FlashOptim

Optimizers for Memory Efficient Training

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

databricks

* Equal Contribution

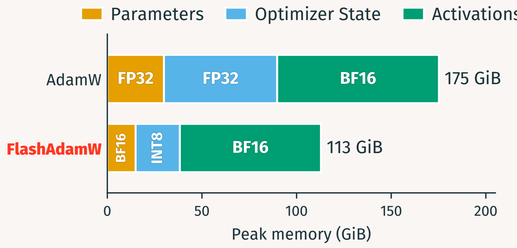
FlashOptim reduces per-parameter training memory by >50% while preserving training convergence, model quality, and optimizer API compatibility.

We open source our fused kernel implementations for AdamW, SGD, and Lion.

1 Motivation

Large model training requires vast amounts of accelerator memory to store parameters, gradients, optimizer states, and activations.

Low-precision methods can save model memory but often worsen model quality or slow down training.



2 Approach

FlashOptim compresses master weights **32 → 24 bit** and optimizer states **32 → 8.5 bit**.

We introduce novel techniques to minimize weight compression error and optimizer quantization error.

1 Reconstruct

$m \leftarrow Q_m^{-1}(m^q, s_m)$
$v \leftarrow Q_v^{-1}(v^q, s_v)$
$\theta \leftarrow C^{-1}(\theta', \rho)$

Dequantize optim states & reconstruct parameters.

2 Standard update

$m_t = \beta_1 m + (1-\beta_1) g_t$
$v_t = \beta_2 v + (1-\beta_2) g_t^2$
$\theta_t = \theta - \eta\, \Delta_t$

Run optimizer update in transient full precision.

3 Compress back

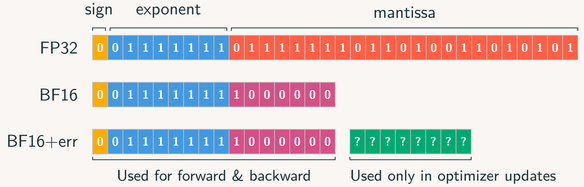
$m_t^q, s_{m,t} \leftarrow Q_m(m_t)$
$v_t^q, s_{v,t} \leftarrow Q_v(v_t)$
$\theta_t', \rho_t \leftarrow C(\theta_t)$

Quantize optim states & split parameters.

For efficiency, these operations are **fused** into the **optimizer update kernels** as a prologue and epilogue.

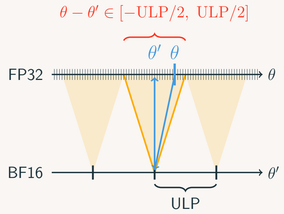
3.1 Method: Weight Splitting

We split the parameters into two terms: a BF16 term used for forward & backward, and an **error term only used in optimizer updates**.



Key insight – We know the **maximum error of downcasting** the weight given its value.

We compute the scale of the error from the floating-point value of the low-precision weight.



Weights are reconstructed to FP32 before the optimizer update and **split before writing them back to memory**. Errors are rescaled and stored using integer datatypes.

Weight Splitting

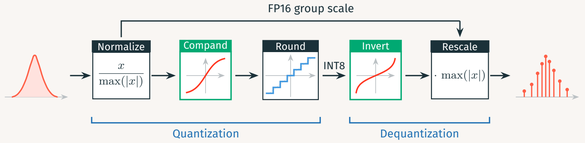
$$\theta' = \text{downcast}(\theta)$$
$$\rho = \text{round}\left(\frac{\theta - \theta'}{\text{ULP}(\theta')/2} \cdot N\right)$$

Weight Reconstruction

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

3.2 Method: Companded Quantization

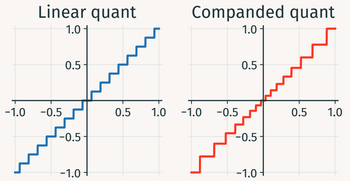
Naively applying grouped quantization to optimizer states incorrectly **assumes uniformly distributed** data.



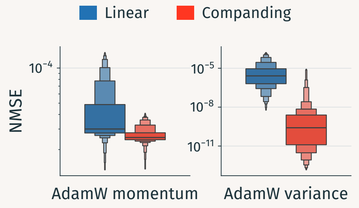
Companding – We apply fast and invertible functions to perform **non-uniform quantization** of optimizer states.

We tune companding functions for momentum and variance tensors based on data from training runs.

$$\phi_m(x) = \frac{2x}{1+|x|}$$
$$\phi_v(x) = \sqrt{x}$$



Quantizing with companding **consistently reduces quantization errors** across all tasks and optimizers.

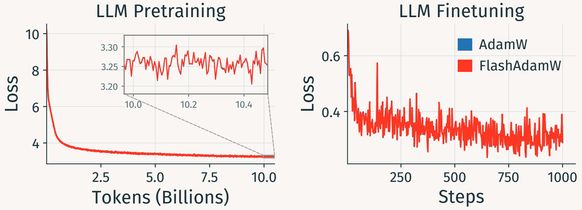


4 Results

Efficiency – We get 50% parameter associated memory reduction, **36% peak memory reduction**, and 57% checkpoint size reduction – with **faster wall-clock step times** from smaller memory reads and writes.

Method	Memory (GiB)			Step (ms)
	Params	Optim	Peak	
AdamW	29.9	59.8	175.2	10.2
Kahan Splitting	15.0 -50%	44.9 -25%	134.2 -23%	15.8
8-bit AdamW	29.9	15.2 -75%	130.6 -25%	31.9
FlashAdamW	15.0 -50%	23.4 -61%	112.7 -36%	9.4

Convergence – In all tested settings, **models converge and evaluate identically** despite the reduced precision in weights and optimizer states.



Stability – FlashOptim **improves training stability** compared to existing low precision methods.