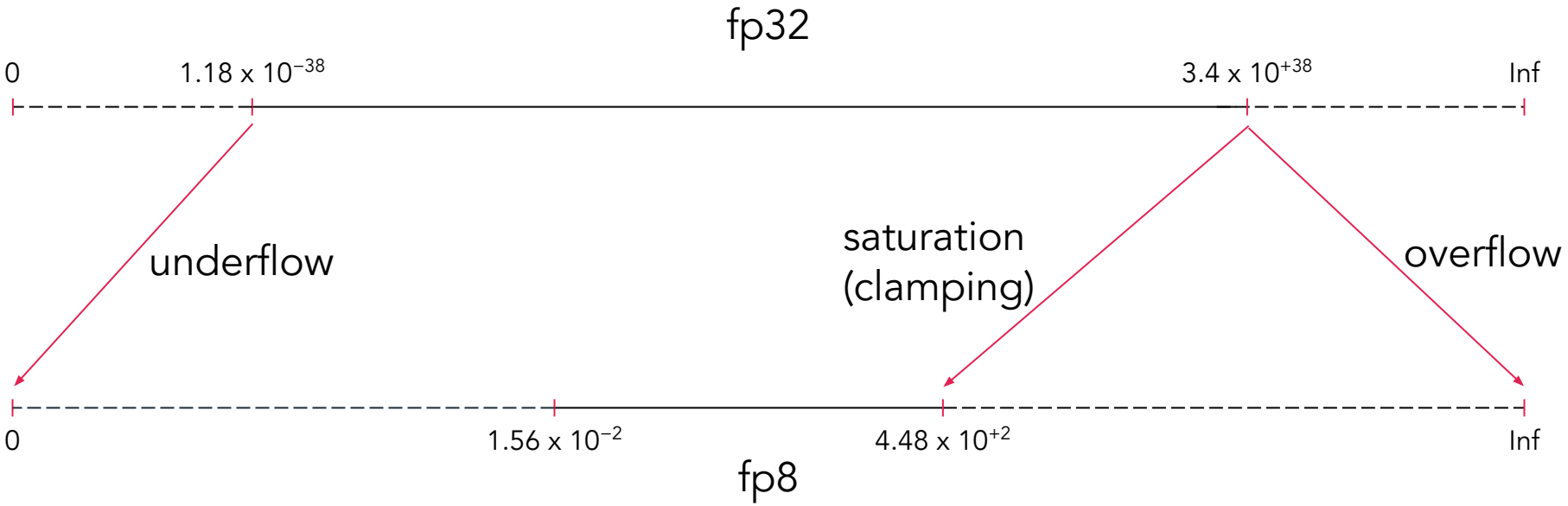
Different bits for different parts



Range vs precision



Overflows and underflows



Priorities

1. Avoid overflows
   If Infs or NaNs appear in the model, training stops

2. Avoid underflows
   If small gradients are rounded to zero, training might not converge

History lesson: fp16 vs bf16

Fp16 suffered these same issues!

Countermeasures included operator allow/blocklist, and gradient scaling

Bf16 was devised to avoid this: same range as fp32, but small precision

What we learned: AI training is robust to loss of precision (SGD can handle noise) but it requires ample range (otherwise signal gets lost)

Scaling

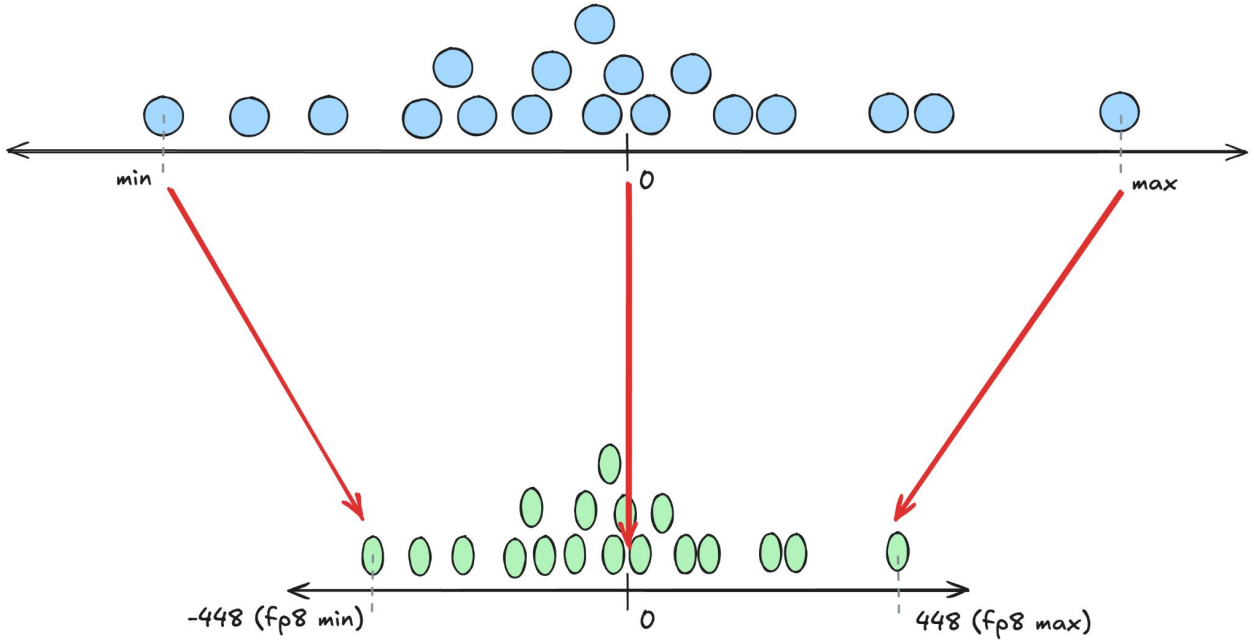
Fp8 needs scaling

We can’t do “bf8”

We need scaling, and need to make it work!

Map range of data to range of fp8

To avoid overflow and maximize range usage

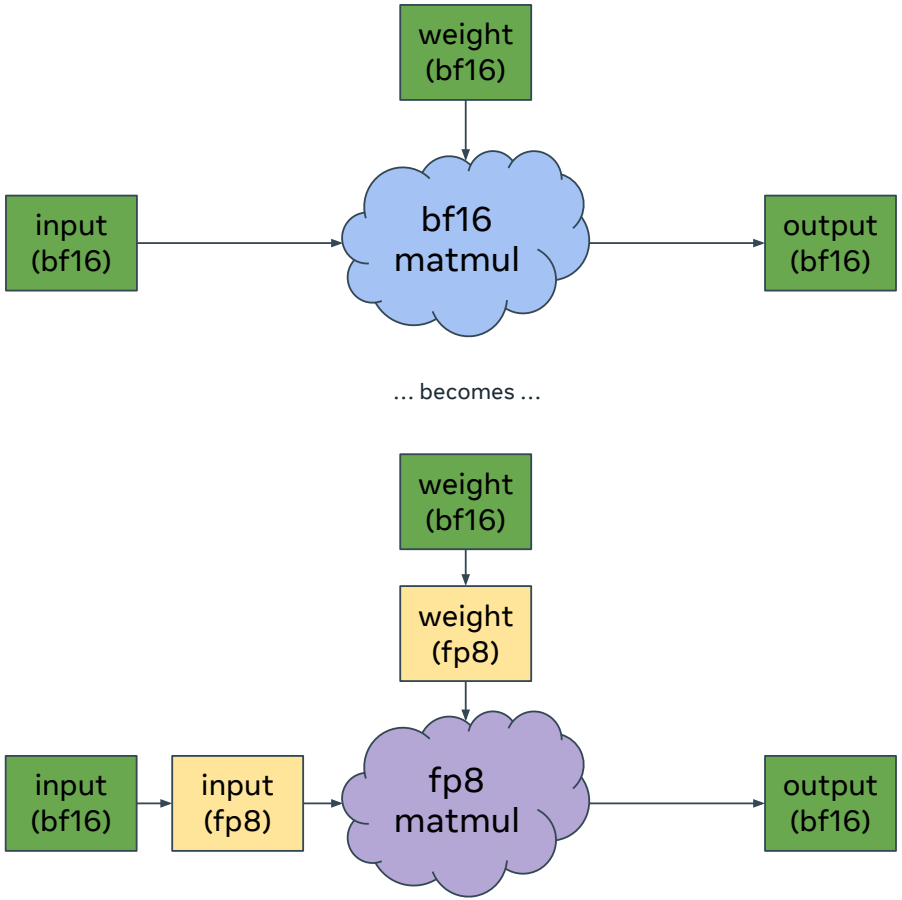


Only scale matmul inputs

The only computation that gets faster with fp8 is the one that goes through TensorCores

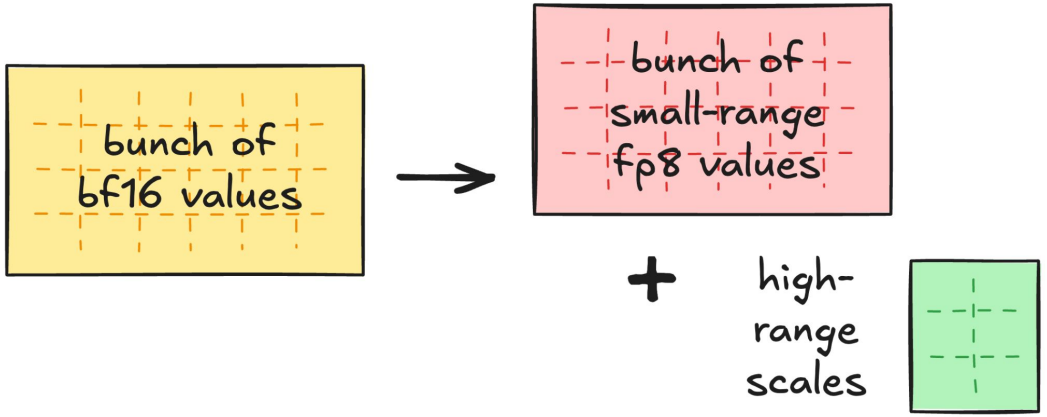
We convert inputs to matmuls

(and maybe self-attention?)



Scales tag along

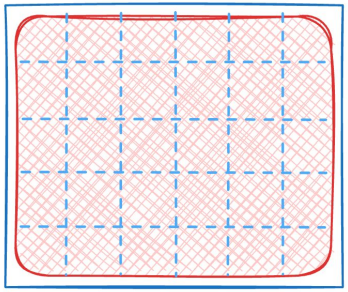
An fp8 “data” tensor must be accompanied by a metadata tensor with the scales

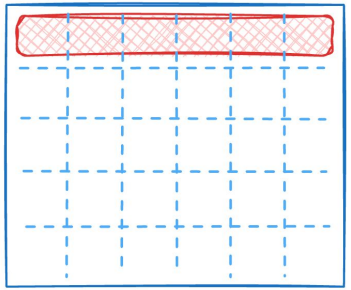


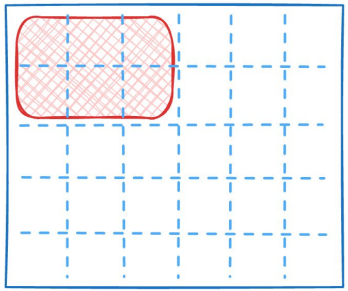
How many scales?

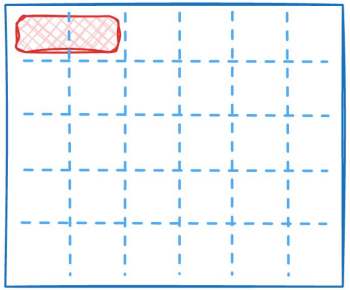
How many values should we pool together and have them share a single scale?

Tensor-wise, row-wise, block-wise, group-wise, channel-wise, microscaled (mxfp8), …









Accuracy

Smaller scaling tiles are more fine-grained and thus more accurate (if we had a separate scale for each element we'd be perfectly accurate)

Matmul overhead

Larger tiles are faster as we amortize the cost of a multiplication over more elements (except on Blackwell which has hardware acceleration)

Quantization overhead

Smaller scales are faster to quantize if they fit in working memory as we can compute statistics and quantize in a single step

Fwd-bwd consistency

Weights might benefit from using square tiles as there are risks associated with underflowing a param in the forward but not in the backward

What dtype for scales

On Hopper, we use float32

On Blackwell, mxfp8 uses the standalone exponent of a float32 (a.k.a., ue8m0)
Obtained by rounding up a float32 scale to the next power of 2

Dynamic or delayed

Quantization requires knowing the abs-max of the tile (a.k.a., “amax”)

Early approaches used to estimate this based on statistics from previous iterations

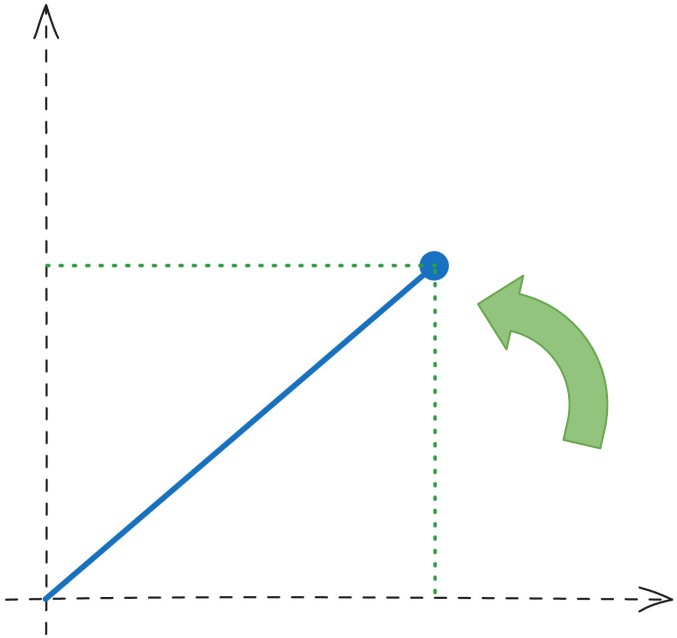
This was complex and prone to overflows

We don’t do this anymore

Rotations to flatten outliers

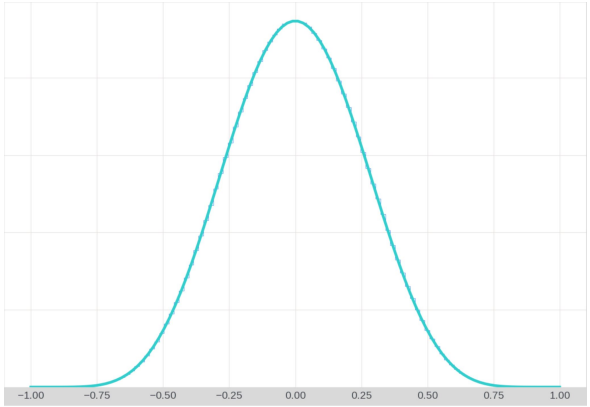
Rotating a vector can help “spread” one coordinate across all other ones

Matmuls only look at the “angle” between rows/columns, thus if both inputs are rotated in the same way they cancel out and nothing changes



Random rotation

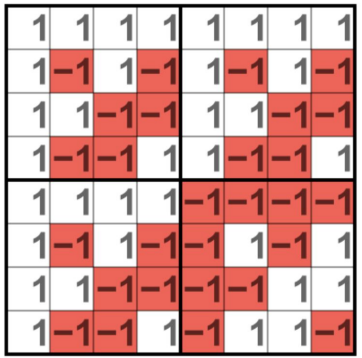
Elements in a random rotation matrix follow a Beta distribution (basically a clamped Gaussian)



Applying the rotation might add too much numerical error and ultimately be net negative

Hadamard rotation

A specific class of rotations consisting only of +1/−1 coefficients, thus easy/cheap to apply



In case of a single extreme outlier, the output might follow a bimodal distribution

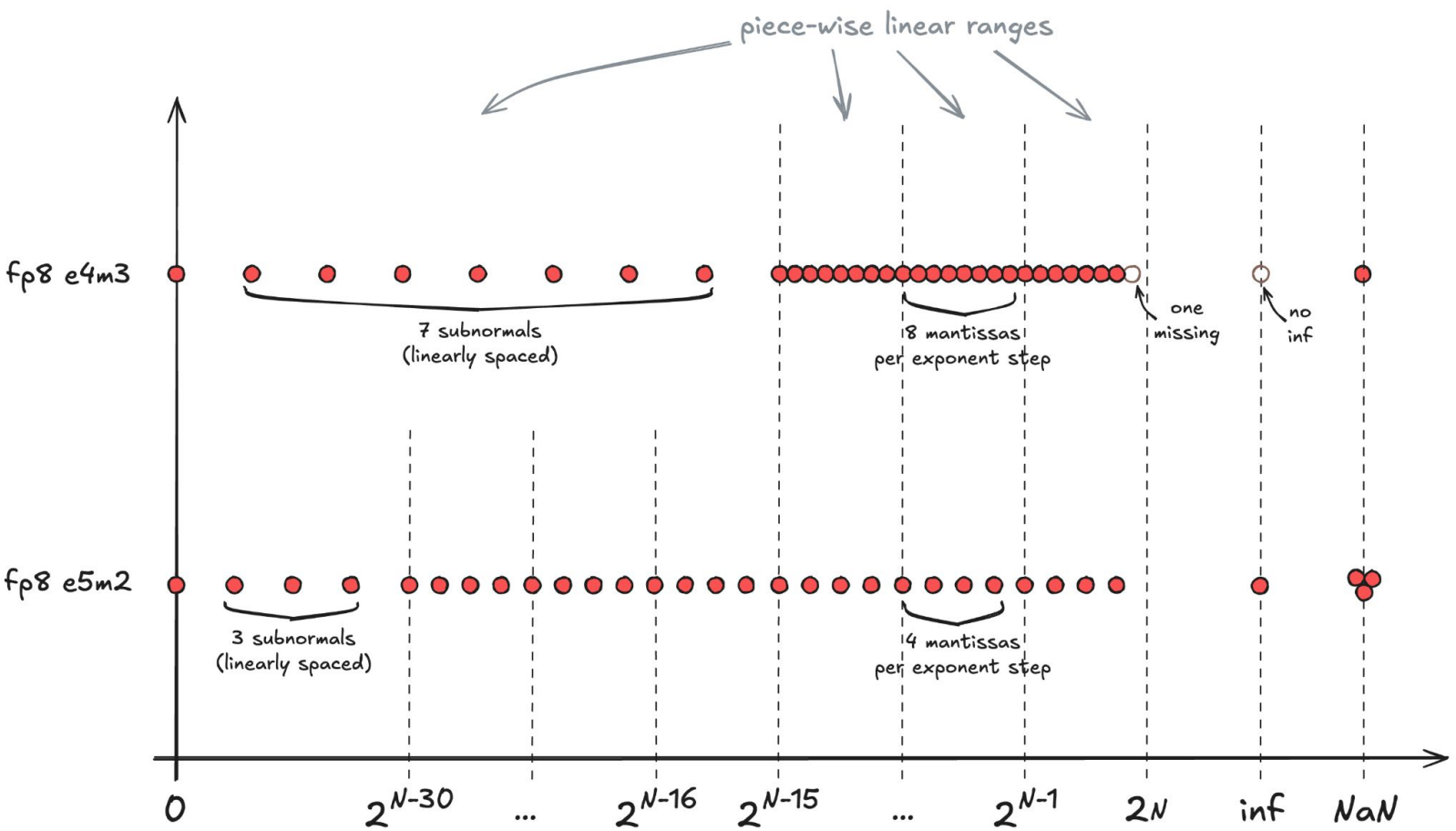
Gotchas

Two dtypes: e4m3 vs e5m2

Different tradeoffs between range and precision

The intention was to use e5m2 for gradients, as they need wider range
However, this worked poorly, and empirically we're better off with full e4m3

This is counterintuitive: we said AI models need range more than precision, but apparently the scaling already provides enough of it, and sacrificing range to add more of it ends up being counterproductive

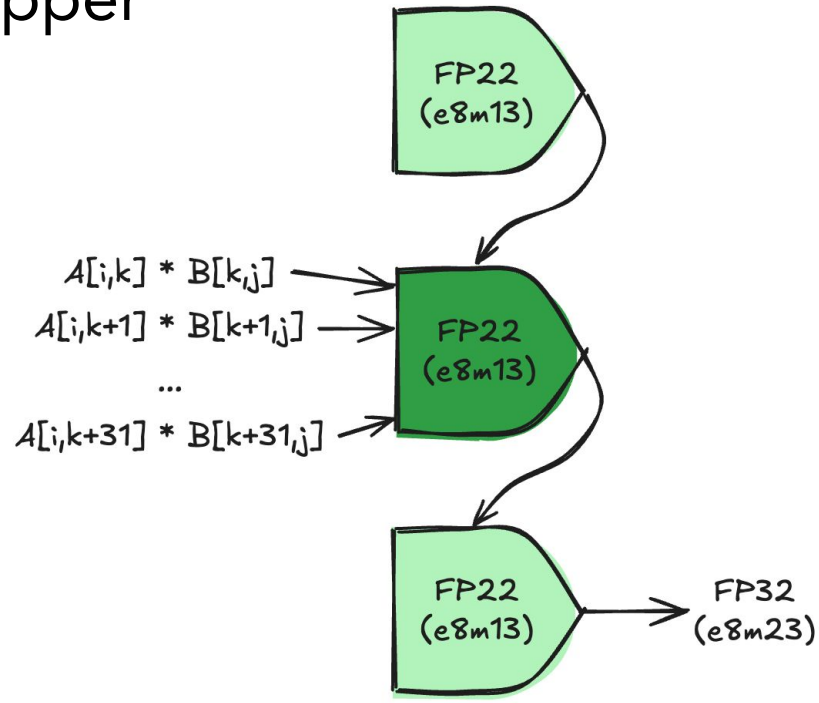

piece-wise linear ranges
fp8 e4m3
7 subnormals
(linearly spaced)
8 mantissas
per exponent step
one
missing
no
inf
fp8 e5m2
3 subnormals
(linearly spaced)
4 mantissas
per exponent step
0
2^N-30
...
2^N-16
2^N-15
...
2^N-1
2N
inf
NaN

Fast accumulation mode on Hopper

"Fast accum" relies on fp8 TensorCore native accumulation, which occurs in "fp22" precision

This is too inaccurate for the bwd pass

Disabling fast accum mitigates this by regularly copying results from the TC to a separate external fp32 accumulator

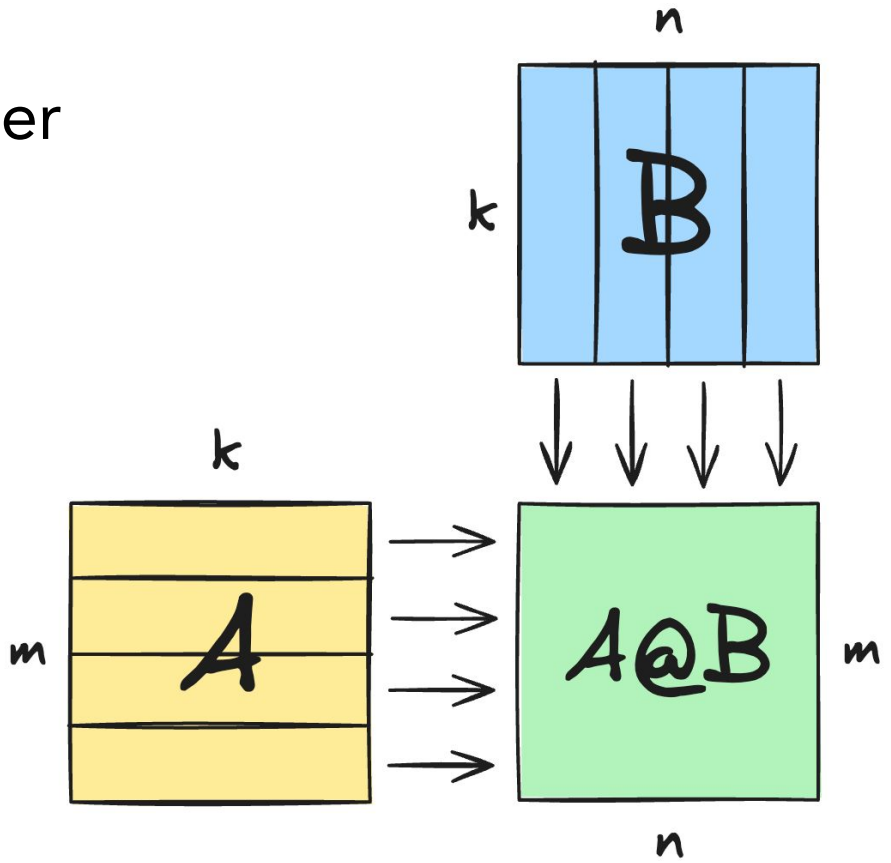


Strict memory layouts on Hopper

Fp8 matmuls need their inputs to be contiguous on the reduction dim

Problem: across fwd+bwd, each of the three matrices (in, w, out.grad) occurs in *both* row- and column-major layouts!

We thus need extra copies, and cannot fuse quantization into previous kernels

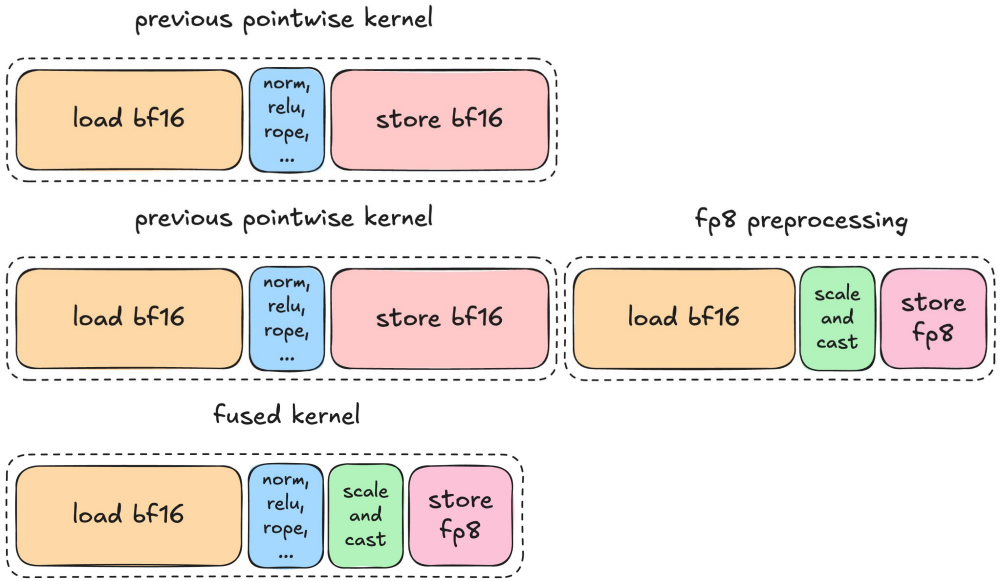


Quantization is best when fused

Bf16 baseline:

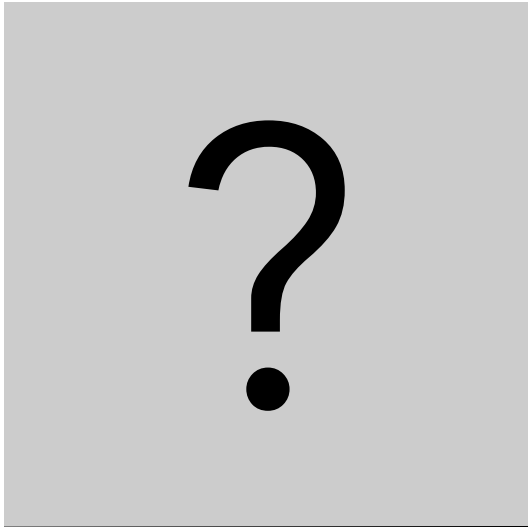
Naive fp8 quantization:

Fused fp8 quantization:

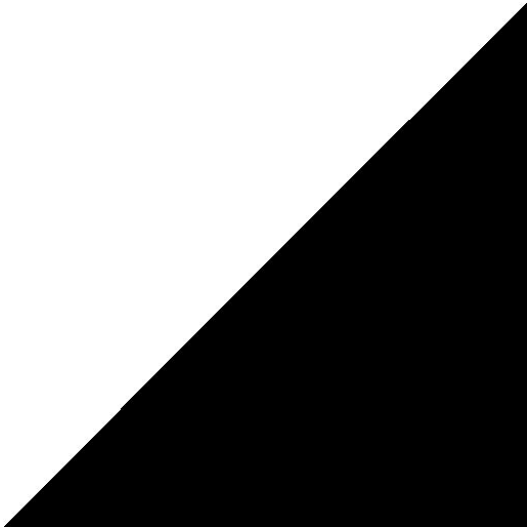


Stochastic rounding

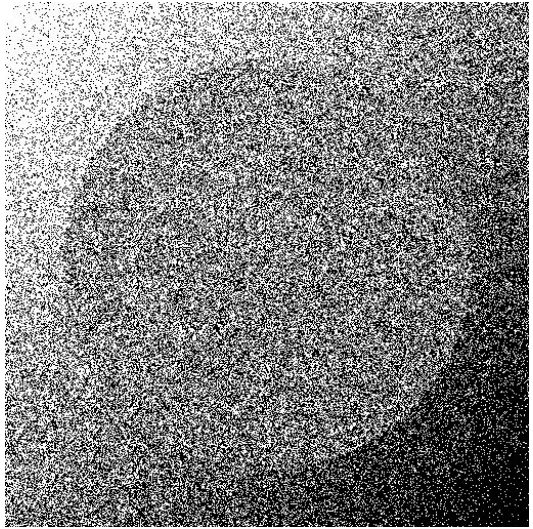
Dithering for floats



Mystery grayscale image



Black/white
round-to-nearest



Black/white
noisy round-to-nearest

Preserving discarded mantissa bits, on average

Goal:

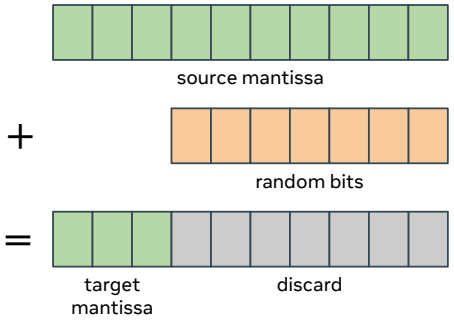
$$E[round(x)] = x$$

For example: if a gradient is very small,
we want to *sometimes* round it a non-zero value

How-to:
- add uniform noise [−0.5, +0.5] × target-ULP
- then round-to-nearest

(equivalent: noise in [0, 1] then round-down)

In theory: uniformly sample as many bits as we drop from mantissa, independently per value



In practice: can use fewer bits, reuse them, …

Back to PyTorch…

How to access all this?

All the fp8 goodies in PyTorch are provided by the TorchAO package

You can look at torchtitan for how to put them together

Thanks