

FlashOptim

Optimizers for Memory-Efficient Training

Jose Javier Gonzalez Ortiz, Abhay Gupta, Christopher Rinard, Davis Blalock**



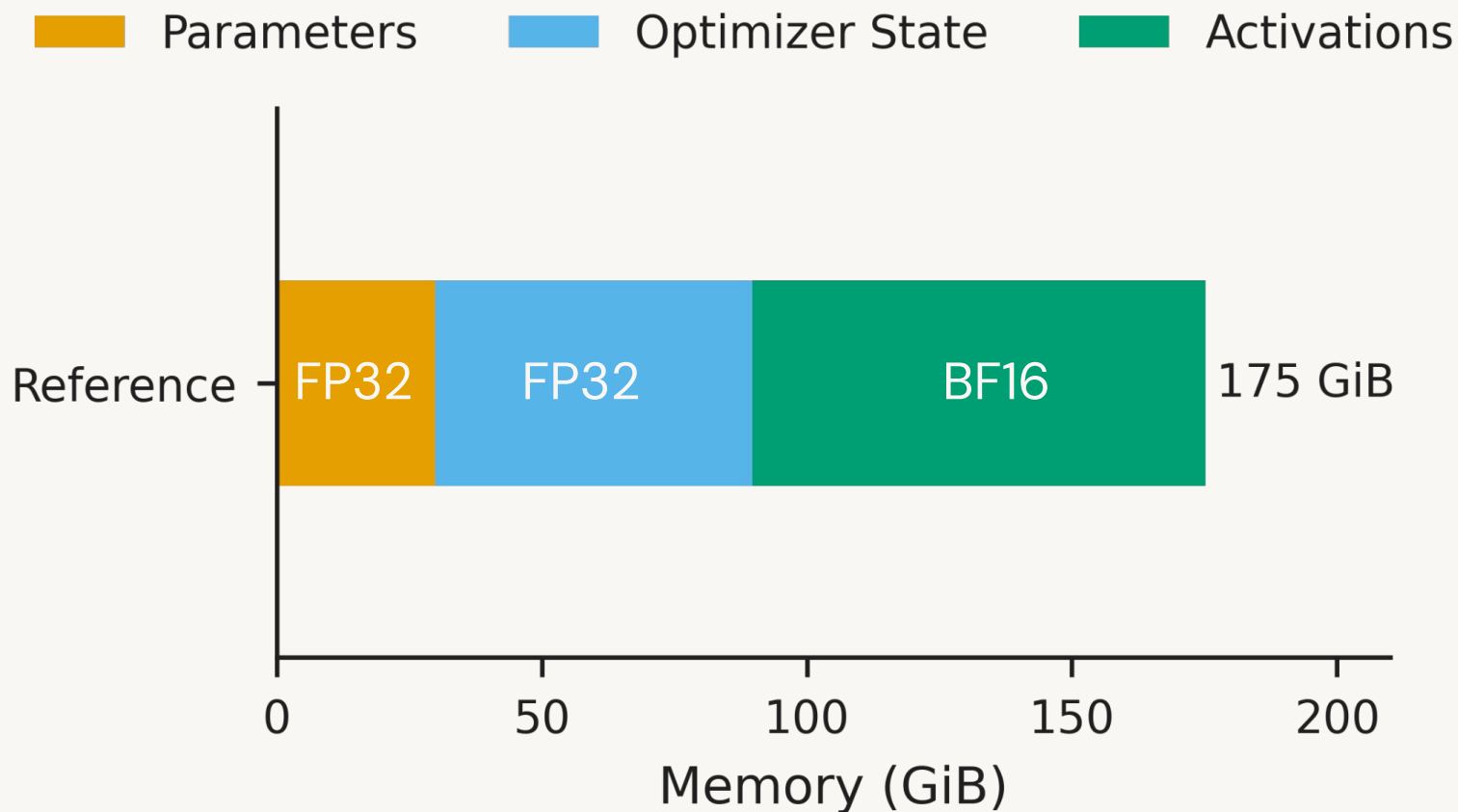
Databricks AI Research



* Equal Contribution

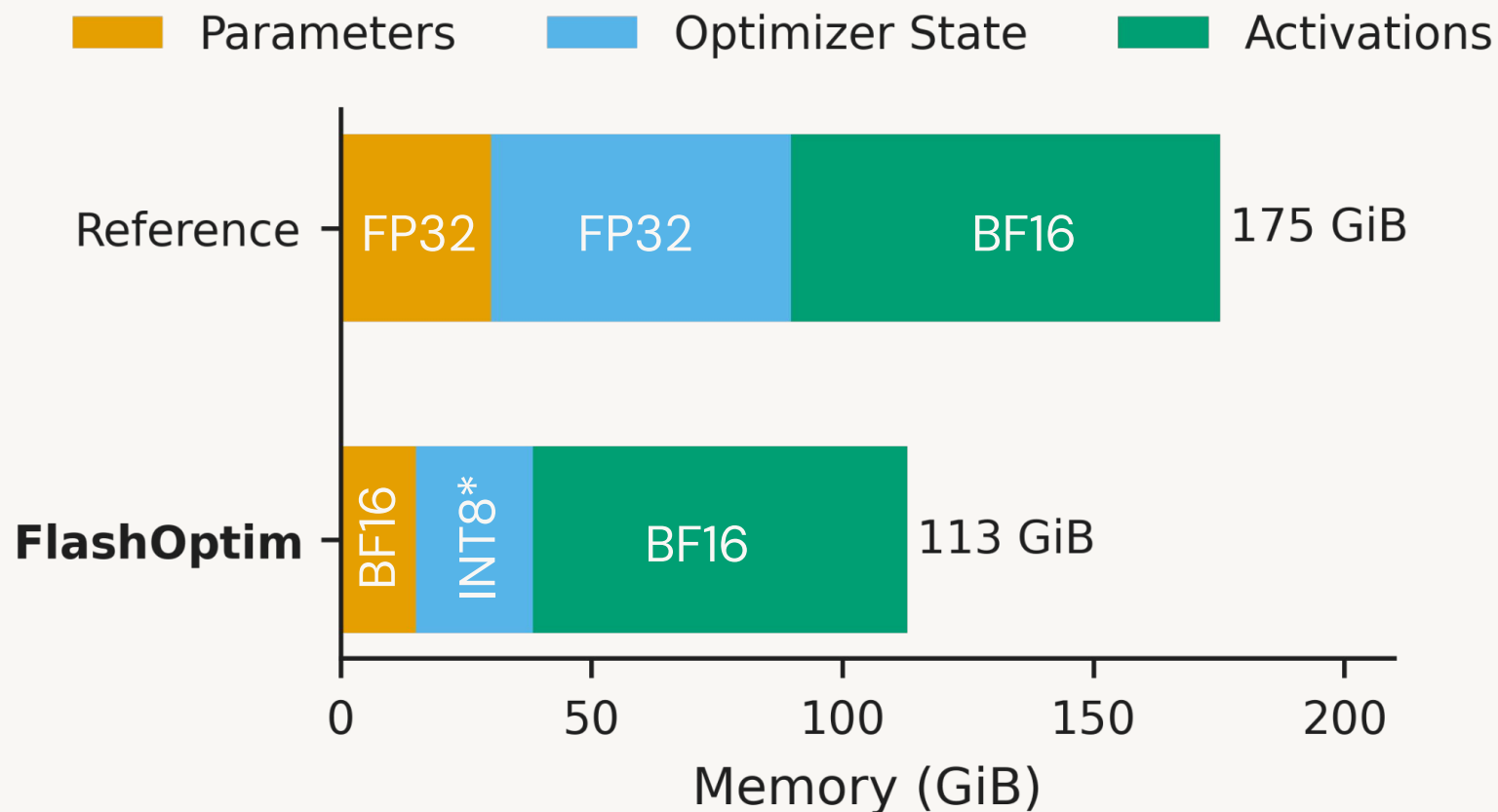
LLM training is really memory intensive

E.g. Training an 8B model (bsz=1, seq=1024, gpus=8) takes 175GiB of GPU memory



FlashOptim reduces training memory

Two key ideas: **Weight splitting** – parameters 32bit → 24bit, and **Companded quantization** – optimizer states 32bit → 8.5bit

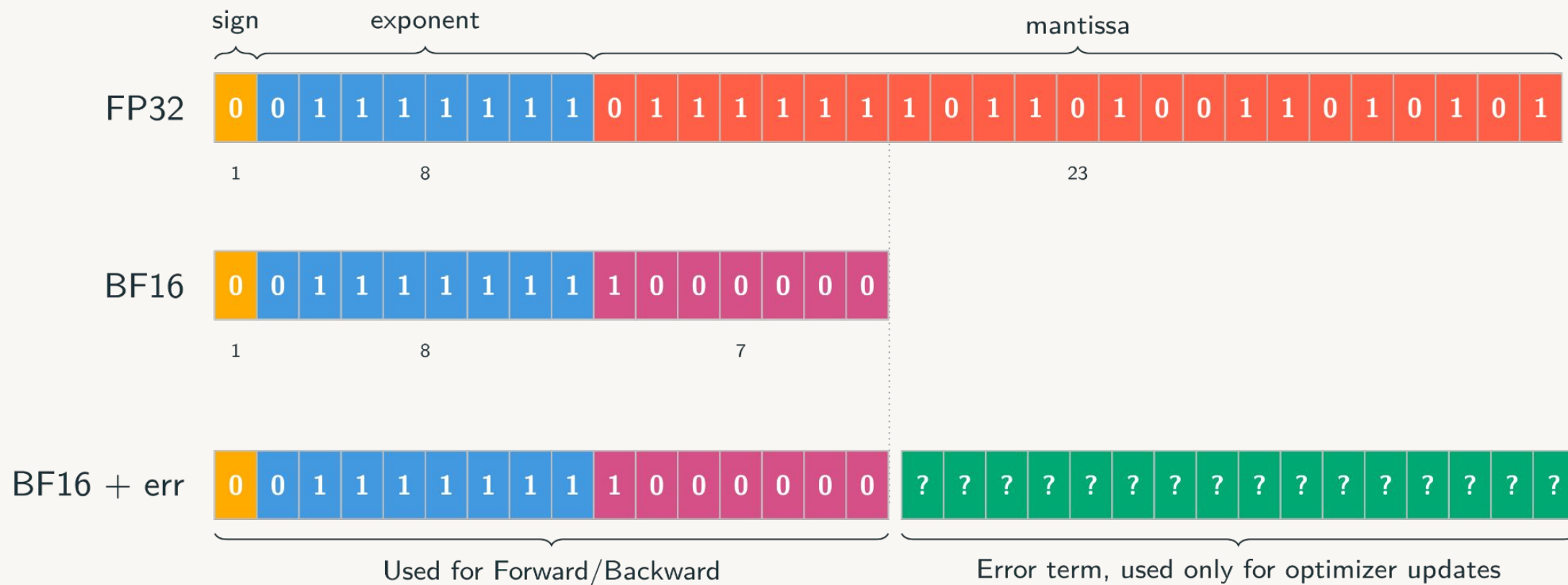


1. Weight Splitting

Reducing parameter precision



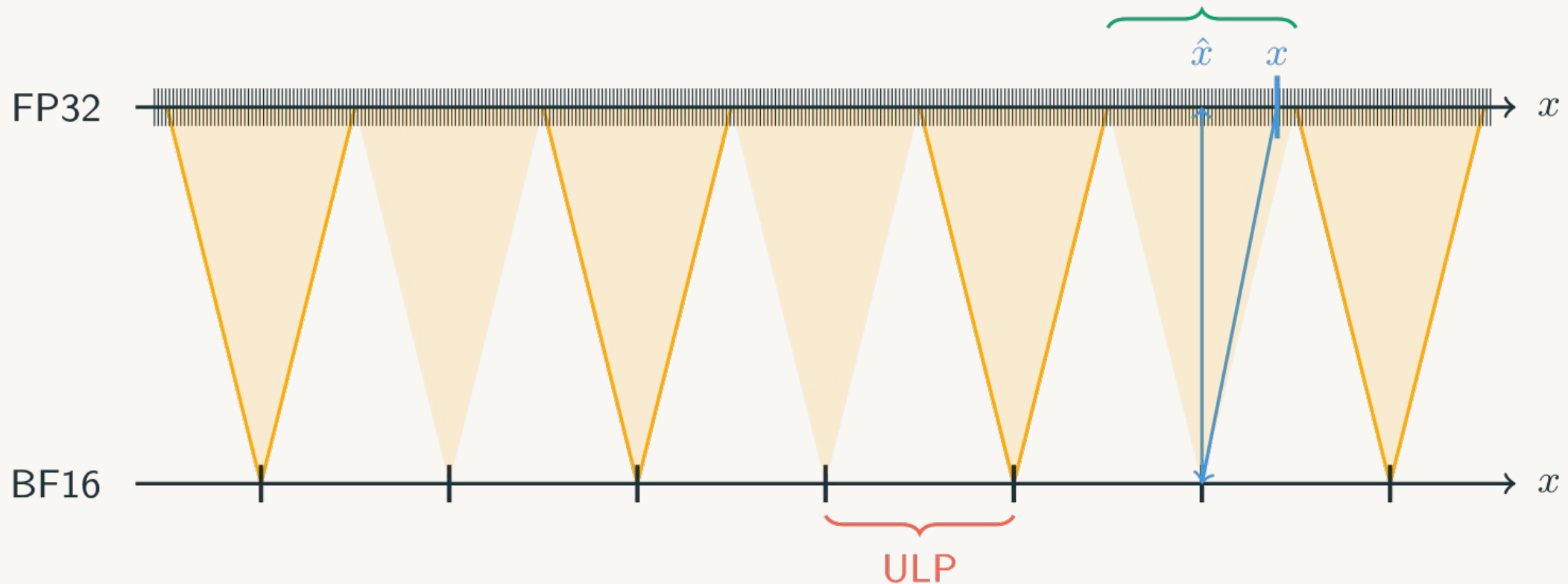
Parameter representation



Downcasting errors

Casting back to FP32 introduces a reconstruction error $e = |x - \hat{x}|$

The error magnitude is bounded by half ULP $e \in \left[-\frac{\text{ULP}}{2}, \frac{\text{ULP}}{2}\right]$



Weight splitting

The distribution of casting errors is uniform, we rescale the error and store as an Integer format (INT8 or INT16) as part of the optimizer state

Splitting $\theta \rightarrow (\theta', \rho)$

$$\theta' = \text{downcast}(\theta)$$

$$\rho = \text{round} \left(\frac{\theta - \theta'}{\text{ULP}(\theta')/2} \cdot N \right)$$

Reconstruction $(\theta', \rho) \rightarrow \hat{\theta}$

$$\hat{\theta} = \theta' + \frac{\rho}{N} \cdot \frac{\text{ULP}(\theta')}{2}$$



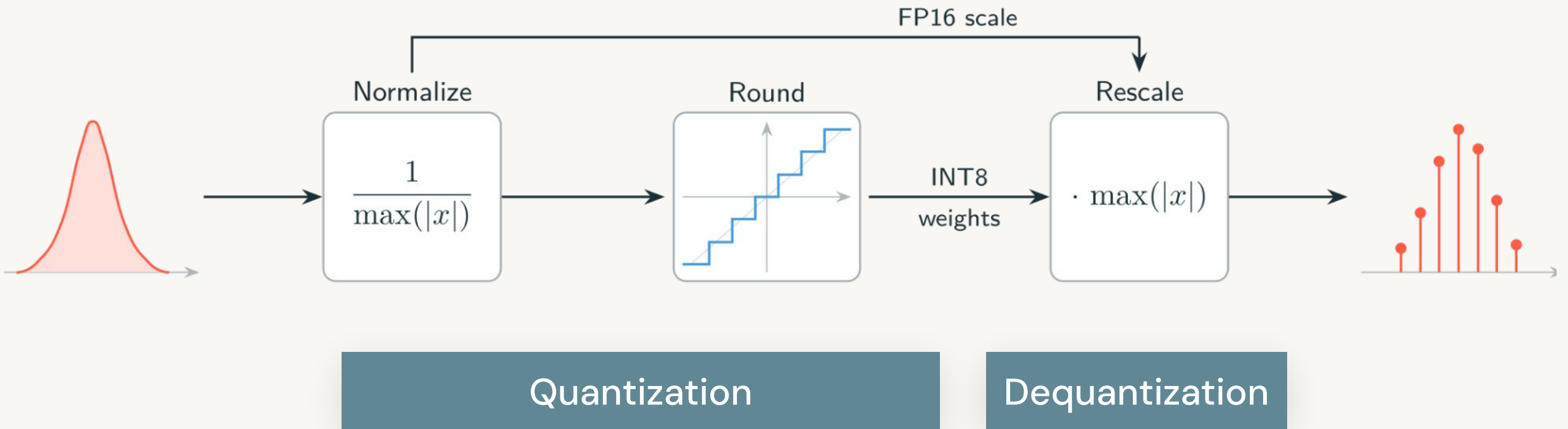
2. Companded Quantization

Reducing optimizer state precision



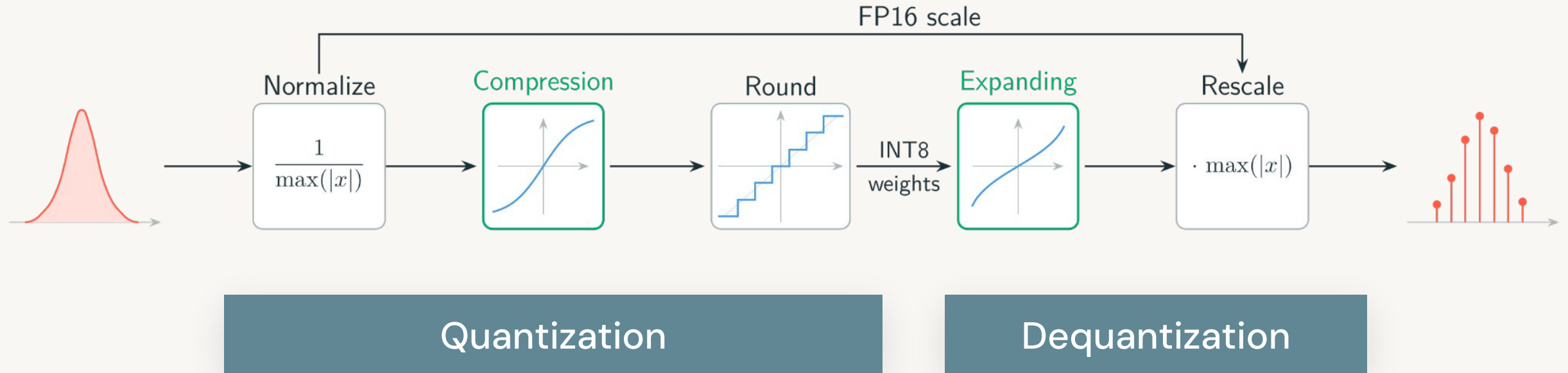
How does grouped quantization work?

Split tensors into groups of size G, normalize and cast to int8 (e.g. G=32)



How does grouped quantization work?

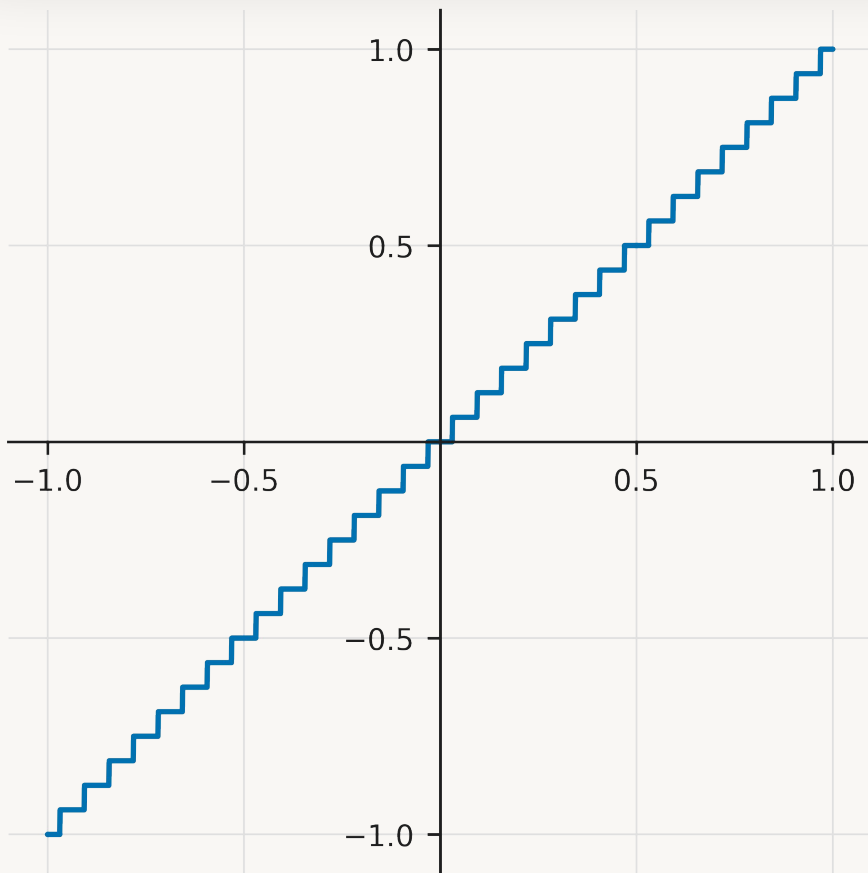
But, linear quantization **assumes uniformly distributed** data – however, optimizer states do not follow this assumption



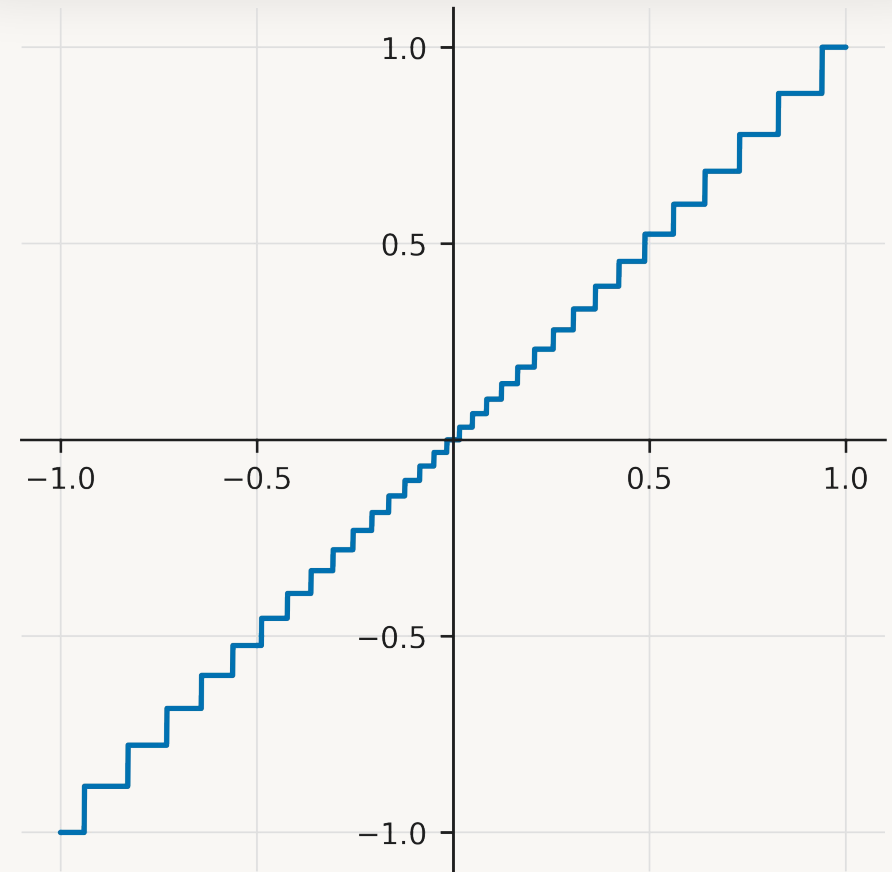
By companding, we are effectively doing **non-uniform rounding**.

We visualize this by plotting $f^{-1}(\text{Round}(f(x)))$

Linear Quantization



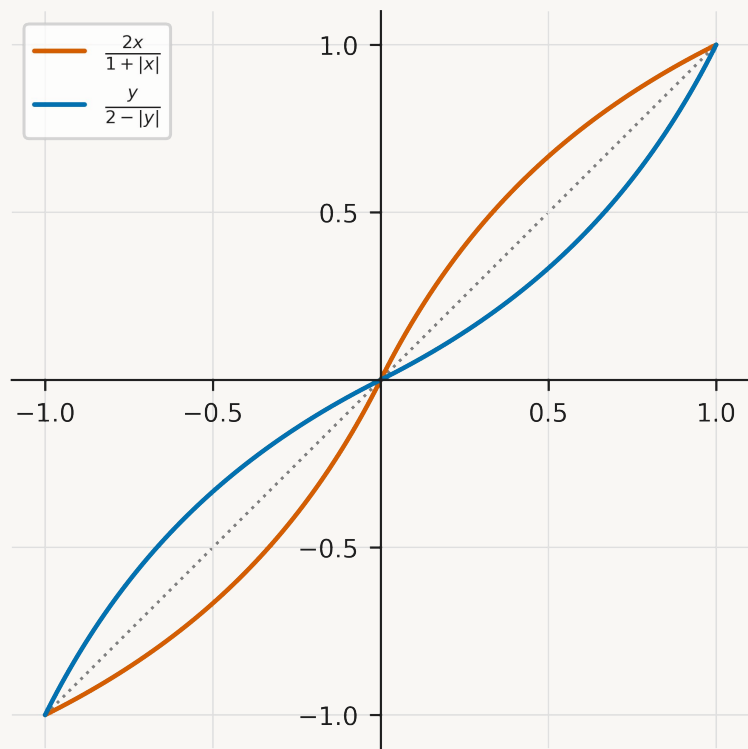
Companded Quantization



We collected lots of FP32 optim states and tested many functions – **these were the best**

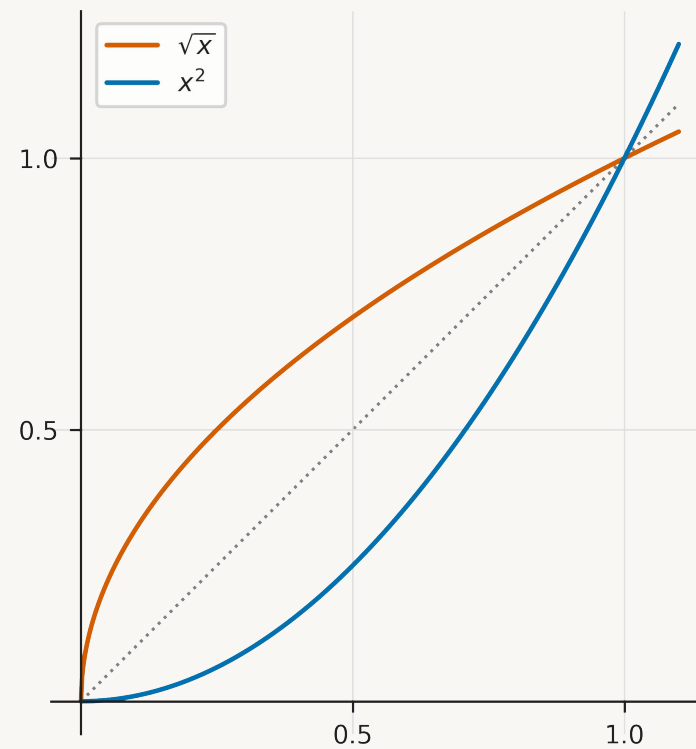
Momentum Companding

$$\phi_m(x) = \frac{2x}{1 + |x|} \quad \phi_m^{-1}(z) = \frac{z}{2 - |z|}$$



Variance Companding

$$\phi_v(x) = \sqrt{x} \quad \phi_v^{-1}(z) = z^2$$



FlashAdamW

To implement both techniques we only need to define a custom optimizer

These steps run as a single fused Triton kernel

Require: Parameters θ_0 , learning rate schedule $\{\eta_t\}_{t=1}^T$, $\beta_1, \beta_2 \in [0, 1)$, $\varepsilon > 0$, weight decay $\lambda \geq 0$, loss $\mathcal{L}(\theta)$, minibatch sampler $\mathcal{B}(\cdot)$

- 1: $m_0^q, m_0^s \leftarrow \mathcal{Q}_m(0)$
- 2: $v_0^q, v_0^s \leftarrow \mathcal{Q}_v(0)$
- 3: $\theta'_0, \rho_0 \leftarrow \mathcal{C}(\theta_0)$
- 4: Initialize $t \leftarrow 0$
- 5: **while** not converged **do**
- 6: $t \leftarrow t + 1$
- 7: $B_t \sim \mathcal{B}$
- 8: $g_t \leftarrow \nabla_{\theta} \mathcal{L}(B_t; \theta'_{t-1})$
- 9: $m_{t-1} \leftarrow \mathcal{Q}_m^{-1}(m_{t-1}^q, m_{t-1}^s)$
- 10: $v_{t-1} \leftarrow \mathcal{Q}_v^{-1}(v_{t-1}^q, v_{t-1}^s)$
- 11: $\theta_{t-1} \leftarrow \mathcal{C}^{-1}(\theta'_{t-1}, \rho_{t-1})$
- 12: $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
- 13: $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
- 14: $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$
- 15: $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$
- 16: $\theta_t \leftarrow \theta_{t-1} - \eta_t (\hat{m}_t / (\sqrt{\hat{v}_t} + \varepsilon) + \lambda \theta_{t-1})$
- 17: $m_t^q, m_t^s \leftarrow \mathcal{Q}_m(m_t)$
- 18: $v_t^q, v_t^s \leftarrow \mathcal{Q}_v(v_t)$
- 19: $\theta'_t, \rho_t \leftarrow \mathcal{C}(\theta_t)$

Reconstruct optimizer state and master weights

Standard AdamW update

Quantize optimizer state and split master weights

Experimental Results

Tasks

- **Vision Pretraining** – ResNet-50 on ImageNet ILSVRC2012
- **LLM Pretraining** – GPT-2 on FineWeb-10B, eval with ICLs (HellaSwag, Arc, &c)
- **LLM Finetuning** – Llama-3.1-8B on OpenMathInstruct-2, eval with GSM8k

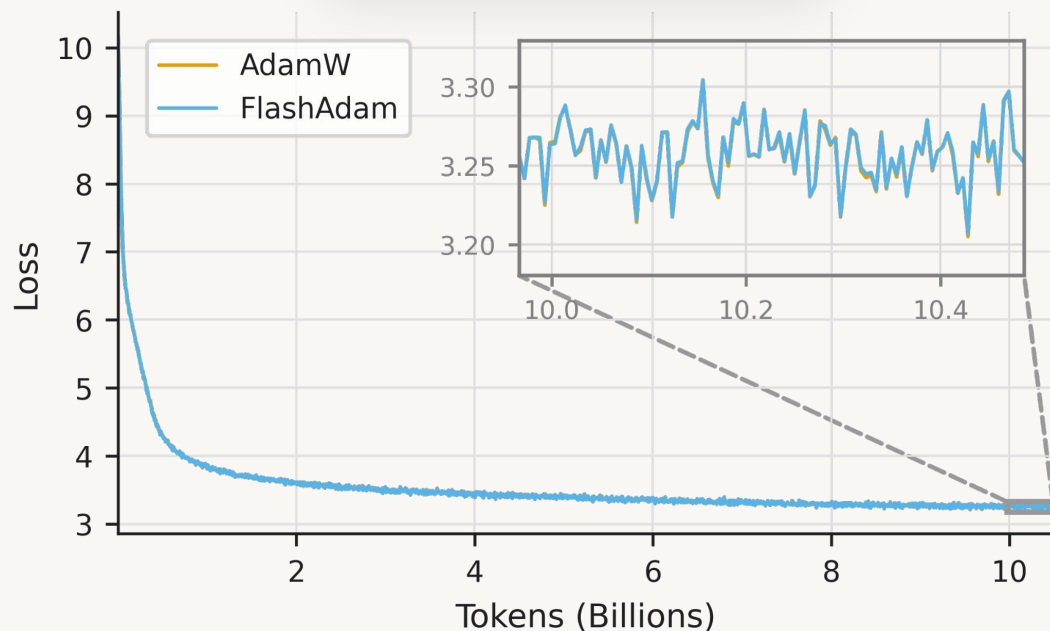
Optimizers – AdamW, Lion, SGD



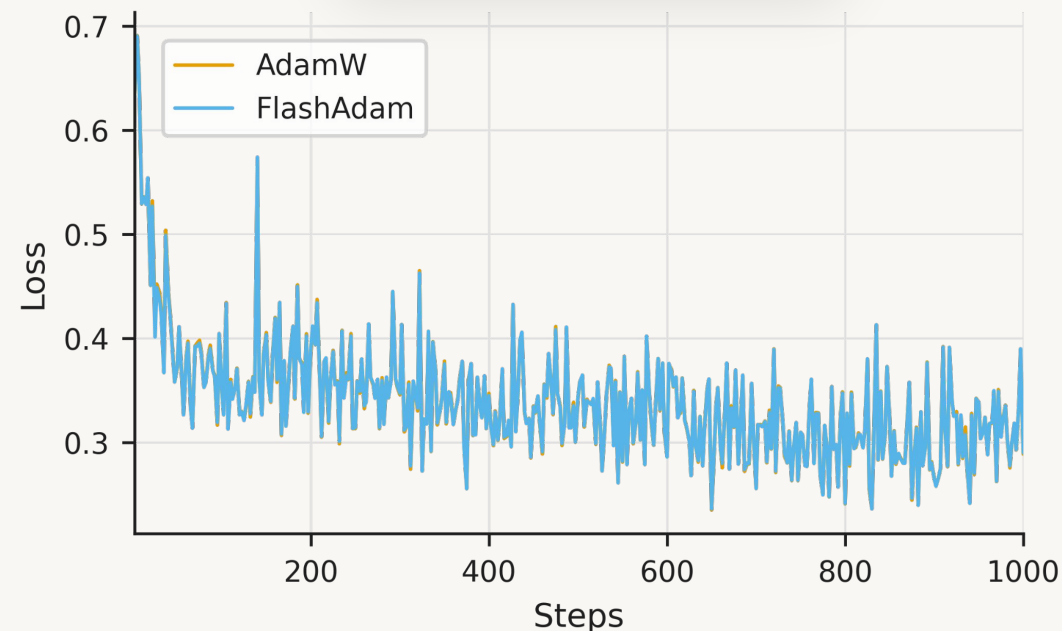
Convergence

Models **converge identically** despite the reduced precision in weights (-25%) and optim states (-75%)

LLM Pretraining



LLM Finetuning



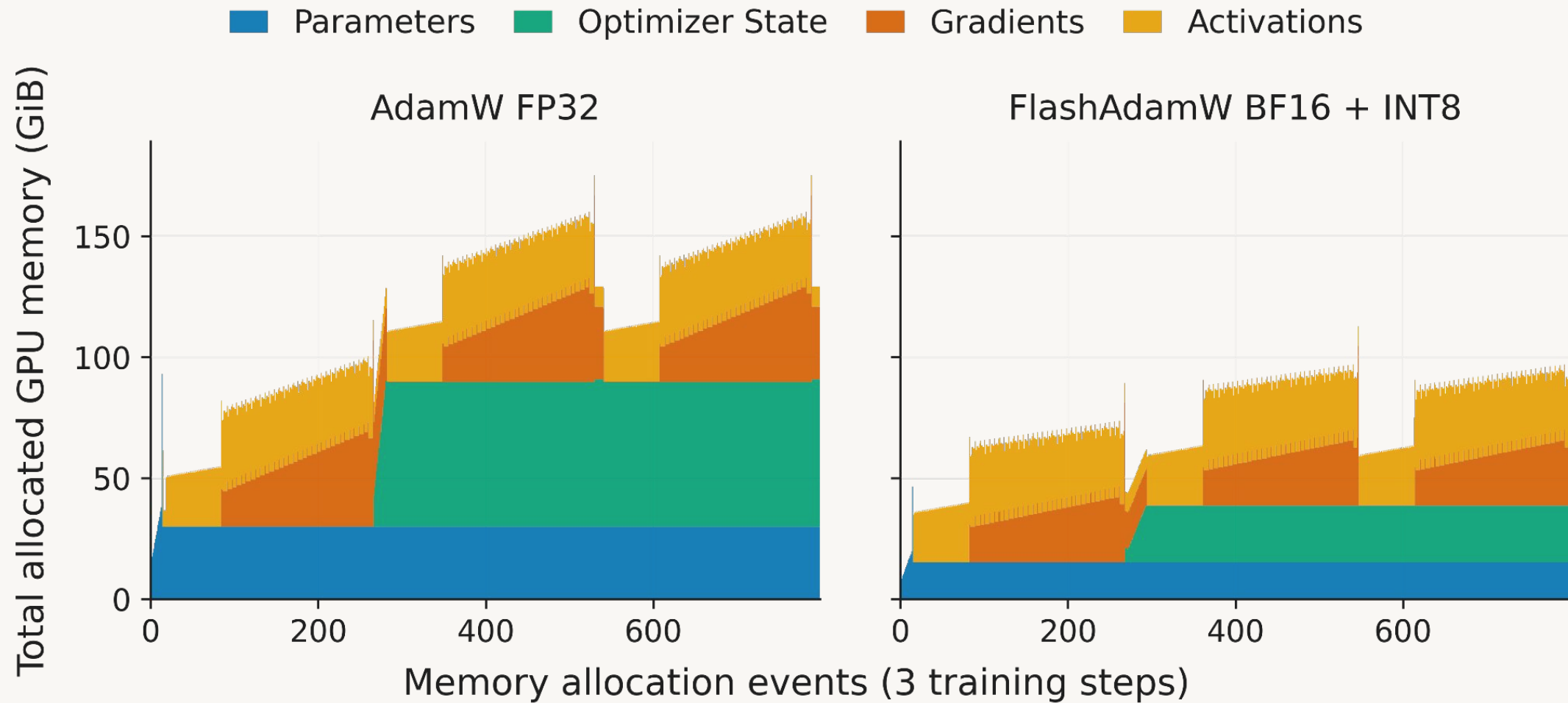
Accuracy

Models trained with
FlashOptim **match the**
reference optimizers
in all settings

Task	Optimizer	Score
Vision Classification	SGD	77.01 $\pm$ 0.02
	FlashSGD	77.16 $\pm$ 0.09
	AdamW	75.51 $\pm$ 0.09
	FlashAdamW	75.67 $\pm$ 0.04
LLM Pretraining	AdamW	44.0 $\pm$ 0.4
	FlashAdamW	45.0 $\pm$ 1.0
	Lion	44.0 $\pm$ 0.5
	FlashLion	44.7 $\pm$ 0.5
LLM Finetuning	AdamW	75.1 $\pm$ 0.4
	FlashAdamW	75.0 $\pm$ 0.8



Memory trace



Memory (Llama-3.1-8b)

We get **50%** param associated memory reduction, **36%** peak memory reduction, and **57%** checkpoint size reduction – with no slowdown in optim step times

Variant	Params		Optim		Peak		Step
	GiB	Δ	GiB	Δ	GiB	Δ	ms
Reference	29.9		59.8		175.2		10.2
FlashOptim	15.0	-50%	23.4	-61%	112.7	-36%	9.4
Kahan Splitting (Zamirai, 2020)	15.0	-50%	44.9	-25%	134.2	-23%	15.8
8-bit AdamW (Dettmers, 2022)	29.9		15.2	-75%	130.6	-25%	31.9



TL;DR

FlashOptim reduces peak training memory at no quality cost

- General mechanism for splitting weight precision
- Better quantization transforms for optimizer states
- No efficiency/quality tradeoff
- Fused kernels introduce no wall-clock overhead
- Tested in distributed training (FSDP)

Source code and paper available at

<https://github.com/databricks/flashoptim>

