Scalable AI: Bridging Theory, Understanding, and Practice

Fall 2026

UC Berkeley

Lecture 10: Modern Optimizers and Architectures

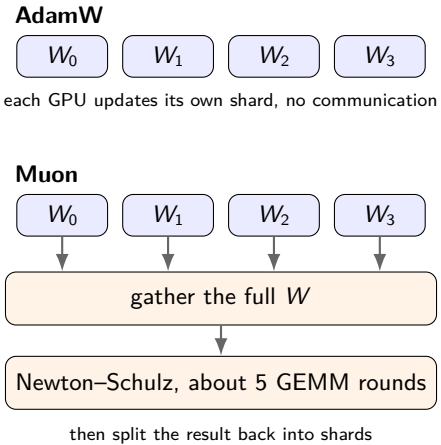
Muon at scale, stable optimizers, evaluating optimizer claims, and memory-aware architecture

Fall 2026

Two changes we study later today

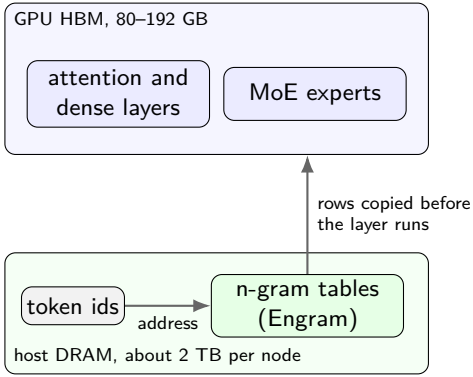
Muon at scale

updating a weight matrix split across GPUs



Architecture and memory traffic

where parameters are stored



Roadmap for today

1. From AdamW to Muon	What each update computes, how Newton–Schulz is tuned, and what it costs on one GPU
2. Muon at scale	Sharded weights, owners, kernels, and Shampoo-style amortization
3. Controlling scale	Bounding weights, attention logits, and activations, and optimizers that fix the weight norm (MuonH, AdamH)
4. Evaluating optimizers	How to read a speedup claim, and a change that holds up (MuonH at 1.2B)
5. Architecture and memory traffic	N-gram memory in host DRAM, wider residual streams

1. From AdamW to Muon

What each update computes and how Newton–Schulz is tuned

An AdamW step is elementwise and bandwidth-bound

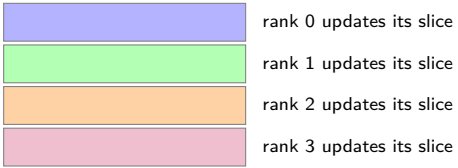
$$\theta \leftarrow \theta - \eta\left(\frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon} + \lambda\theta\right)$$

Each parameter is updated from its own gradient, momentum, and variance. No entry looks at any other entry.

Per parameter	**AdamW**
Bytes read and written	≈ 28
FLOPs	≈ 15
Arithmetic intensity	≈ 0.5 FLOP/byte
H100 ridge point	≈ 300 FLOP/byte

FP32 master weights, m, v; BF16 gradient. The ridge point is peak FLOPs over HBM bandwidth (lecture 2).

ZeRO / FSDP shards of W



Because the update is elementwise, any way of slicing the parameters works. ZeRO and FSDP (lecture 5) give each rank a slice and the optimizer step needs no communication of its own.

A 1B-parameter AdamW step moves $\approx$28 GB, which takes a few milliseconds of HBM time.

Muon replaces each update matrix with its nearest orthogonal matrix

For a weight matrix $W \in \mathbb{R}^{A \times B}$ with gradient G:

$$M \leftarrow \mu M + G$$
$$O = \mathrm{NS}(M) \approx UV^\top \quad \text{where } M = U\Sigma V^\top$$
$$W \leftarrow W - \eta\big(0.2\sqrt{\max(A,B)}\, O + \lambda W\big)$$

Muon keeps the directions U, V of the momentum and sets every singular value to 1. Adam rescales each coordinate separately. Muon rescales the matrix along its singular directions.

The factor $0.2\sqrt{\max(A,B)}$ gives the update the same RMS as a typical AdamW update, so AdamW's learning rate and weight decay can be reused.

Only hidden 2D weight matrices use Muon. Embeddings, the output head, norms, and biases stay on AdamW.

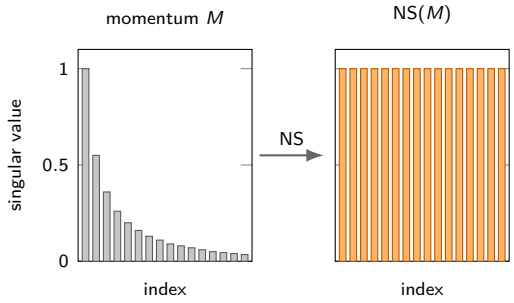
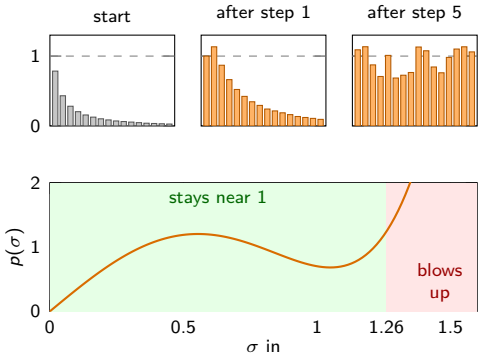


Illustration. Gradients of weight matrices are dominated by a few directions. Orthogonalizing gives the rare directions the same step size as the common ones.

Jordan et al., Muon (2024); Liu et al., Muon is Scalable for LLM Training (arXiv:2502.16982) §2.2.

Newton–Schulz orthogonalizes using only matrix multiplies



The top row is the spectrum from the previous slide flattening toward 1. The bottom plot is one step of the polynomial. Inputs above 1.26 grow without bound, so X is divided by its norm first.

If $X = U\Sigma V^\top$, then $XX^\top X = U\Sigma^3 V^\top$. Multiplying by $XX^\top$ raises every singular value to a higher power and leaves U and V unchanged. One step

$$X \leftarrow aX + b\,(XX^\top)X + c\,(XX^\top)^2 X$$

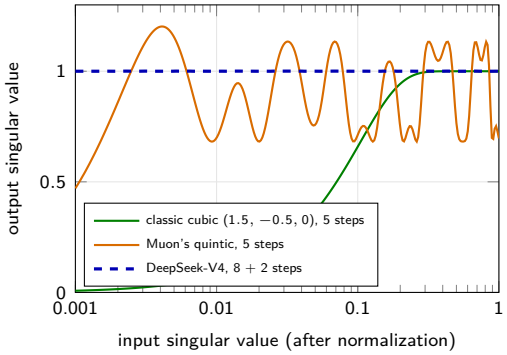
therefore applies $p(\sigma) = a\sigma + b\sigma^3 + c\sigma^5$ to every singular value at once, without an SVD. Dividing by the norm first puts every σ in $[0, 1]$.

$A = XX^\top$ → $B = bA + cA^2$ → $X \leftarrow aX + BX$

Each step is three BF16 GEMMs, the operation GPUs are built for. For an $m \times n$ matrix ($m \le n$) this costs $O(m^2 n)$ FLOPs per step, where AdamW costs $O(mn)$.

Spectra computed from Muon's coefficients $(3.4445, -4.7750, 2.0315)$. github.com/KellerJordan/Muon, muon.py.

Newton–Schulz coefficients trade speed for accuracy



Each curve shows where a singular value ends up after all steps. A perfect orthogonalization would output 1 for every input.

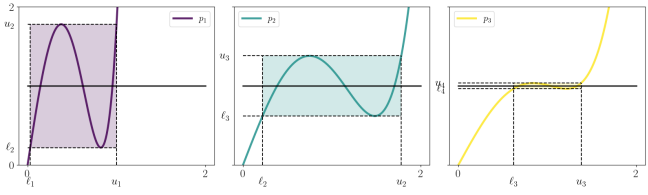
The classic cubic converges to exactly 1 but only lifts small singular values slowly.

Muon's tuned quintic has a large slope near 0, so small singular values grow fast. After 5 steps they land anywhere in about $[0.5, 1.2]$. At small scale this noise does not hurt the loss.

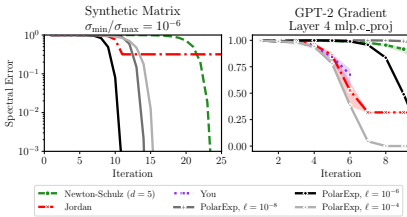
DeepSeek-V4 runs 8 steps of the tuned quintic, then 2 steps of $(2, -1.5, 0.5)$, a polynomial whose fixed point at 1 is flat. The output is 1 to within 10^{-3}. It spends twice the GEMMs on precision.

Curves computed from the coefficients. Jordan et al., Muon (2024); DeepSeek-V4 (arXiv:2606.19348) §2.4, Alg. 1.

Polar Express chooses new coefficients for every step



The first three polynomials. Each maps the current interval of singular values $[\ell_k, u_k]$ to a narrower one around 1.



Error vs iteration. The fixed quintic (red) stops improving at ≈0.3.

Amsel et al., The Polar Express (arXiv:2505.16932) Figs. 2–3, §3.4.

If you know the range of singular values going in, you can pick the polynomial that shrinks it the most at each step. Polar Express solves this minimax problem once, offline, and hard-codes a short list of coefficients.

Early polynomials are aggressive (first step $\approx (8.29, -23.6, 17.3)$), and the last ones are gentle and settle at 1.

Evaluating $p(x/1.01)$ instead of $p(x)$ keeps rounding in BF16 from pushing values out of the interval. The cost per step is the same three GEMMs.

Shampoo, SOAP, and Kron are the other matrix optimizers

Shampoo keeps two running statistics per weight matrix and preconditions on both sides:

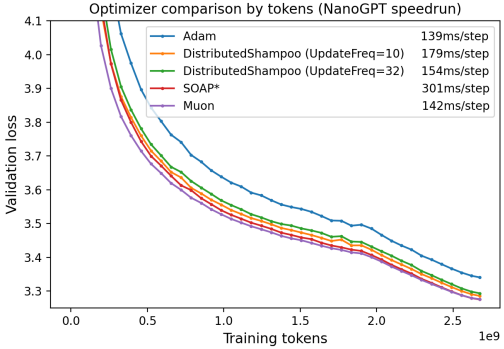
$$L \leftarrow L + GG^\top, \qquad R \leftarrow R + G^\top G$$
$$W \leftarrow W - \eta\, L^{-1/4}\, G\, R^{-1/4}$$

With no accumulation, $L = GG^\top$ and $R = G^\top G$, and the update direction is exactly $UV^\top$. Muon is Shampoo that forgets past gradients, which lets it replace the matrix roots with Newton–Schulz.

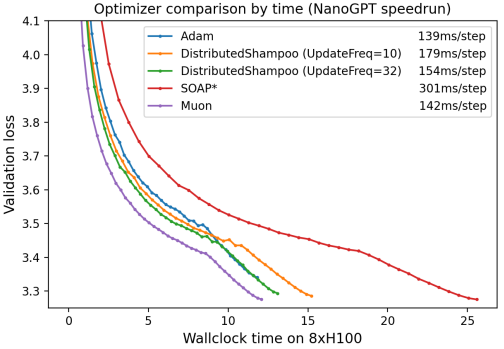
Optimizer	**What the update computes**
Muon	$UV^\top$ of the momentum, by Newton–Schulz
Shampoo	inverse fourth roots of L and R, by eigendecomposition
SOAP	Adam in the eigenbasis of L and R, basis refreshed every k steps
Kron (PSGD)	a Kronecker-factored preconditioner fit online, with no roots or eigendecompositions

Gupta, Koren, Singer, Shampoo (arXiv:1802.09568); Bernstein and Newhouse, Old Optimizer, New Norm (arXiv:2409.20325) Eqs. 12–16; Vyas et al., SOAP (arXiv:2409.11321); Li, PSGD (arXiv:1512.04202).

On one node, a Muon training step takes about as long as an AdamW step



*SOAP is under active development. Future versions will significantly improve the wallclock overhead.

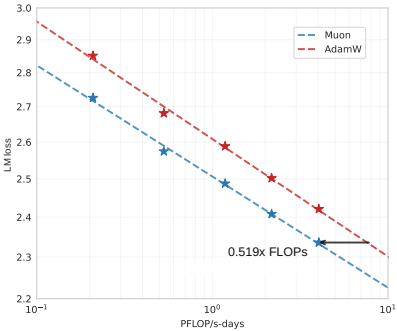


*SOAP is under active development. Future versions will significantly improve the wallclock overhead.

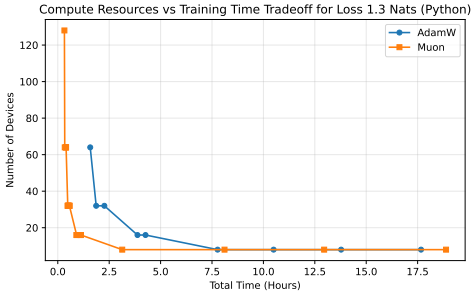
NanoGPT speedrun (124M, 8×H100). Muon reaches each loss in fewer tokens (left) at 142 ms per step vs 139 ms for Adam, so it also wins in wall-clock time (right). SOAP needs even fewer tokens but its 301 ms steps make it slowest.

Jordan et al., Muon (2024).

Muon reaches the same loss with fewer FLOPs and tolerates larger batches



Compute-optimal scaling fits. Muon needs 0.52× AdamW's FLOPs for the same loss.



Devices vs time to reach a fixed loss. Muon's curve is lower and further left.

A larger critical batch size matters for systems. With more devices, the global batch grows, and past the critical batch size each token does less work. Muon keeps its data efficiency at larger batches, so a run can use more data parallelism before it stops paying off. We return to the 2× figure when we look at how optimizer comparisons are run.

Moonlight (arXiv:2502.16982) Fig. 1; Essential AI, Practical Efficiency of Muon for Pretraining (arXiv:2505.02222) Fig. 1, §2.4.

2. Muon at Scale

The update is now a sequence of GEMMs

Computed once, Newton–Schulz adds under 1% of the FLOPs

For an $m \times n$ weight ($m \le n$) and a batch of B tokens:

$$\text{forward} + \text{backward} = 6\,mnB$$

$$\text{one NS step} \le 6\,m^2 n$$

$$\frac{\text{NS FLOPs}}{\text{training FLOPs}} \le \frac{T\,m}{B}$$

with T Newton–Schulz steps. The update is paid once per batch, so a larger batch hides it.

Model	m	B (tokens)	Tm/B
NanoGPT 124M	768	0.5M	0.7%
Llama 3 405B	16,384	16M	0.5%

Jordan et al., Muon (2024), "Runtime analysis".

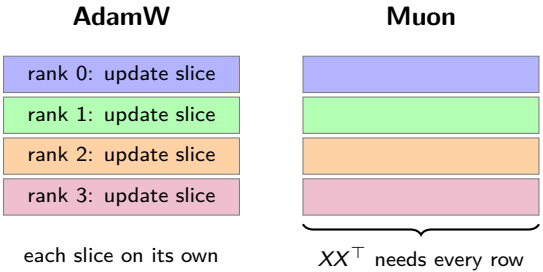
Question: If the FLOPs are this small, why did every lab that trained with Muon at scale write about its optimizer infrastructure?

The estimate makes three assumptions.

1. Each matrix is orthogonalized once, not once per rank.
2. The whole matrix is on one GPU.
3. The GEMMs run at full tensor-core speed.

At scale none of these holds by default. The next slides take them in turn.

Newton–Schulz needs the whole matrix, but each rank holds a slice



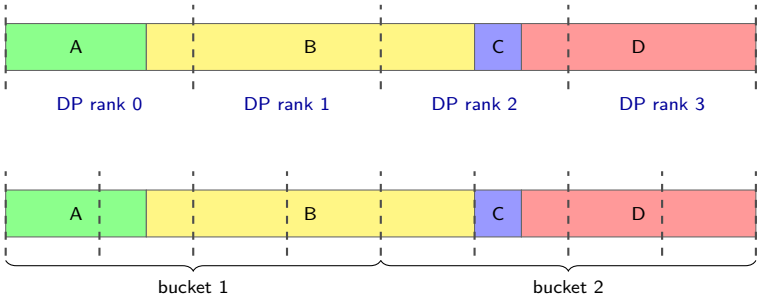
AdamW updates each entry using only that entry's gradient and state, so every rank can update its own slice.

Muon's update multiplies the matrix by its transpose. Every entry of $\mathrm{NS}(M)$ depends on every row of M, so a rank cannot compute its part of the update from its slice alone.

This breaks assumption 2.

Essential AI, Layer Sharding for Large-Scale Training with Muon (blog, May 2025).

Flat sharded buffers split matrices across ranks

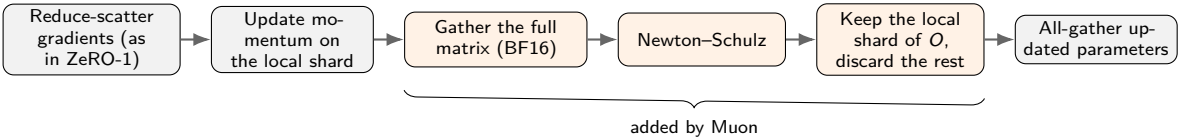


Megatron's distributed optimizer and ZeRO flatten all parameters into one buffer and cut it into equal shards (lecture 5). The cuts ignore matrix boundaries, so B and D each land on two ranks. AdamW does not care.

Buckets let communication overlap the backward pass, but they add more cuts, and every split matrix needs a gather before Newton–Schulz. FSDP2 goes further and splits every matrix across every rank.

B. Wang (NVIDIA), Emerging Optimizers, guest lecture, Scalable AI, Spring 2026.

The simplest fix repeats Newton–Schulz on every data-parallel rank



Moonlight's Distributed Muon does this. Each rank gathers the full matrix, runs Newton–Schulz, and keeps only its own shard. The extra gather is in BF16, so communication is at most 1.25× that of ZeRO-1 AdamW.

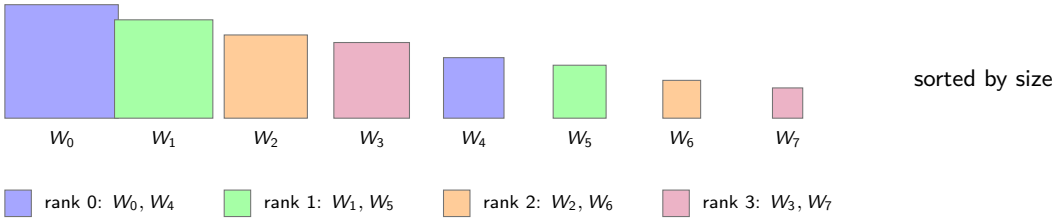
Every rank repeats the same Newton–Schulz, which breaks assumption 1.

	Optimizer cost
Moonlight ($m = 2{,}048$)	1–3% of step
Llama 3 405B ($m = 16{,}384$), 128 FSDP ranks	$0.5\% \times 128 \approx 64\%$

The repeated work is cheap for a small model and prohibitive for a wide model sharded over many ranks.

Moonlight (arXiv:2502.16982) §2.3, Alg. 1; Jordan et al., Muon (2024), "Runtime analysis" ($B = 16$M); Essential AI, Layer Sharding (blog, 2025).

Give each matrix one owner



Round-robin owners. Sort parameters by size. With N ranks, rank r orthogonalizes parameters $r, r+N, r+2N, \ldots$ and an all-gather shares the results. Each matrix is orthogonalized once and the work is split N ways, which restores assumption 1.

Balancing owners. Matrix sizes differ, so some owners get more work than others and the all-gather is uneven. DeepSeek-V4 packs dense matrices onto ranks with a knapsack solver (under 10% padding) and stacks same-shaped matrices into one batched GEMM.

github.com/KellerJordan/Muon; Megatron-LM layer_wise_optimizer.py; DeepSeek-V4 (arXiv:2606.19348) §3.4.1.

Moving whole matrices costs less than distributing every GEMM

Strategy	How it works	NS FLOPs / rank	Traffic	Share of step
Replicated	every rank repeats it	all	1	43%
Sharded matmul	distributed GEMMs	$1/S$	6.25	26%
Layer sharding	whole matrices to owners	$1/S$	$\approx$0.1 (GPU)	10%

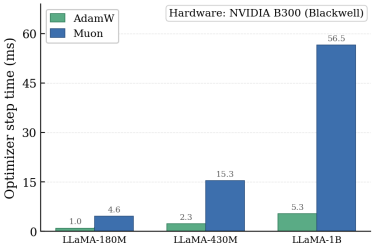
S ranks. Traffic in model-sized gathers. Share of step is Muon's share of step time for a 2B model on 8 MI300Xs.

Sharded matmul communicates inside each of the $3T$ GEMMs. Layer sharding communicates once before and once after. This is the same trade-off as in tensor parallelism (lecture 5).

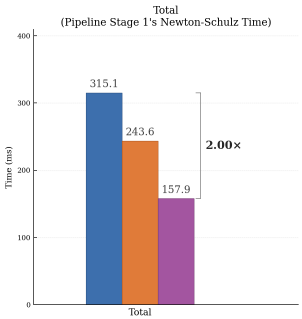
With this engineering in Megatron-Core, Kimi K2 on 256 GB300s runs at 1,029 model TFLOP/s per GPU with Muon and 1,051 with AdamW, about 2% slower.

Essential AI, Layer Sharding for Large-Scale Training with Muon (blog, May 2025); NVIDIA, Emerging-Optimizers in Megatron-Core (blog, Apr 2026).

Custom kernels make Newton–Schulz about twice as fast



Optimizer step time on one B300. Muon's step is 5–11× AdamW's.



Newton–Schulz time, Kimi K2 pipeline stage 1, H100. Blue is PyTorch, orange uses custom kernels, purple is Gram Newton–Schulz.

This is assumption 3. Plain PyTorch does not run these GEMMs at full speed.

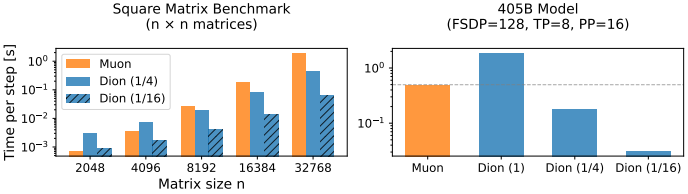
$XX^\top$ is symmetric, so a kernel can compute only half of its tiles.

Gram Newton–Schulz iterates on the small $m \times m$ matrix $XX^\top$ and multiplies by the rectangular X only at the ends.

With a large pre-training batch, Newton–Schulz was about 2% of Kimi K2's step. In fine-tuning with a small batch it was 17% for Llama 3 70B, where these kernels matter most.

Zhang, Amsel, Chen, Dao, Gram Newton–Schulz (Dao AI Lab blog, 2026); Flash-Muon (github.com/nil0x9/flash-muon); Dion3 (arXiv:2608.11612) §4–5.

Dion reduces communication with a low-rank orthogonal update



Optimizer step time. Left, single square matrices. Right, the full 405B configuration; Dion (1/16) keeps rank $r = d/16$.

Dion replaces Newton–Schulz with one step of power iteration per update, keeping a rank-r factorization warm between steps. Error feedback carries the discarded part to the next step.

	Muon	**Dion**
Traffic	$O(mn)$	$O((m+n)r)$
Sharded	gather first	works on shards

Owners and layer sharding adapt the parallel layout to Muon. Dion adapts the optimizer to the existing layout.

Ahn et al., Dion (arXiv:2504.05295) Fig. 1, §4–5.

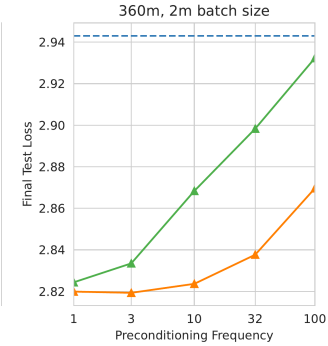
Shampoo and SOAP refresh an expensive preconditioner every few steps

Shampoo's inverse roots of L and R need an eigendecomposition, which is far slower than a GEMM. The usual fix is to recompute them only every few steps and reuse the stale roots in between.

Distributed Shampoo (Meta, ResNet-50 on 8 V100s)

Recompute roots every	**Step-time overhead**
1 step	53%
20 steps	12.8%
50 steps	5.2%
100 steps	2.0%

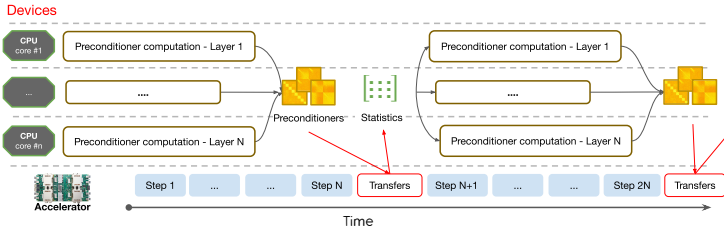
SOAP runs Adam in Shampoo's eigenbasis, so a stale basis costs less loss than a stale Shampoo root (right).



Final loss vs refresh interval. Orange is SOAP, green is Shampoo, dashed blue is AdamW.

Shi et al., Distributed Shampoo (arXiv:2309.06497) §4, App. D.2.1, Table 4; Vyas et al., SOAP (arXiv:2409.11321) Fig. 1.

Preconditioner work can run on the idle host CPUs



Anil et al. computed Shampoo's inverse roots on the host CPUs in FP64 while the accelerators kept training with the previous preconditioner. The new one is copied over when ready.

Transformer on WMT: 1.95× fewer steps, 16% slower per step, 40% less wall-clock time.

The same idea returns in the architecture section. Work whose inputs are known early can move to the host.

Anil et al., Scalable Second Order Optimization for Deep Learning (arXiv:2002.09018) §5.

What each optimizer costs

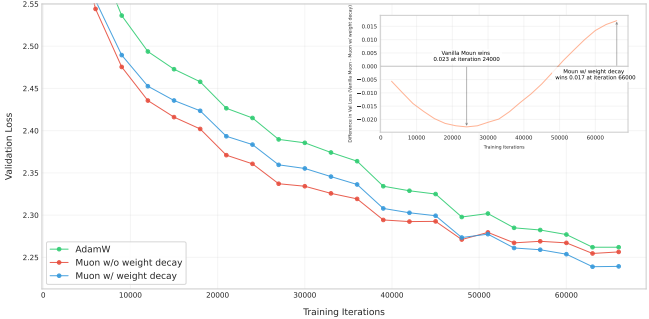
	State per param	Update compute	Fits ZeRO / FSDP slices	Usual fix
AdamW	m, v	elementwise, bandwidth-bound	yes	none needed
Muon	m	$3T$ GEMMs per matrix, every step	no, needs whole matrices	owners, batched NS, symmetric kernels
Dion	m, low-rank factor	power iteration, rank r	yes	none needed
Shampoo	m, L, R	eigendecomposition	no	refresh every k steps, block, distribute, or move to CPU
SOAP	m, v, L, R	Adam in a rotated basis, basis refreshed every k steps	no	same as Shampoo

All matrix optimizers pay per matrix, per step. They are cheap in total FLOPs and expensive in how the work has to be placed.

3. Controlling Scale

Normalized updates do not shrink on their own, so weights, logits, and activations need explicit bounds

Without weight decay, the weights keep growing



Validation loss. Muon without weight decay (red) leads early and falls behind by the end. Muon with weight decay (blue) stays ahead of AdamW with weight decay (green).

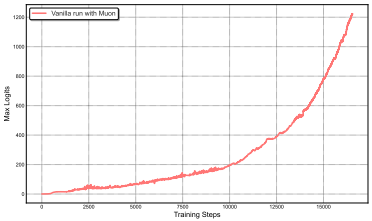
Adam and Muon both normalize their updates, so the step size does not shrink as the gradient shrinks. Without weight decay, the weights and layer outputs grow for the whole run, past the range where BF16 is precise.

Muon makes this more pressing. Its update sets every singular value to 1, so each step pushes on all directions of a matrix at once. Moonlight adds AdamW-style decoupled weight decay and matches the update RMS to AdamW's.

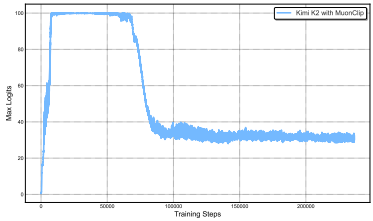
The rest of this section covers three things that need a bound: attention logits, activations, and the weights themselves.

Moonlight (arXiv:2502.16982) §2.2, Fig. 2.

Attention logits explode more often under Muon



Largest attention logit, vanilla Muon, 53B MoE.



Kimi K2 with QK-Clip ($\tau = 100$).

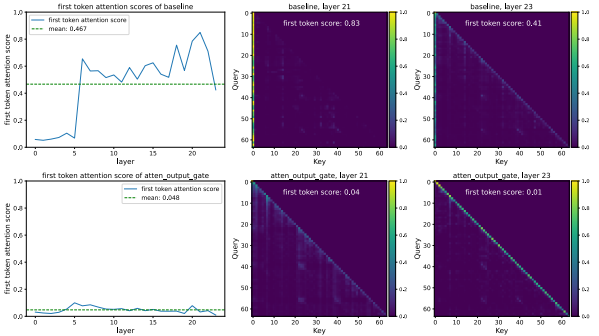
Kimi K2 bounds the logits with QK-Clip, which rescales weights after each optimizer step. For every head h whose largest pre-softmax logit $S^h_{\max}$ in the batch exceeds τ, it sets $\gamma = \tau / S^h_{\max}$ and multiplies that head's W_q and W_k by $\sqrt{\gamma}$.

QK-Norm (lecture 8) would also bound the logits, but MLA (lecture 4) never materializes the full keys, so K2 could not use it. 12.7% of heads were clipped early in training. After that the clip never fired.

Monitoring is free, since the fused attention kernel already tracks each row's maximum for its online softmax (lecture 6).

Kimi K2 (arXiv:2507.20534) §2.1, Alg. 1, Fig. 2, App. D.2; Megatron-LM multi_latent_attention.py (clip_qk).

A gate after attention removes attention sinks and outlier activations



Attention to the first token by layer. Baseline (top) averages 46.7%; with the output gate (bottom) it is 4.8%.

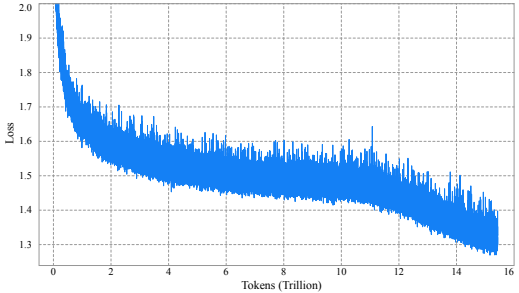
Large values also show up in activations, with any optimizer. Multiply each head's attention output elementwise by $\sigma(xW_g)$. A head with nothing to contribute can output zero instead of dumping its attention on the first token.

	Baseline	**Gated**
Max activation	1053	94
48 layers, LR 8×10^{-3}	diverges	trains

Smaller outliers make BF16, FP8, and FP4 formats (lecture 8) easier to scale. Qwen3-Next uses the gate; gpt-oss and DeepSeek-V4 clamp SwiGLU outputs for the same reason.

Qiu et al., Gated Attention for Large Language Models (arXiv:2505.06708) Fig. 2, Tables 2 and 4; gpt-oss (arXiv:2508.10925) §2; DeepSeek-V4 (arXiv:2606.19348) §4.2.3.

Stability fixes in recent Muon runs



Kimi K2 per-step training loss, unsmoothed, over 15.5T tokens. There are no spikes.

Run	What kept it stable
Moonlight	weight decay, update RMS matched to AdamW
Kimi K2	QK-Clip on MLA heads
GLM-4.5	QK-Norm; Muon on all matrices except embeddings
GLM-5	Muon applied to each head's Q, K, V separately
DeepSeek-V4	RMSNorm on queries and KV entries; 8+2 NS schedule; SwiGLU outputs clamped to $[-10, 10]$

Switching a run from AdamW to Muon midway can fail. Marin tried it on its 32B run and the loss rose. Moving to a backbone with QK-Norm fixed the instability.

Kimi K2 (arXiv:2507.20534) Fig. 3; GLM-4.5 (arXiv:2508.06471); GLM-5 technical report (Feb 2026), as summarized in B. Wang's guest lecture; DeepSeek-V4 (arXiv:2606.19348) §2.4, §4.2.3; Marin 32B retrospective (marin.readthedocs.io).

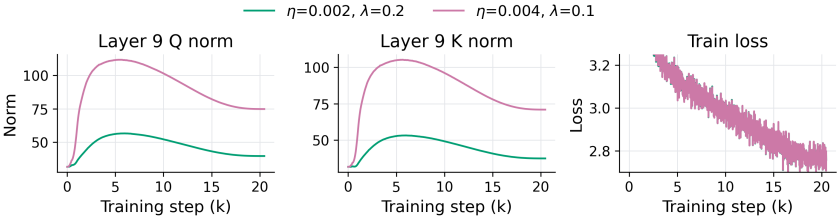
Two of the fixes change the effective learning rate

Fix	**Knob it adds**	**Acts through η?**
Update RMS matching	update scale	**yes**, it multiplies η
Decoupled weight decay	λ	**yes**, through $\eta\lambda$
QK-Clip	logit cap τ	no, a bound
SwiGLU clamp	clamp range	no, a bound

Logits and activations get hard bounds, which are set once and rarely touched. The weights get no bound of their own. Their norm is controlled indirectly through η and λ, so a new learning rate or a new model size means retuning both. The next slides look at what weight decay actually controls, and then replace it with a direct bound on the weight norm.

Moonlight (arXiv:2502.16982) §2.2; Kimi K2 (arXiv:2507.20534) §2.1; DeepSeek-V4 (arXiv:2606.19348) §4.2.3.

Weight decay sets the angular step size



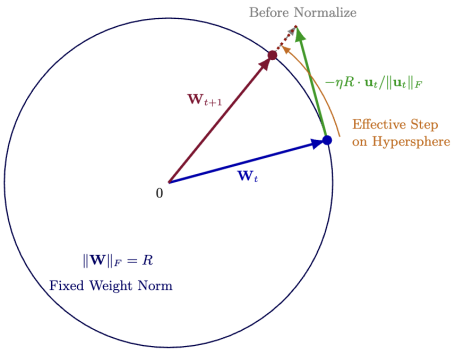
Two AdamW runs with the same $\eta\lambda$. Losses match (right) while the query and key weight norms differ by $2\times$ (left).

With decoupled weight decay, a weight's norm settles where growth from updates balances shrinkage from decay. At that point, how far the weight *rotates* per step depends mainly on the product $\eta\lambda$.

Behind normalization layers, the weight's direction matters and its scale does not. So weight decay is an indirect way of setting the angular learning rate, and it takes a long time to reach its equilibrium.

Wen, Dang, Lyu, Ma, Liang, Fantastic Pretraining Optimizers II: Hyperball Optimization (arXiv:2606.16899) §5, Fig. 10.

MuonH and AdamH keep every weight matrix at a fixed norm



$$W_{t+1} = R \cdot \frac{W_t - \eta_t R\, u_t/\|u_t\|_F}{\left\| W_t - \eta_t R\, u_t/\|u_t\|_F \right\|_F}, \qquad R = \|W_0\|_F$$

AdamH	$u_t = \hat{m}_t/(\sqrt{\hat{v}_t} + \epsilon)$
MuonH	$u_t = \mathrm{msign}(M_t) \approx \mathrm{NS}(M_t)$

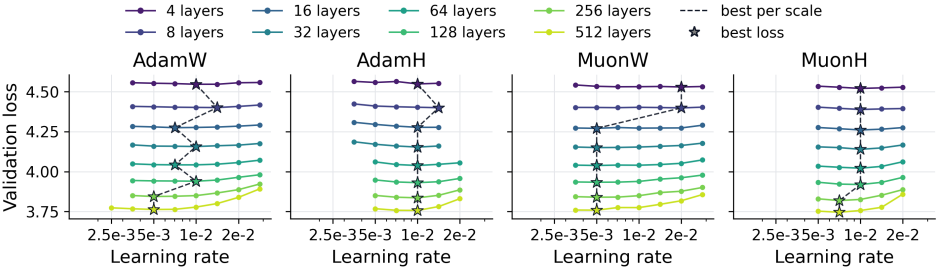
Take a step of length $\eta_t R$, then project back onto the sphere of radius R. There is no weight decay, and η_t is directly the relative step $\|\Delta W\|_F/\|W\|_F$ (good values 0.0025–0.01).

Applied to attention and MLP matrices. Embeddings, norm gains, and scalars use Adam.

Systems cost is two Frobenius norms per matrix per step. On a sharded matrix each norm is one scalar all-reduce.

Fantastic Pretraining Optimizers II (arXiv:2606.16899) Eq. 1, Alg. 1, Fig. 1; Fantastic Pretraining Optimizers 2.1 (blog, Dec 2025).

With a fixed norm, the best learning rate barely moves with depth



Loss vs learning rate for depths 4 to 512 (width 128, 10B tokens each). Stars mark the best rate at each depth.

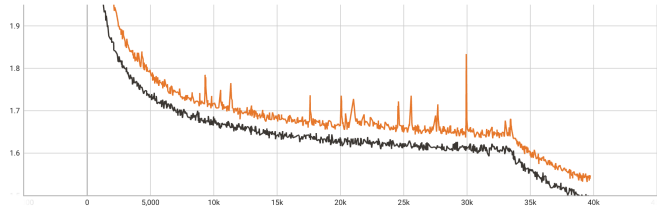
The best learning rate drifts by $\approx 1.4\times$ across depth and across width for AdamH and MuonH, and by 3–4$\times$ for AdamW and MuonW. A rate tuned on a small model is closer to right on the large one, which is the goal of μP, reached here by fixing the norm.

Fantastic Pretraining Optimizers II (arXiv:2606.16899) Fig. 5.

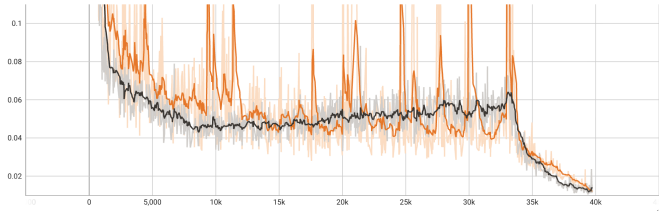
4. Evaluating Optimizers

How to read a speedup claim, and a change that holds up

In a 30B MoE run, Muon had lower loss and no spikes



Training loss. Orange is AdamW, black is Muon.



Gradient norm, same runs.

B. Wang (NVIDIA), Emerging Optimizers, guest lecture, Scalable AI, Spring 2026. Internal TensorBoard logs.

NVIDIA trained a 3B-active, 30B-total MoE on 1T tokens with a 24M-token batch. AdamW's loss and gradient norm spike repeatedly. Muon's do not, and its final validation loss is lower (1.195 vs 1.206).

The runs differ in more than the optimizer. The Muon run also used QK layer norm and split Q, K, V for Newton–Schulz.

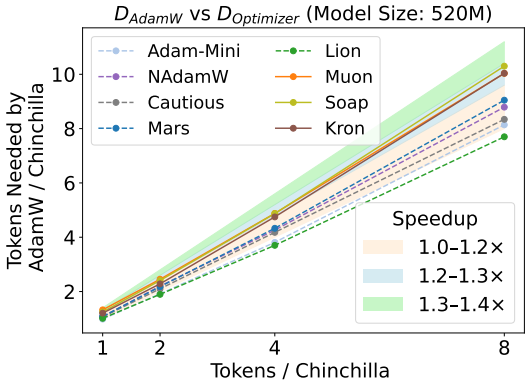
The logged wall-clock was 5.1 days for Muon vs 1.7 for AdamW at the same step, on an experimental branch.

Question: How much of this result can be credited to Muon? The rest of this section is about what to check before answering.

A controlled comparison of eleven optimizers

Models	130M, 300M, 520M, 1.2B (Llama-style)
Data	1, 2, 4, 8× Chinchilla (20 tokens per parameter)
Scalar	AdamW, NAdamW, Mars, Cautious, Lion, Adam-mini, Sophia
Matrix	Muon, Scion, Kron, SOAP
Tuning	coordinate descent over every hyperparameter, per optimizer, per scale

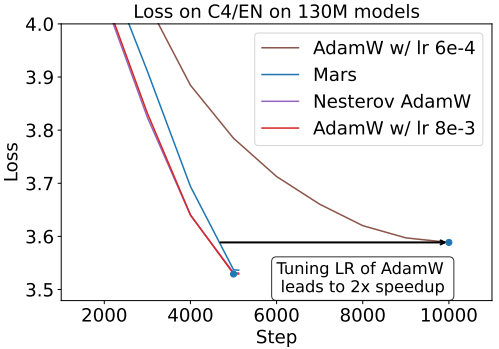
Speedup is measured in data. Fit AdamW's loss as a function of tokens, then ask how many tokens AdamW would need to reach the other optimizer's final loss.



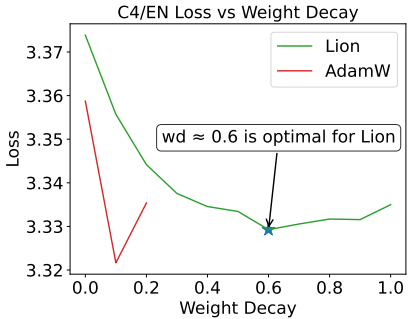
520M. AdamW-equivalent tokens vs actual tokens. The shaded bands mark 1.0–1.4×.

Wen, Hall, Ma, Liang, Fantastic Pretraining Optimizers and Where to Find Them (arXiv:2509.02046) Tables 2–3, Fig. 3.

Tuning one learning rate gives AdamW a 2× speedup



130M. Raising AdamW's peak rate from GPT-3's 6×10^{-4} to 8×10^{-3} halves the steps to a given loss.

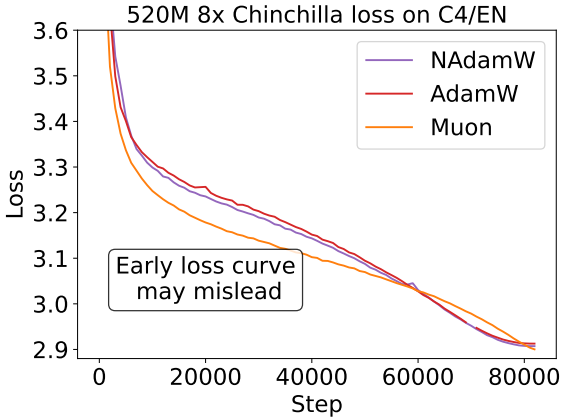


The best weight decay is 0.1 for AdamW and 0.6 for Lion.

A new optimizer compared against an untuned AdamW can show a 2× speedup that comes entirely from the baseline. Copying the baseline's hyperparameters to the new optimizer is also unfair, since each optimizer has its own best values.

Fantastic Pretraining Optimizers (arXiv:2509.02046) Fig. 1.

Loss curves cross during learning-rate decay



520M, 8× Chinchilla.

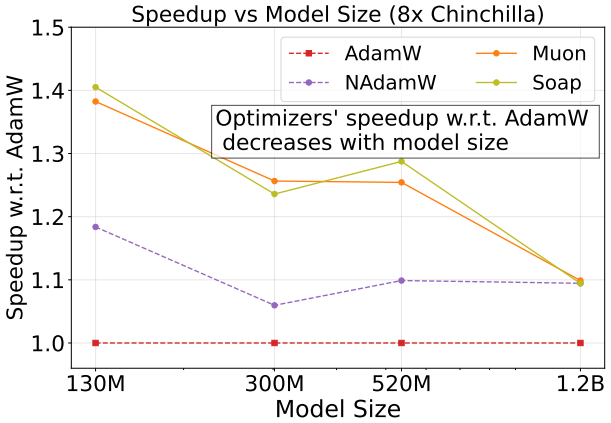
Fantastic Pretraining Optimizers (arXiv:2509.02046) Fig. 5.

Muon leads for most of the run. During the final decay, AdamW and NAdamW catch up and end at the same loss.

The same happens inside one optimizer. For Muon, no weight decay looks best early, and weight decay 0.1 is best at the end.

Compare optimizers only at the end of a full schedule, each with its own decay.

Speedups shrink as models grow



Speedup over tuned AdamW at 8× Chinchilla.

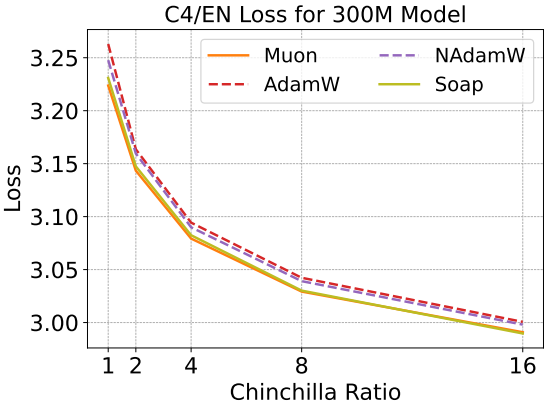
Fantastic Pretraining Optimizers (arXiv:2509.02046) Fig. 1, App. A.1.

Against a tuned AdamW, no optimizer beats $\approx 1.4\times$. Muon and SOAP fall from $\approx 1.4\times$ at 130M to $\approx 1.1\times$ at 1.2B.

Below 520M, matrix optimizers (Muon, SOAP, Kron, Scion) give $\approx 1.3\times$. Tuned scalar optimizers give under $1.2\times$, and Lion and Adam-mini stay within 5% of AdamW.

Question: Their fitted scaling law predicts Muon slightly behind AdamW at 7B, yet trillion-parameter production runs chose Muon. What could explain the difference?

The best optimizer depends on tokens per parameter



300M, final loss vs data budget.

Fantastic Pretraining Optimizers (arXiv:2509.02046) Fig. 4.

Muon is best from 1 to 4× Chinchilla. At 8× and beyond, SOAP and Kron pass it.

Muon throws away the singular values of the momentum. SOAP and Kron keep a running estimate of the gradient's second moments, which takes more steps to become accurate and helps more in long runs.

Lecture 9 showed that modern small models train at 100× Chinchilla or more, a regime where no one has run this comparison.

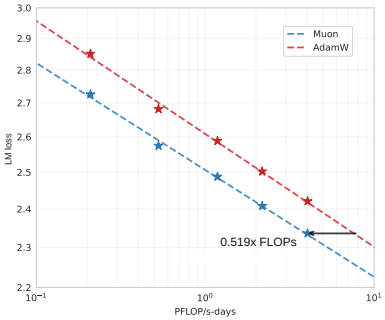
Questions to ask of an optimizer result

Question	Why it matters
Was the baseline tuned as hard as the new method?	One learning rate is worth 2× on AdamW
Does each optimizer have its own hyperparameters?	Best weight decay differs by 6× between AdamW and Lion
Is it compared at the end of a decayed schedule?	Curves cross during decay
At what model size, and does the gap shrink with size?	1.4× at 130M became 1.1× at 1.2B
At what tokens per parameter?	The winner changes between 4× and 8× Chinchilla
Is the speedup in tokens or in wall-clock time?	A 2× slower step cancels a 2× token saving
Was the architecture tuned for AdamW?	Norm placement, warmup, and other defaults were tuned for AdamW over a decade, so a new optimizer runs on its competitor's recipe

A second benchmark (Semenov et al.) reached the same conclusions about tuning and horizon, with a different ranking of the winners.

Fantastic Pretraining Optimizers (arXiv:2509.02046); Semenov, Pagliardini, Jaggi, Benchmarking Optimizers for LLM Pretraining (arXiv:2509.01440).

Checking Moonlight's 2× claim



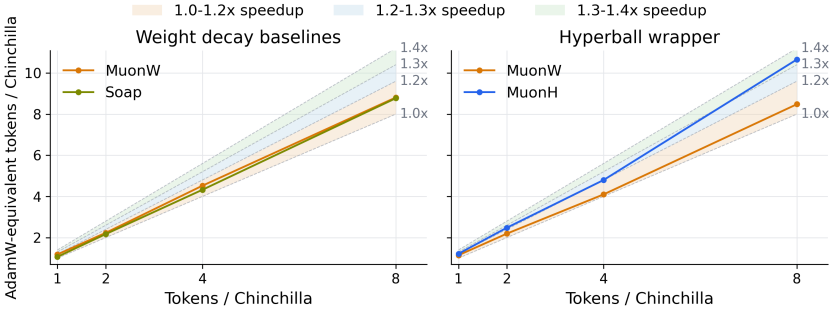
Muon needs 0.52× AdamW's training FLOPs for the same loss.

Question	Moonlight's scaling-law runs
Baseline tuned?	yes, AdamW grid-searched at each compute-optimal size
Own hyperparameters?	no, Muon reused AdamW's through RMS matching
Model size	399M to 1.5B
Tokens per parameter	≈22–26, near Chinchilla, where Muon does best
End of a decayed schedule?	not reported; runs were "trained to convergence"
Tokens or wall-clock?	training FLOPs

The baseline was tuned and Muon was not, which favors AdamW. The runs also sit where Muon does best, at small sizes near Chinchilla. With every AdamW hyperparameter tuned by coordinate descent, the controlled benchmark measured about 1.3–1.4× at similar sizes. Neither study goes past 1.5B.

Moonlight (arXiv:2502.16982) §2.2, §3.2, Table 2, Fig. 3, App. B; Fantastic Pretraining Optimizers (arXiv:2509.02046) Fig. 1.

A fixed norm raises Muon's speedup at 1.2B from 10% to 20–30%



1.2B, AdamW-equivalent tokens vs actual tokens, with the same speedup bands as before. Left, weight-decay baselines; right, MuonW vs MuonH.

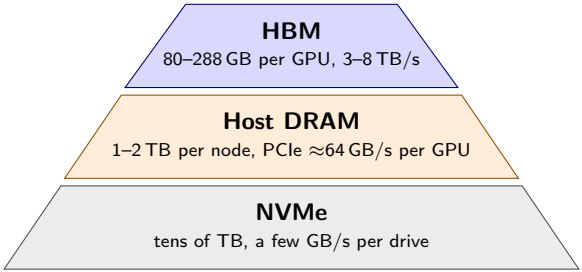
This result comes from the same controlled benchmark, with every optimizer tuned. MuonW (Muon with weight decay) gives ≈10% over AdamW at 1.2B. MuonH, the fixed-norm variant from earlier, gives 20–30%, and the gap grows with the data budget. MuonH starts at a higher loss and overtakes during the decay, which is one more reason to compare only final checkpoints.

Fantastic Pretraining Optimizers II (arXiv:2606.16899) Fig. 2.

5. Architecture and Memory Traffic

Where parameters and activations sit in the memory hierarchy

Parameters can live in three tiers of memory



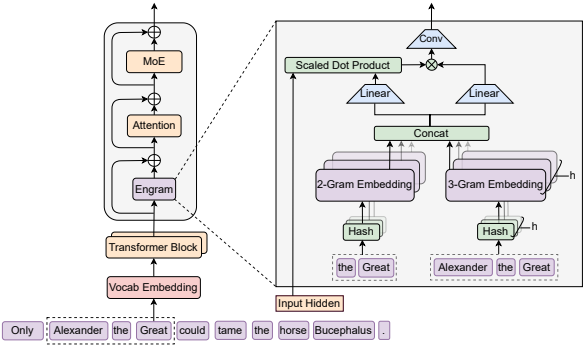
Host memory is about 10× larger than HBM and about 50× slower to reach.

A parameter can live in host memory if the GPU knows which rows it needs early enough to copy them before the layer runs.

Dense and MoE weights fail this test. Every dense weight is used every step, and which experts a token uses depends on the hidden state at that layer.

Typical H100/B200 node figures; PCIe Gen5 ×16.

Engram adds hashed n-gram lookup tables to a MoE



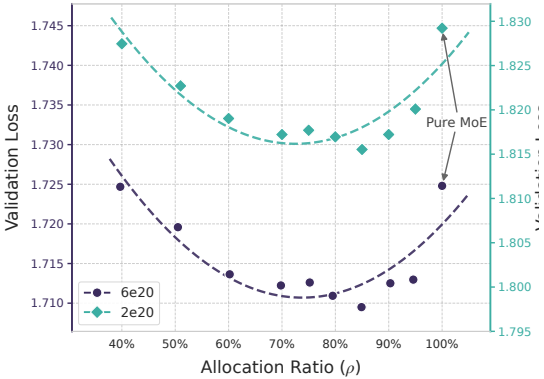
For each position, take the last 2 and 3 tokens. Several hash functions map each n-gram to a row in its own table, and the rows are concatenated.

The hidden state decides how much of the retrieved vector to use through a sigmoid gate, then it is added to the residual stream. Engram sits at a few early layers, before attention.

Frequent local patterns (names, phrases, code idioms) become a table lookup, so attention and experts do not have to rebuild them.

DeepSeek, Conditional Memory via Scalable Lookup (Engram, arXiv:2601.07372) Fig. 1, §2.

The best split of sparse parameters gives about a quarter to memory



Loss vs ρ, the share of inactive parameters given to experts, at two compute budgets. $\rho = 100\%$ is a pure MoE.

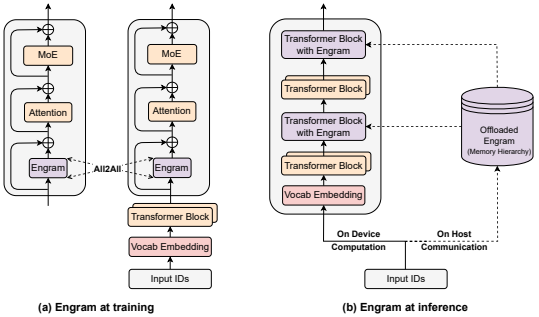
At fixed total parameters and FLOPs, loss is lowest at $\rho \approx 75$–80%.

26.7B total, 3.8B active, 262B tokens

	MoE	**Engram**
Routed experts	72	55
Engram parameters	–	5.7B
MMLU	57.4	60.4
BBH	50.9	55.9
HumanEval	37.8	40.8
Multi-query NIAH	84.2	97.0

Engram (arXiv:2601.07372) Fig. 3, Tables 1–2.

Addresses from token ids let the table live in host memory



(a) Training shards tables across GPUs. (b) Inference fetches rows from host memory using only the input ids.

The row indices depend only on the input tokens, so they are known before the forward pass starts. The copy over PCIe overlaps the layers that run before the Engram layer.

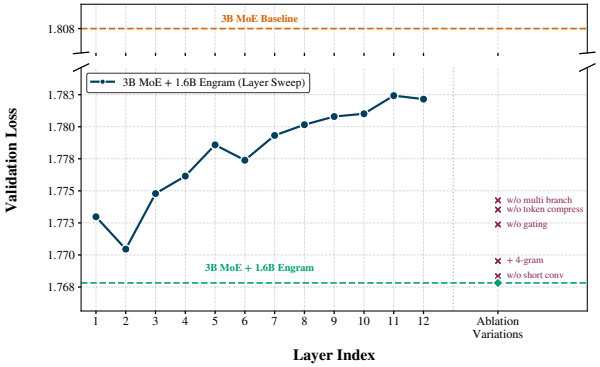
100B-parameter table entirely in host DRAM (H800)

Backbone	tok/s	Change
4B dense	9,032 → 8,858	−1.9%
8B dense	6,316 → 6,140	−2.8%

In training, the tables are sharded across GPUs and rows are exchanged with an all-to-all, as in expert parallelism.

Engram (arXiv:2601.07372) Fig. 2, §2.5, §6.4, Table 4.

Placement trades model quality for time to hide the copy



Loss vs the layer where a single Engram module is inserted (12-layer 3B MoE).

A deeper layer gives the prefetch more time to finish, but the model does best with the lookup early (layer 2).

Question: How much PCIe bandwidth does the lookup need?

Illustrative sizes. 3 n-gram orders $\times$ 8 hash heads $\times$ 2 layers is 48 rows per token. At 512 FP8 values per row that is 24 KB per token. At 20,000 tokens/s per GPU the lookup reads $\approx$0.5 GB/s, under 1% of a PCIe link.

Word frequencies are Zipfian, so a small cache of hot rows in HBM would serve most lookups.

Engram (arXiv:2601.07372) Fig. 5, §6.

Production models now keep lookup tables off the accelerator

Model	Table size	Where the table lives and what was found
DeepSeek-V4.1-Flash (Sep 2026)	196B, FP8, 2 layers	host memory, prefetched by background RDMA; first production Engram
Qwen3.8-Flash-Next (Aug 2026)	51B, 1 layer	off the accelerator, at layer 2 so the copy overlaps layer 1. Lower loss, but no clear downstream gain over the MoE baseline
Google Gemma 3n	per-layer embeddings	CPU or fast storage; 5B total, $\approx$2B in accelerator memory

DeepSeek-V4.1-Flash (arXiv:2609.19969) §2.4.2, §3.1.3; Qwen3.8 (arXiv:2608.30320) §2.3.1, Tables 7–8; Gemma 3n documentation (ai.google.dev).

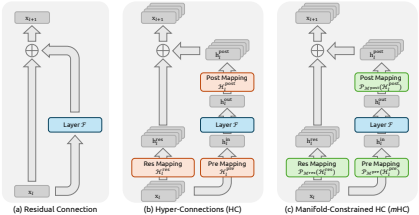
Prefetching needs the address before the layer runs

Sparse component	Address comes from	Where it can live
N-gram tables (Engram, SCONE, Over-Encoding)	input token ids	host DRAM or NVMe, prefetched
Hash routing (first MoE layers of DeepSeek-V4)	input token ids	expert choice known in advance
Learned MoE routing	hidden state at that layer	HBM; experts moved by all-to-all
Product-key memory layers (Meta, UltraMem)	hidden-state query	HBM, with fast gather kernels
KV cache	position	HBM, or offloaded by the serving engine

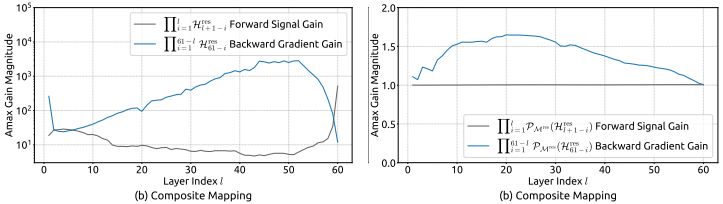
Whether a table can leave the GPU depends on when its address is known. Meta's memory layers scale to 128B memory parameters, but their lookups depend on the hidden state, so they need custom gather kernels (3 TB/s on H100) and stay on the GPU.

Engram (arXiv:2601.07372); DeepSeek-V4 (arXiv:2606.19348) §2.1; Berges et al., Memory Layers at Scale (arXiv:2412.09764); UltraMem (arXiv:2411.12364).

Hyper-connections widen the residual stream



(a) Residual. (b) Hyper-connections keep n streams mixed by learned matrices. (c) mHC constrains the mixing.



Gain of the composed residual mixing by layer (27B). Unconstrained (left, log scale) peaks near 3000; mHC (right) stays below 1.6.

Multiplying $n \times n$ mixing matrices across 60 layers can amplify the signal by thousands. mHC projects each residual mixing matrix onto the doubly stochastic matrices (rows and columns sum to 1) with 20 Sinkhorn–Knopp iterations, so the product cannot grow. DeepSeek-V4 uses mHC with $n = 4$.

DeepSeek, mHC: Manifold-Constrained Hyper-Connections (arXiv:2512.24880) Figs. 1, 3, 7; DeepSeek-V4 (arXiv:2606.19348) §2.2.

A wider residual stream costs memory traffic

Per token, per layer, hidden size C

	Residual	**mHC,** n=4
Elements read	$2C$	$\approx 21C$
Elements written	C	$\approx 13C$
Pipeline activations	C	$4C$

The mixing is a few small matrix products, so it is bandwidth-bound, like the residual add and RMSNorm from lecture 3.

DeepSeek's fixes

- Fuse the mixing, norm, and Sinkhorn steps into a few kernels.
- Do not store the wide stream. Recompute it for blocks of layers aligned with pipeline stages.
- Overlap the $4\times$ larger pipeline sends with compute in DualPipe.

The result is 6.7% extra time at $n = 4$.

V4.1 moves part of the mixing one block later and halves the activation traffic again. Qwen3.8 stores its widened residual in FP8.

mHC (arXiv:2512.24880) Table 2, §4.3; DeepSeek-V4.1-Flash (arXiv:2609.19969) §2.4.1; Qwen3.8 (arXiv:2608.30320) §2.2.

Conclusion

- Matrix optimizers add few FLOPs. Their cost is that each update needs a whole matrix, which changes how the optimizer is sharded and scheduled.
- The systems fixes reuse a few ideas. Give each matrix one owner, batch same-shaped work, and do expensive work less often or off the critical path.
- Large runs stay stable by controlling weight norms, attention logits, and activations explicitly.
- Optimizer speedups depend on baseline tuning, model size, and training length. Against a tuned AdamW, the gains are real but modest and shrink with scale.
- Architecture choices increasingly follow the memory hierarchy. Tables indexed by token id can live off the GPU, and a wider residual stream costs memory traffic.