EVOLVING MACHINES

The Echelon Optimizer

A rank-r spectral clip with a $\sqrt{2}$ stability bound,
machine-checked convergence, and a fused GPU kernel

Tushar Jain
Evolving Machines

November 16, 2026 · Version 2

Abstract

Muon orthogonalizes the momentum of each hidden weight matrix with five Newton–Schulz iterations, at a cost that grows with the cube of the layer width. We introduce Echelon, which tracks an orthonormal frame for the top-r singular directions of the momentum, applies Muon's treatment to that small head, and scales the remaining tail by an estimate of its largest singular value. Echelon costs $O(mnr)$ for an $m \times n$ matrix. With an exact frame it is a spectral clip, $\sigma_i \mapsto \min(1, \sigma_i/\sigma_{r+1})$, whose threshold is set by the rank rather than by a tuned constant. For any orthonormal frame, however stale, its update has spectral norm at most $\sqrt{2}$ provided the head factor has norm at most one and the tail scale is at least the tail's spectral norm, and one cubic Newton–Schulz step brings it to at most 1. We prove these bounds, a convergence rate of the same order as Muon's, and a set of cost and separation results in Lean 4 with Mathlib (35 audited theorems, three standard axioms, no `sorry`). Our training code does not meet the bounds' conditions: measured in training, the tail scale is below the tail's norm in about 92% of updates, and the update exceeds $\sqrt{2}$ in about 9%, mostly early. On one NVIDIA RTX PRO 6000 Blackwell GPU, the batched implementation that our training runs use cuts the optimizer step over every hidden matrix of a model with 70B-class layer shapes from 33.9 s to 3.23 s (10.5×, rank 64). In a 1.3B-parameter training run (three seeds per optimizer, 2,000 steps) the optimizer step is 2.5× faster, the full training step 14% shorter, and Echelon's validation loss is within seed noise of Muon's (4.3751 ± 0.0115 against 4.3757 ± 0.0109). A fused Triton kernel moves 20 bytes per matrix element instead of about 78 and, at rank 32, is 10.4–17.6× faster than Muon on the real layer shapes of five current open models on the RTX PRO 6000 (6.7–13.5× at rank 64). The speedup depends on the hardware: on a B300, where Muon's matrix products run 3.4× faster, the whole-model step of DeepSeek V4.1 Flash is 3.7× faster at rank 32. Combining optimizer timings with a timed forward and backward pass, both measured on a B300, gives an estimated 35% shorter training step for DeepSeek V4.1 Flash on 16 B300 GPUs, the fewest that hold its training state (31% on 24), at 256k tokens per step and rank 32. That is a per-GPU accounting with the optimizer work split evenly and no communication, not a training run. At small widths Echelon trails Muon, and we trace the gap to how flat the update's spectrum is, a quantity that worst-case rate theory does not see. We state every limitation of these measurements in Section 9. The Lean proofs are available at https://github.com/evolving-machines-lab/echelon-lean (Apache-2.0); the code, including the fused kernel, and a package will be released on GitHub soon.

Contents

1 Introduction **5**

2 Background: what Muon computes and what it costs **8**
- 2.1 The polar factor . . . 8
- 2.2 Newton–Schulz and its cost . . . 8

3 The Echelon optimizer **9**
- 3.1 Splitting the spectrum . . . 10
- 3.2 Echelon is a spectral clip . . . 11
- 3.3 Stability under a stale frame: a bound of $\sqrt{2}$ . . . 11
- 3.4 Polishing . . . 12
- 3.5 From the theorem to the code . . . 13
- 3.6 The alignment given up . . . 14
- 3.7 Implementation: batching, numerics and the fused kernel . . . 15
- 3.8 Reference algorithm and settings . . . 15

4 Theory: proofs a machine checked **16**
4.1 Setting and definitions . . . 17
4.2 Main theorems . . . 17
4.3 Pinned statements . . . 18
4.4 Three refuted claims about Newton–Schulz . . . 19
4.5 Pilot: informal optimizer claims tested in Lean . . . 19
4.6 What the proofs do not cover . . . 19

5 Where Echelon fits: the cost model **20**
5.1 A faster optimizer saves only the optimizer's share . . . 20
5.2 Why wider layers . . . 20
5.3 Why mixture-of-experts . . . 20
5.4 Which runs benefit . . . 21

6 Experiments **22**
6.1 Setup and conventions . . . 22
6.2 Model-shape timings: five current models . . . 22
6.3 What the timed-parts step assumes . . . 23
6.4 The rank of the large-model timings . . . 25
6.5 Batch-size dependence . . . 25
6.6 Speed in detail: per-layer optimizer time and the flop count . . . 26
6.7 A fused kernel . . . 28
6.8 Training at small width: GPT-2 small and a character model . . . 29
6.9 Training at 1.3B . . . 30
6.10 Wall-clock time and cost . . . 31
6.11 Savings projections and the cost calculator . . . 32

7 What the theory misses **34**

8 What we tried that didn't work **35**

9 Limitations and measurement notes **35**
9.1 The model-shape savings: four caveats . . . 35
9.2 Limits, in one place . . . 36
9.3 Scope of evidence: seeds and run length . . . 37
9.4 Timings versus training runs, and communication . . . 37
9.5 Batch-size dependence . . . 38
9.6 Kernel differences . . . 38
9.7 Ranks . . . 38
9.8 The small-width gap . . . 38
9.9 The width trend is confounded . . . 39
9.10 Baselines . . . 39
9.11 Scope of the proofs . . . 39
9.12 Other measurement notes . . . 40

10 Related work **40**

11 Directions for future work **41**

12 Conclusion and release **42**

References **43**

A Sources and measurement details **45**

B Reproducibility **51**

C Changes from version 1 **53**

D Lean sources **53**

1 Introduction

Muon [13] is one of the strongest optimizers for the hidden layers of transformers. Its update is easy to state: take the momentum matrix, keep its singular vectors, and set every singular value to one. The hard part is doing it fast. Muon approximates it with five Newton–Schulz iterations. For a square layer of width n they cost $30n^3$ floating-point operations per step, while everything else in the optimizer costs $O(n^2)$.

At frontier widths that cubic term is no longer small. On one Blackwell GPU, the Newton–Schulz step over every hidden matrix of a model with 70B-class layer shapes takes 33.9 seconds. Real training shards that work across GPUs, but it still grows with the cube of the width.

This report introduces the Echelon optimizer, a cheaper way to get most of what Muon does. For an $m \times n$ matrix and a rank r chosen by the user, it costs $O(mnr)$. It has a clear meaning in terms of singular values, and it is provably stable with a wrong frame, as long as its head factor and tail scale meet two conditions that our training code meets only approximately (Section 3.5). On the same GPU its batched implementation, the one our training runs use, cuts the 33.9-second step to 3.23 seconds. In our 1.3B run (three seeds per optimizer) its validation loss was within seed noise of Muon's, while each training step took 14% less time. At smaller scale it trails Muon slightly, and working out why was the most interesting thing we learned.

A companion blog post presents the results for a general audience [12]; this report gives the method, the proofs and every measurement in full.

We also wrote the convergence argument in Lean [5], a proof assistant, and had the computer check every step. That is still rare for optimizer work. A later section covers what that buys and what it doesn't (Section 4, in particular Section 4.6).

Contributions.

- **Method** (Section 3). A head–tail update $O = QP + T/\tau$ that equals the spectral clip $\sum_i \min(1, \sigma_i/\sigma_{r+1})\, u_i v_i^\top$ with an exact frame, costs $6mnr + 6(m+n)r^2 + 20r^3 + 10mn$ flops against Muon's $5(4m^2n + 2m^3)$, and has $\|O\|_2 \le \sqrt{2}$ for any orthonormal frame whenever the head factor has norm at most one and the tail scale bounds the tail's spectral norm.
- **Machine-checked theory** (Section 4). A Lean 4/Mathlib library of 35 audited theorems: a convergence rate for any update with certified alignment and bounded spectral norm, its specialization to Echelon, the $\sqrt{2}$ and polishing bounds, Muon's 1.25 overshoot bound, flop-count separations, a spiked-spectrum separation, and the refutation of three informal claims about Newton–Schulz from a popular write-up.
- **Implementation** (Sections 3.7 and 6.7). A batched PyTorch implementation that removes the per-matrix launch-latency floor, and a fused Triton [23] kernel that does the same math in 6 passes and 20 bytes per element.
- **Measurements** (Section 6). Optimizer timings at four widths; trained runs at GPT-2-small scale and at 1.3B parameters; timed parts of five current open models combined into an estimated training step; the cost model saving $= f(1 - 1/S)$ that ties them together.
- **A finding the theory misses** (Section 7). The update's spectral flatness, not the score the convergence theorem rewards, tracks training loss.

Table 1. Performance at a glance. The first two rows are optimizer-step speedups; the third and fourth are the time per full training step (the fourth estimated from timed parts); the fifth is validation loss. Sources: Figures 1–3 and their notes in Appendix A.

10.5×	faster optimizer step at 70B-class layer shapes, one RTX PRO 6000 (benchmark, batched implementation, rank 64)
2.5×	faster optimizer step inside a real 1.3B training run
−14%	time and GPU cost per full training step at 1.3B
−35%	less time per training step on DeepSeek V4.1 Flash on 16 B300 GPUs (about 31% on 24), estimated from parts timed on a B300, rank 32, 256k tokens per step, communication not counted (Section 6.3)
4.375	validation loss at 1.3B, within seed noise of Muon (4.3751 ± 0.0115 vs 4.3757 ± 0.0109, three seeds each; AdamW 5.184, one seed)

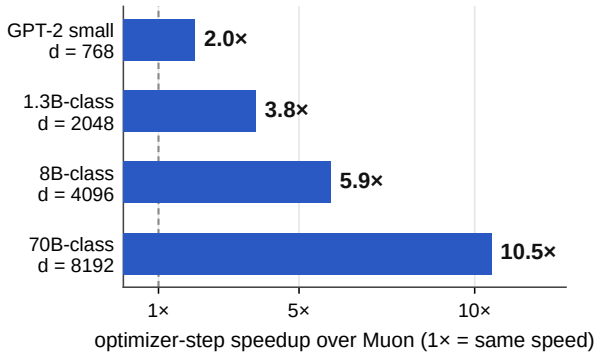


Figure 1. How many times faster the optimizer step is (benchmark, random inputs). Echelon's batched implementation at rank 64 against per-layer Muon over every hidden matrix of a model with the given layer shapes, one NVIDIA RTX PRO 6000 Blackwell. On a B300 the speedups are smaller (Table 9). At width 768, batched Muon is faster than per-layer Muon; against it the benchmark speedup there is 1.4×, and inside the trained GPT-2-small run (rank 96) the optimizer step was 1.1× faster (Table 8). (Details: Appendix A.1.)

Main findings. In summary:

- Muon flattens every singular value, at a cost that grows with the cube of the layer width. Echelon flattens only the top r and scales the rest down, at $O(mnr)$ cost.
- On one GPU, the optimizer step for 70B-class layers drops from 33.9 s to 3.23 s (10.5× at rank 64). At 1.3B the optimizer is 2.5× faster and each training step 14% shorter, with validation loss within seed noise of Muon's (three seeds).
- The gains grow on big MoE models. On DeepSeek V4.1 Flash, our estimate from parts timed on a B300 is about 35% less time per training step on 16 B300 GPUs, the fewest that hold its training state (about 31% on 24), at 256k tokens per step and rank 32, communication not counted (Section 6.3); no MoE was trained.
- The speedup depends on the hardware. Muon's Newton–Schulz step is compute-bound and Echelon's is memory-bound, so on a B300 the whole-model optimizer step of DeepSeek V4.1 Flash is 3.7× faster at rank 32, against 10.7× on the RTX PRO 6000 (Table 9).
- We proved that Echelon's update stays bounded and that it converges. Why Muon still wins on small models comes down to how evenly the update spreads across directions.

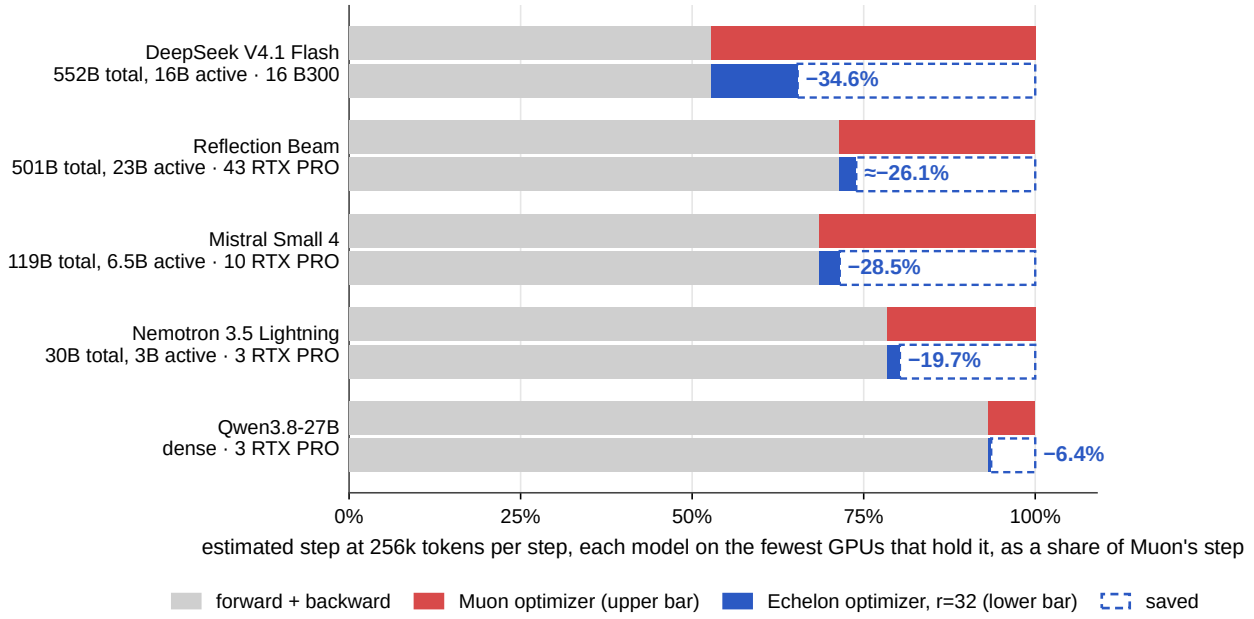


Figure 2. What it does to a training step on today's models (estimated). One training step at 256k tokens per step, each model on the fewest GPUs that hold its training state (DeepSeek V4.1 Flash on 16 B300s, timed on a B300; the others on RTX PRO 6000s, timed there), optimizer work split over those GPUs, Echelon at rank 32, Muon's step = 100%; grey is forward and backward, colour is the optimizer; ≈: forward and backward extrapolated below the measured tokens per GPU. Timed parts in a per-GPU accounting without communication, not training runs (Section 6.3). (Details: Appendix A.2.)

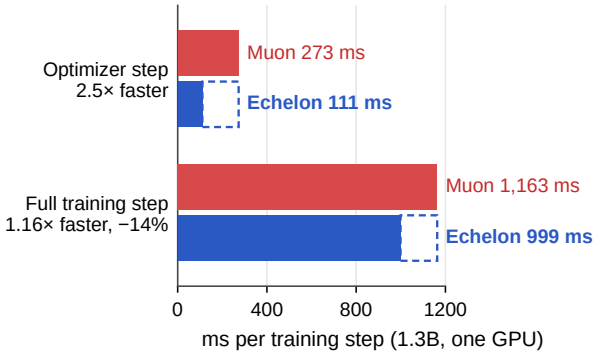


Figure 3. End-to-end, the gain is smaller. Optimizer step and full training step in the 1.3B run, Muon against Echelon (rank 128). (Details: Appendix A.3.)

Organization. Section 2 reviews what Muon computes and what it costs. Section 3 derives Echelon, its spectral-clip form, its stability bounds and its implementation. Section 4 states the machine-checked theorems. Section 5 gives the cost model that says where the speedup turns into end-to-end time. Section 6 reports every measurement. Sections 7 and 8 cover what the theory misses and what did not work. Section 9 collects the limitations and measurement notes, Section 10 related work, and Appendix A the setup and source of every figure and table.

2 Background: what Muon computes and what it costs

This section fixes the objects Echelon approximates: the polar factor that Muon targets, and the Newton–Schulz iteration Muon uses to compute it, with its cost.

2.1 The polar factor

We write the momentum of one weight matrix as its singular value decomposition, $M = U\Sigma V^\top$. Muon wants its polar factor, $\mathrm{polar}(M) = UV^\top$: the same directions, with every singular value replaced by one. The polar factor is the linear-minimization oracle of the spectral-norm ball [4, 21].

There is a good reason for that choice. Among all matrices with spectral norm at most one, the polar factor is the one that lines up best with M:

$$\mathrm{polar}(M) = \arg\max_{\|O\|_2 \le 1} \underbrace{\langle M, O\rangle}_{\text{alignment}}, \qquad \langle M, \mathrm{polar}(M)\rangle = \|M\|_* = \textstyle\sum_i \sigma_i. \tag{1}$$

So it is steepest descent when step size is measured in the spectral norm, which is a natural way to measure how much a linear layer can change its outputs. Throughout, $\langle A, B\rangle = \mathrm{tr}(A^\top B)$, $\|\cdot\|_2$ is the spectral norm, $\|\cdot\|_*$ the nuclear norm (its dual), and $\|\cdot\|_F$ the Frobenius norm. We call $\langle M, O\rangle / \|M\|_*$ the *alignment* of an update O and

$$\kappa(O) = \frac{\langle M, O\rangle}{\|M\|_* \, \|O\|_2} \tag{2}$$

its *score*; the exact polar factor scores 1. Figure 4 shows the geometry for a 2×2 matrix.

2.2 Newton–Schulz and its cost

Computing $UV^\top$ exactly needs an SVD, which is poorly suited to GPUs. Muon instead normalizes M by its Frobenius norm and applies an odd quintic polynomial five times,

$$X \leftarrow aX + b(XX^\top)X + c(XX^\top)^2X, \qquad (a, b, c) = (3.4445, -4.7750, 2.0315). \tag{3}$$

Only matrix multiplies, each of cost proportional to m^2n. The polynomial is tuned for speed rather than accuracy. It does not send singular values to exactly one; it moves them all, large and small, into a band around $[0.7, 1.2]$ (Figure 5). We use this band again in Section 7, "What the theory misses".

For an $s \times l$ matrix with $s \le l$, five iterations cost $5(4s^2l + 2s^3)$ flops, which is $30n^3$ for a square layer of width n. Faster polynomial schedules exist [2]; all of them remain cubic in the width. Counted correctly, Newton–Schulz does 2.5 to 5 times the arithmetic of a classical SVD [8] (Section 4.4); it wins on GPUs because matrix products parallelize, not because it does less work. Muon is used at scale for LLM training [15]; for the background on matrix sign and polar iterations see Higham [11].

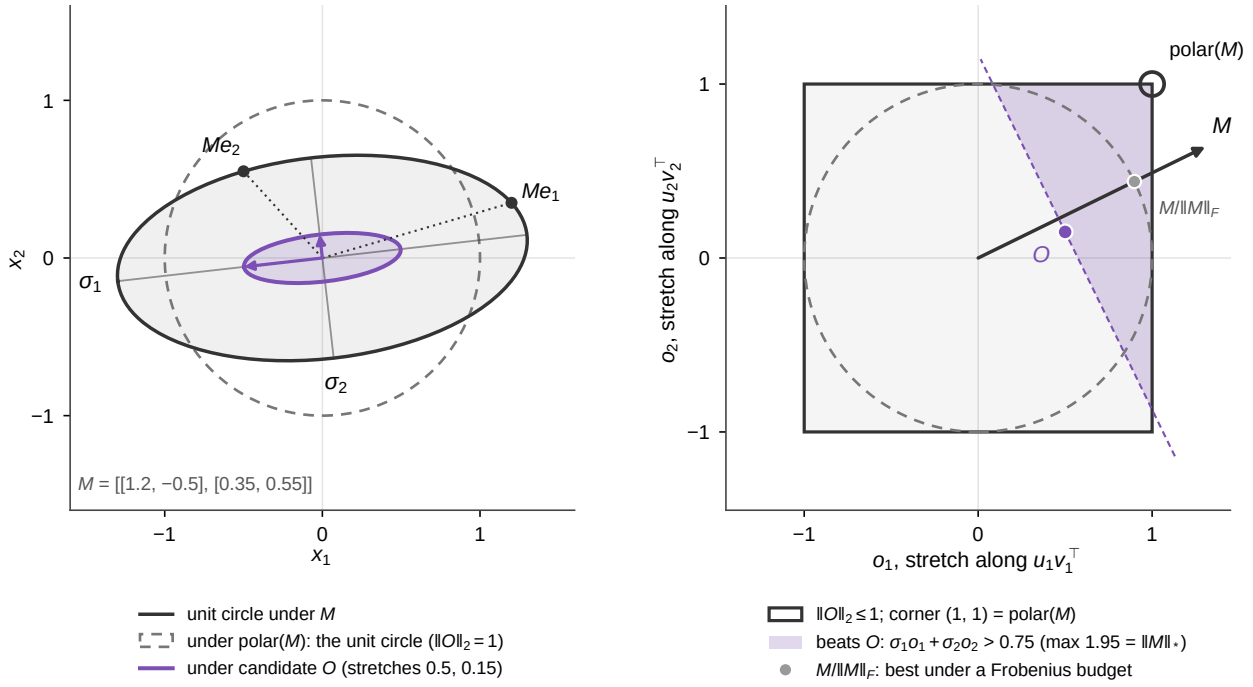


Figure 4. The best update inside the spectral-norm ball is the polar factor. Left: what M, the polar factor and a candidate O do to the unit circle. Right: the same candidate as stretches along M's singular directions; the spectral-norm ball is the square and the best point is its corner $(1, 1)$. Computed, no measurement. (Details: Appendix A.6.)

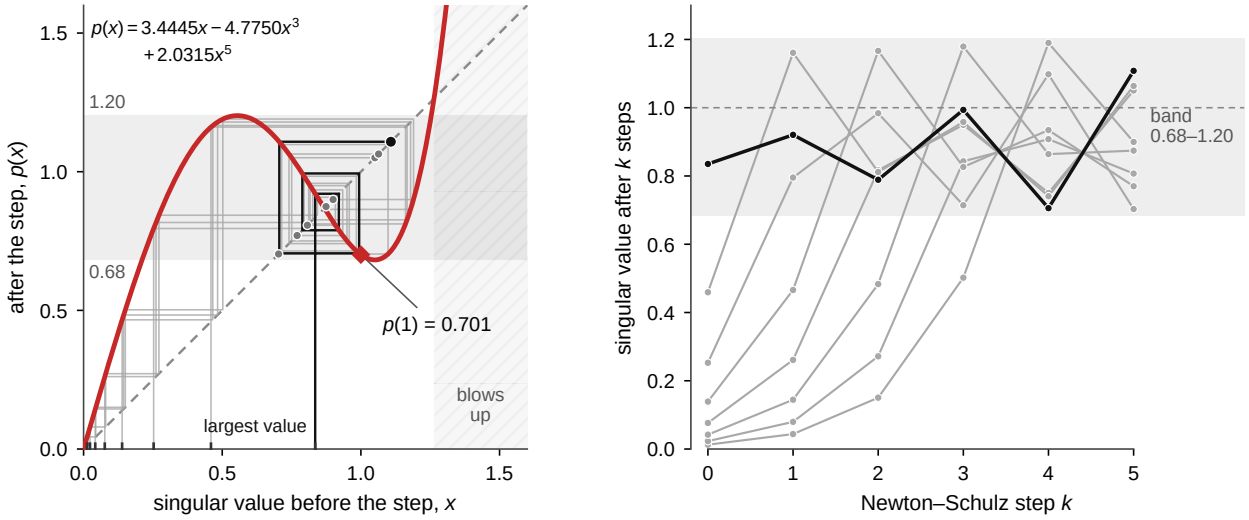


Figure 5. What five Newton–Schulz steps do to singular values. Left: one step $p(x)$ with eight synthetic singular values; right: the same values step by step. Computed, no measurement. (Details: Appendix A.7.)

3 The Echelon optimizer

This section derives Echelon, shows that it is a spectral clip when its frame is exact, bounds its update for any frame, and describes the implementation used in every experiment.

Table 2. Notation: the same objects in the derivation, the implementation and the Lean development.

Object	Derivation (Sections 3.1–3.6)	Code (Algorithm 1, Listing 1)	Lean (Section 4)
Momentum	M, $m \times n$	X, $s \times l$, $s \le l$	X; m_t in the rate theorems
Frame	Q, $m \times r$	U, $s \times r$	U with $U^\top U = I$ exactly
Head factor	$P = \mathrm{polar}(H)$, $H = Q^\top M$	$P = \mathrm{NS5}(H)$, $H = U^\top X$	any P with $\|P\| \le 1$
Tail	$T = M - QH$	$X - UH$, never formed	$\mathrm{tail}(U, X)$
Tail scale	τ	$\tau = 1.05\,\|Tz\|$ after 2 power iterations	any $\tau > 0$ with $\|\mathrm{tail}(U, X)\| \le \tau$
Update	$O = QP + T/\tau$	$O = X/\tau + U(P - H/\tau)$	$\mathrm{htDir}(U, P, \tau, X)$
Momentum rule	—	Nesterov: $\bar{M} \leftarrow \mu\bar{M} + G$, $M \leftarrow G + \mu\bar{M}$	exponential average
Cost model	S speedup (the post writes s), n_{tok} tokens per step	stack of K matrices, (K, s, l)	—

Notation. Table 2 maps the symbols of the derivation, the implementation and the Lean development onto each other. In the derivation (Sections 3.1–3.6) the momentum is $M \in \mathbb{R}^{m\times n}$ with singular values $\sigma_1 \ge \sigma_2 \ge \dots$ and the frame is $Q \in \mathbb{R}^{m\times r}$. The implementation (Figure 6, Algorithm 1) and the Lean development (Section 4) orient each matrix as $X \in \mathbb{R}^{s\times l}$ with $s \le l$ and call the frame $U \in \mathbb{R}^{s\times r}$. Throughout, T is the tail, $\|T\|_2$ its true spectral norm, and τ the scale the tail is divided by. In the code τ is the margin-scaled power-iteration estimate $1.05\,\|Tz\|$, which can fall below $\|T\|_2$ (Section 3.5). Elsewhere f is the optimizer's share of a training step, S the optimizer speedup, n_{tok} the tokens per step and K the number of same-shaped matrices in a stack.

3.1 Splitting the spectrum

Gradients and momenta of trained networks are far from full rank: a few directions carry most of the mass and the rest decay. That suggests treating the top of the spectrum and the rest differently. Echelon keeps one extra piece of state per matrix, an orthonormal frame $Q \in \mathbb{R}^{m\times r}$ that tracks the top-r left singular directions of the momentum from one step to the next. Each step:

1. Refresh the frame with one power step on the current momentum: $Q \leftarrow \mathrm{orth}(MM^\top Q)$. Cost $O(mnr)$.
2. *Head.* Project, $H = Q^\top M$, a small $r \times n$ matrix, and take its polar factor P. It is small, so this is cheap: exact on CPU, five Newton–Schulz steps on the small matrix on GPU. Cost $O(mnr)$.
3. *Tail.* The rest of the momentum is $T = M - QH$. Never store it. Estimate its largest singular value with two power iterations and take τ as that estimate times a 1.05 margin. Cost $O(mn)$.
4. Output $O = QP + T/\tau$. Cost $O(mnr)$.

In total that is $6mnr + 6(m+n)r^2 + 20r^3 + 10mn$ flops against Muon's $5(4m^2n + 2m^3)$: linear in the width for a fixed rank, where Muon is cubic. When $r = m$ the tail is empty and the output is exactly Muon's ideal, the polar factor. Warm-started subspace tracking is also the core of Dion [1]. What differs here is what happens to the head and the tail. Figure 6 shows one step as implemented.

	Operation (stack X of momenta, each $s \times l$, short side first, $s \le l$)	**Precision, cost, fused pass**
in	Momentum X, $s \times l$; fp32 plus a bf16 copy. State: warm frame U ($s \times r$) and vector z ($l \times 1$), kept from the last step.	fp32
1	*Right half-step*: $V = \mathrm{orth}(X^\top U)$, $l \times r$.	bf16 product, fp32 Cholesky-QR; $O(slr)$; pass A
2	*Left half-step*: $U = \mathrm{orth}(XV)$, $s \times r$.	bf16 product, fp32 Cholesky-QR; $O(slr)$; pass B
Head: the top r directions		
3	*Project*: $H = U^\top X$, a small $r \times l$ matrix.	fp32; $O(slr)$; pass C
4	*Polar factor of the head*: $P = \mathrm{NS5}(H)$, five Newton–Schulz steps on $r \times l$ only.	fp32; $O(r^2 l)$; small op
Tail: everything else, never formed		
5	*Largest tail singular value*: power iteration on $T = X - UH$: $w = Xz - U(Hz)$, $z = X^\top w - H^\top(U^\top w)$, $\tau = 1.05\,\|w\|$.	fp32; $O(sl)$ per iteration; passes C–E
6	*Output*: $O = X/\tau + U(P - H/\tau) = UP + T/\tau$.	fp32; $O(slr)$; pass F

Flops per step: Echelon $6slr + 6(s+l)r^2 + 20r^3 + 10sl$; Muon $5(4s^2 l + 2s^3)$. *Memory traffic*: first version about 20 passes, 78 bytes per element; fused kernel 6 passes (A–F), 20 bytes per element.

Figure 6. One Echelon step. Data flows from top to bottom for a stack of same-shaped momenta. (Details: Appendix A.8.)

3.2 Echelon is a spectral clip

Suppose the frame is exactly right, so Q spans $u_1, \ldots, u_r$, and $\tau = \sigma_{r+1}$. Then the head contributes $\sum_{i \le r} u_i v_i^\top$ and the tail contributes $\sum_{i > r} (\sigma_i/\sigma_{r+1})\, u_i v_i^\top$. Put together:

$$O = \sum_i \underbrace{\min\Big(1,\ \frac{\sigma_i}{\sigma_{r+1}}\Big)}_{\text{spectral clip}} u_i v_i^\top. \tag{4}$$

Echelon is a spectral clip. The top r directions get equalized, as in Muon. The rest keep their shape, rescaled so the largest of them is exactly one. The threshold is not a tuned constant: it is set by the rank chosen and moves with the scale of the momentum. At $r = 1$ it is close to normalized gradient descent; at full rank it is Muon (Figure 7).

3.3 Stability under a stale frame: a bound of $\sqrt{2}$

The clip picture needs an exact frame. Ours is refreshed by a single power step per training step, so it is always a little stale. The second property is what makes that safe. For any orthonormal frame, however wrong, the head lives inside the span of Q and the tail lives outside it, so they cannot add up in the same direction:

$$\|Ox\|^2 = \underbrace{\|QPx\|^2}_{\text{head}} + \underbrace{\|Tx\|^2/\tau^2}_{\text{tail}} \le 2\|x\|^2 \quad \Longrightarrow \quad \|O\|_2 \le \sqrt{2}. \tag{5}$$

The identity uses $Q^\top T = 0$, which holds for any Q with $Q^\top Q = I$ because $T = (I - QQ^\top)M$; the inequality uses $\|P\|_2 \le 1$ and $\|T\|_2 \le \tau$. This is Theorem 3, checked in Lean. Both conditions matter: the bound is about the update the code would produce if they held, and Section 3.5 lists how far our implementation meets them.

Under those two conditions a stale frame never costs stability. What it costs is alignment per unit of norm: the raw alignment can even rise slightly at small angles, because a longer update collects more of it, but alignment divided by the spectral norm, the score, falls from the first degree (Figure 8).

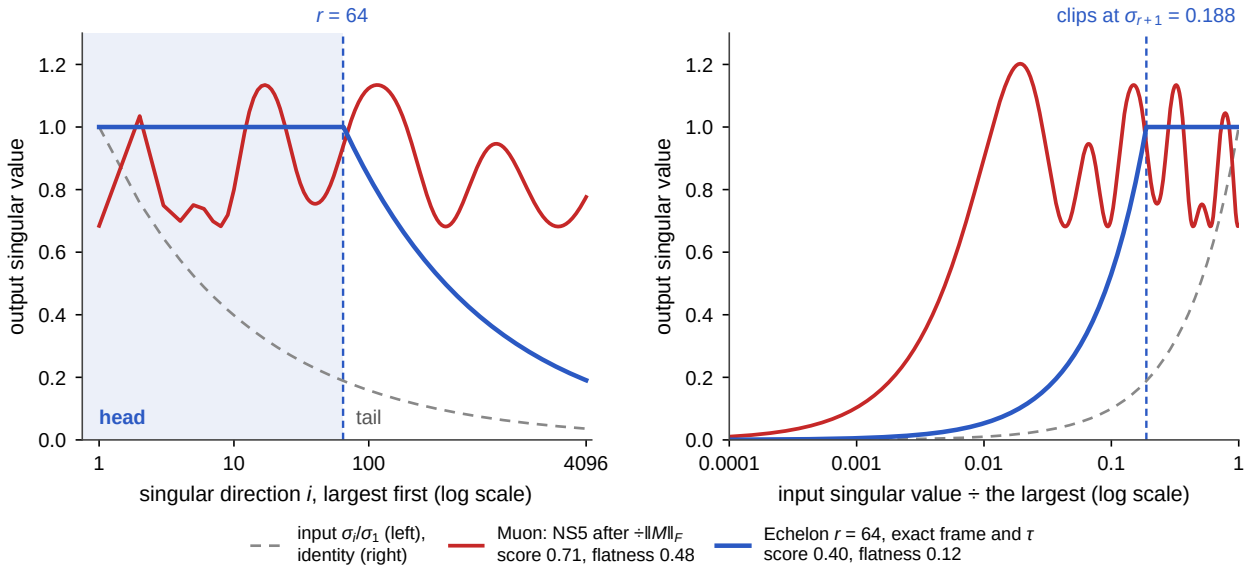


Figure 7. Echelon is a spectral clip at a chosen rank. Synthetic width-4096 momentum, $\sigma_i \propto i^{-0.4}$, $r = 64$, exact frame and exact τ. Left: output singular value per direction. Right: the map each method applies to a singular value. Computed, no measurement. (Details: Appendix A.9.)

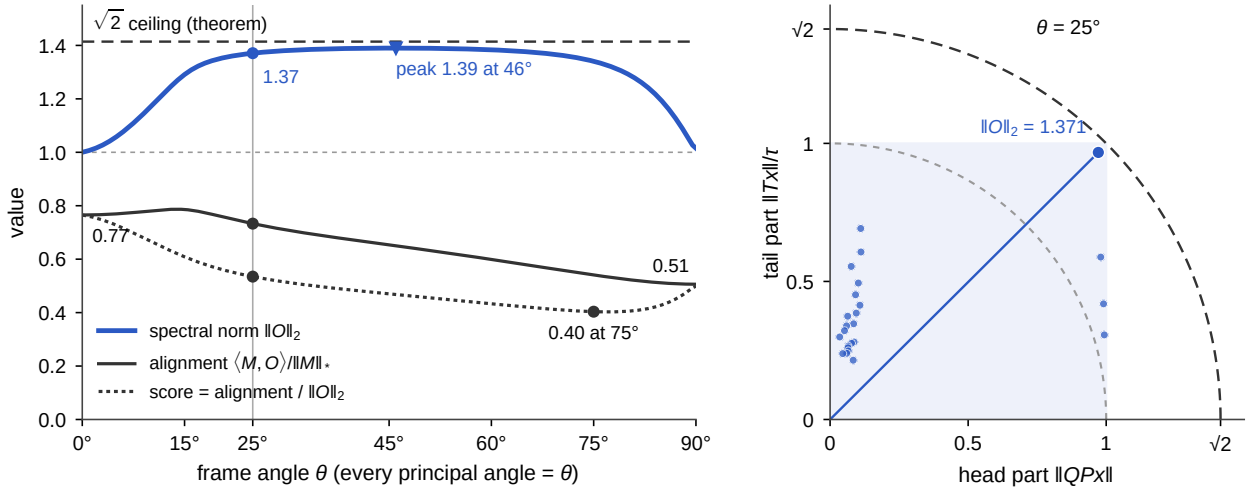


Figure 8. A wrong frame makes the update longer, never by more than a factor of $\sqrt{2}$. Synthetic 24×24 momentum, $\sigma_i \propto i^{-0.7}$, $r = 4$, frame turned by θ. Left: spectral norm, alignment and score against θ. Right: one dot per singular direction of O at $\theta = 25°$. Computed, no measurement. (Details: Appendix A.10.)

3.4 Polishing

To bring the norm back down to one, we apply a single cubic Newton–Schulz step, $O \leftarrow \frac{3}{2}O - \frac{1}{2}OO^\top O$. Its scalar map $\frac{3}{2}x - \frac{1}{2}x^3$ sends all of $[0, \sqrt{3}]$ into $[0, 1]$, so it fixes the norm even when τ underestimates the tail by up to $\sqrt{3}$. We call this polishing (Figure 9; Theorem 4). Polishing costs a full-size $O(s^2 l)$ product, which is why it is off by default.

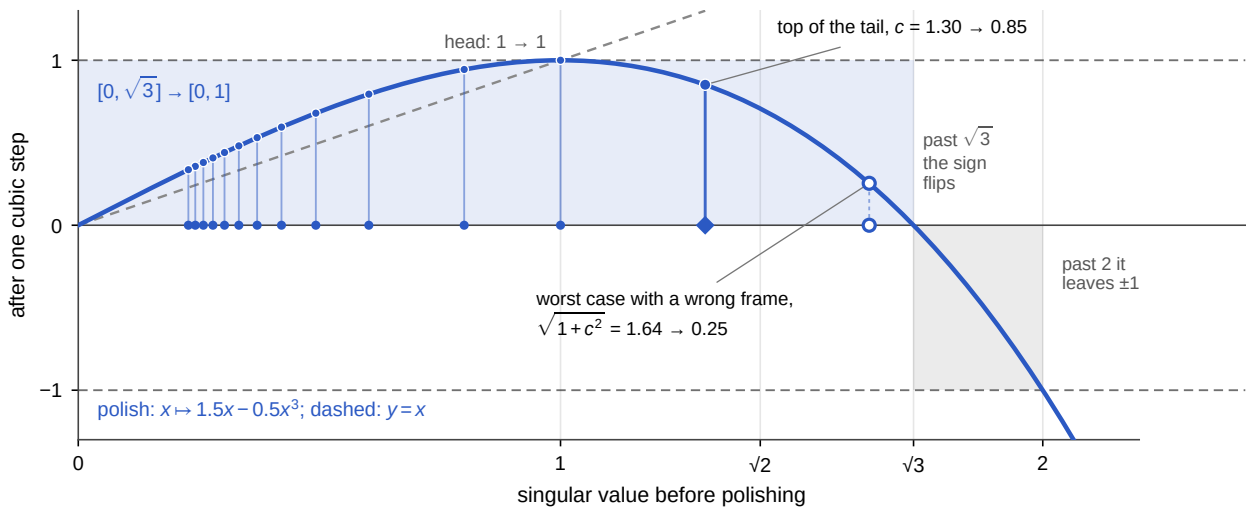


Figure 9. One cubic step pulls the norm back to one. The polish map $x \mapsto 1.5x - 0.5x^3$ acting on the singular values of a synthetic Echelon update whose τ is too small. Computed, no measurement. (Details: Appendix A.11.)

3.5 From the theorem to the code

Table 4 sets each hypothesis of the $\sqrt{2}$ theorem (Theorem 3) against what the code does. When a hypothesis fails, the identity behind the theorem (`htDir_norm_sq_le`) still gives $\|O\|_2^2 \le \|P\|_2^2 + \|T\|_2^2/\tau^2$; with $\|P\|_2 \le 1.25$ and an exact τ that is $\|O\|_2 \le 1.60$, not $\sqrt{2}$.

Measured in training: the hypotheses usually fail. We measured both hypotheses inside two 1.3B Echelon runs (rank 128, the training settings: two warm-started power iterations, $\tau = 1.05\times$ the estimate, NS5 on the head without a cubic step). At steps 0, 1, 2, 5, 10, 20, 50 and every 100th step, on every third layer and all four matrix shapes (864 matrix-steps per run), we compared the estimate with the exact tail norm $\|T\|_2$ in fp32.

- τ is below $\|T\|_2$ in 91.8% of matrix-steps in both runs. The estimate is a median 0.88 of $\|T\|_2$ (1st percentile 0.61–0.63, worst 0.36–0.41, in the first steps), so in steady state τ is typically 5–8% short.
- $\|P\|_2 > 1$ in every matrix-step, up to 1.20.
- The update's spectral norm has a median of 1.21. It exceeds $\sqrt{2}$ in 10.5% and 8.1% of matrix-steps (91 and 70 of 864), almost all in the first 50 steps (after step 50, 4 and 6 of 640); the largest values are 2.77 and 2.38, at step 0, and the 99th percentile is 1.72 and 1.69.

So the $\sqrt{2}$ theorem does not describe our training runs: its hypotheses are not met in about 92% of updates, and its bound is exceeded in about 9%. The bound on $\|P\|_2^2 + \|T\|_2^2/\tau^2$ above still held in all 864 of 864 matrix-steps. More iterations or a larger margin help less than one would hope, because the top of the tail spectrum is flat (median $\|T\|_2/\sigma_{r+1}(X) = 1.17$) and power iteration converges slowly there: from the same warm vector, $\tau < \|T\|_2$ in 84% of matrix-steps with three iterations, 74% with four and 47–50% with six; with a 1.10 margin, 71%, 45–48% and 10% with two, three and six. No cheap setting makes the hypothesis hold reliably. Table 3 shows that at GPT-2-small scale the setting makes no measurable difference to the loss. The defaults of the package we will release are the training settings, so these findings apply to them.

In practice the bound has two conditions: the head factor has spectral norm at most one, and τ is at least the tail's largest singular value. Our code bends both. Five Newton–Schulz steps on the head can overshoot to 1.25, and two power iterations can underestimate τ; the 1.05 margin helps but does not guarantee it. In a stress test with ill-conditioned matrices, bf16 products and rank half the width, the update's spectral norm

Table 3. The tail-estimate setting at GPT-2 small (width 768, rank 96, 1,500 steps, seed 0, one run each, one RTX PRO 6000). Last two columns: share of matrix-steps with $\tau < \|T\|_2$, and matrix-steps (of 1,056 measured) with $\|O\|_2 > \sqrt{2}$. The training setting gave 4.7668 in its original seed-0 run, so loss differences of 0.005 are within run-to-run spread.

Tail estimate	Val loss	Opt. ms	Step ms	$\tau < \|T\|_2$	$\|O\|_2 > \sqrt{2}$
2 iterations, ×1.05 (training setting)	4.7637	11.15	123.64	92.0%	61
3 iterations, ×1.05	4.7688	11.46	124.06	83.4%	13
2 iterations, ×1.10	4.7688	11.28	123.94	73.1%	39

Table 4. The hypotheses of the $\sqrt{2}$ theorem against the implementation.

Hypothesis	Training runs (every loss in this report)	Benchmarks and fused kernel
$U^\top U = I$ exactly	Cholesky-QR with bf16 products and a 10^{-6} shift: approximately	the same; the fused kernel keeps the skinny products in fp32: approximately
$\|P\|_2 \le 1$	$P = \mathrm{NS5}(H)$, which can reach 1.25 (Theorem 6): not guaranteed	one cubic step on the head after NS5: holds up to rounding
$\tau \ge \|T\|_2$	$\tau = 1.05\,\|Tz\|$ after two warm-started power iterations; $\|Tz\| \le \|T\|_2$ always, so it holds only when the estimate is within 5%: not guaranteed	four iterations, same margin: not guaranteed
Exact real arithmetic	bf16 and fp32	bf16 and fp32
Polishing (Theorem 4)	off	off

reached 1.43 and 1.56. A cheap guard, one cubic step on the small head and four tail iterations instead of two, brought those to 1.33 and 1.36. On momentum saved from real training runs, the variant that refreshes the frame before using it, which is what we train with, stayed at or below 1.25; without that refresh it reached 1.52. Inside training, as measured above, the update exceeded $\sqrt{2}$ in about 9% of updates, mostly in the first 50 steps. Polishing the full update is the robust fix, but it costs full-size products, so we leave it off at scale.

The fused kernel (Section 6.7) removes most of the bf16 overshoot by keeping the skinny products in fp32, and it runs the guarded configuration (one cubic step on the head, four tail iterations).

3.6 The alignment given up

The alignment Echelon gives up has a short formula. Since $\langle M, QP\rangle = \|H\|_*$ and $\langle M, T\rangle = \|T\|_F^2$,

$$\langle M, O\rangle \;\ge\; \|M\|_* - \underbrace{\left(\|T\|_* - \frac{\|T\|_F^2}{\tau}\right)}_{\text{alignment given up}}. \tag{6}$$

The loss is zero when the tail is flat and at most the tail's nuclear norm otherwise. Throwing the tail away loses all of it, which is exactly why "head only" was the worst variant in every sweep we ran (Section 8). The Lean version, with an inexact head, is `ht_alignment_lower` (Section 4.2).

Algorithm 1 Echelon, one optimizer step for one hidden matrix $W \in \mathbb{R}^{m\times n}$ (short side s, long side l).

Require: gradient G, momentum buffer $\bar{M}$, frame $U \in \mathbb{R}^{s\times r}$ and vector $z \in \mathbb{R}^{l}$ from the last step, rank r, learning rate η, momentum μ

$\bar{M} \leftarrow \mu\bar{M} + G$; $\quad M \leftarrow G + \mu\bar{M}$ ▷ Nesterov momentum

$X \leftarrow M$ or $M^\top$ so that X is $s \times l$, $s \le l$

$V \leftarrow \mathrm{CholQR}(X^\top U)$; $\quad U \leftarrow \mathrm{CholQR}(XV)$ ▷ one power step, two half-steps

$H \leftarrow U^\top X$; $\quad P \leftarrow \mathrm{NS5}(H)$ ▷ head, $r \times l$

for $k = 1, 2$ **do** ▷ tail power iteration, $T = X - UH$ never formed

  $z \leftarrow z/\|z\|$; $\quad w \leftarrow Xz - U(Hz)$; $\quad e \leftarrow \|w\|$; $\quad z \leftarrow X^\top w - H^\top(U^\top w)$

end for

$\tau \leftarrow 1.05\, e$; $\quad O \leftarrow X/\tau + U(P - H/\tau)$ ▷ $= UP + T/\tau$

optionally polish: $O \leftarrow \frac{3}{2}O - \frac{1}{2}OO^\top O$

$W \leftarrow W - \eta\sqrt{\max(1, m/n)}\; O$ (transposed back if needed)

3.7 Implementation: batching, numerics and the fused kernel

Our first GPU implementation processed one matrix at a time and hit a launch-latency floor of about 1.6 ms per matrix on an L40S, which made Echelon slower than Muon below width 2048. The fix was to stack all same-shaped matrices and handle each stack in one batched call, with bf16 for the thin matrix products and fp32 for everything small. Orthonormalization is Cholesky-QR [6] with a tiny diagonal shift [7], split into two half-steps: our first version factored $MM^\top Q$ in one go, whose Gram matrix squares the singular values twice, and it crashed in fp32 within a minute of training.

Concretely, the batched reference (`ht_batched` in `py/batched.py`) stacks every hidden matrix of one shape into a tensor of shape (K, s, l); computes the two half-step products in bf16 and everything else (Gram matrices, Cholesky factors, triangular solves, head Newton–Schulz, power iteration, output) in fp32; uses two Cholesky-QR passes with a shift of 10^{-6} times the mean diagonal; never synchronizes with the host (`cholesky_ex` with the status ignored); and maps any NaN or infinity in the output to zero, so an all-zero matrix in a stack yields a zero update. It uses no CUDA graphs. On the first call the frame starts from a random matrix refined by three subspace steps.

That implementation was limited by memory traffic, moving about 78 bytes per matrix element in some 20 passes. The fused Triton kernel (`kernels/echelon_fused.py`) does the same math in 6 passes and 20 bytes per element. A program that owns a column strip of X computes one power-iteration half-product and the strip's share of the next product in two sweeps of the same strip, so each power iteration is one pass; the head product rides along with the first power pass; and the first iteration is moved ahead of H by associativity, $Hz_0 = U^\top(Xz_0)$. Every other formula is the reference's. In bf16 mode, the head, the tail iterations and the output all read the same bf16 copy of X (with fp32 accumulation), so the tail stays orthogonal to U for that copy and the $\sqrt{2}$ argument applies to it. Results are in Section 6.7.

3.8 Reference algorithm and settings

Algorithm 1 is one Echelon step on one matrix; Table 5 lists the settings we used.

Listing 1 gives the core of one batched step, for a stack `X` of same-shaped momenta with the short side first. `cholqr` is two passes of Cholesky-QR with a diagonal shift of 1e-6 times the mean diagonal; `newton_schulz5` is Muon's quintic. The first call starts `U` from a random frame refined by three subspace steps. The full code, including the fused kernel, and a package will be released on GitHub soon; the Lean library is available now (Section 12).

Table 5. Using Echelon: the settings behind every result in this report.

Where	The matrices Muon handles: attention and MLP or expert weights. Embeddings, the output head and norms stay on AdamW.
Settings	Muon's carry over unchanged: learning rate 0.02, Nesterov momentum 0.95, each update scaled by the square root of $\max(1, \text{rows}/\text{cols})$, no weight decay. Same warmup and cosine schedule.
Rank	What worked: a quarter of the width at 256 ($r = 64$), an eighth at 768 ($r = 96$) and a sixteenth at 2048 ($r = 128$). Our suggestion is to start at width/16, and not below 64. The large-model timings of Section 6.2 use rank 32, below this; Tables 9 and 11 give ranks up to 512, width/16 included.
Tail	Two power iterations and a 1.05 margin on τ, no cubic step on the head and no polishing in the quality runs, so the conditions of the $\sqrt{2}$ bound are not guaranteed there (Table 4). The benchmarks and the fused kernel run the guarded configuration: one cubic step on the head and four tail iterations.
Memory	The frame Q and the power-iteration vector add $m \cdot r + n$ numbers per $m \times n$ matrix (m the short side): 1.3% more than the weights for a 2304×5120 expert at $r = 64$, 2.5% at $r = 128$.
Implementation	Stack same-shaped matrices and process each stack in one call. Use bf16 for the large thin products and fp32 for everything small. Orthonormalize with Cholesky-QR plus a tiny diagonal shift, as two half-steps.
Sharding	A sharded version runs on the DTensor layouts that FSDP2, tensor and expert parallelism produce: rows, columns, 2-D blocks and expert stacks. On 4 processes it matches single-process Echelon to within 2.1×10^{-6} relative with fp32 products, over three steps and eleven layouts, and to 0.5% with bf16 products, where the reference rounds its partial products to bf16. So far it has run only on CPU.

```
def echelon_step(X, state, r, tau_iters=2, margin=1.05):
    # X: (K, s, l) momenta, s <= l. state keeps U (K, s, r) and z (K, l, 1).
    Xf, Xl = X.float(), X.bfloat16()
    U = state["U"]                                  # warm-started frame
    V = cholqr(Xl.mT @ U.bfloat16())                # right half-step
    U = cholqr(Xl @ V.bfloat16())                   # left half-step
    H = U.mT @ Xf                                   # head, (B, r, l)
    P = newton_schulz5(H)                           # polar factor of the small head
    z = state["z"]
    for _ in range(tau_iters):                      # top singular value of the tail,
        z = z / z.norm(dim=1, keepdim=True)         # without forming T = X - U H
        w = Xf @ z - U @ (H @ z)
        est = w.norm(dim=1, keepdim=True)          # ||T z|| <= ||T||_2
        z = Xf.mT @ w - H.mT @ (U.mT @ w)
    state["U"], state["z"] = U, z
    tau = margin * est                              # the tail scale tau
    return Xf / tau + U @ (P - H / tau)             # = U P + (X - U H) / tau
```

Listing 1. The core of one batched Echelon step (PyTorch).

4 Theory: proofs a machine checked

Optimizer papers prove things on paper, and the paper proofs are usually right in spirit and loose in detail: a matrix inequality waved through, a norm silently swapped for another. We wanted every step checked, so we wrote the argument in Lean 4 [5] with Mathlib [22].

The library has 35 audited theorems. It uses only Lean's three standard axioms and contains no `sorry`. The development uses Lean 4 v4.35.0 with Mathlib at the matching tag. We call a theorem *audited* when `#print axioms` has been run on it; the 35 audited theorems are the ones listed in `Axioms.lean` at the root of the Lean repository (Appendix D), and for every one of them the output lists only `propext`, `Classical.choice` and `Quot.sound`. Table 6 lists the main results; Section 4.2 states them precisely.

Table 6. The main machine-checked results (all in the `FastMuon` namespace). (Details: Appendix A.16.)

Theorem	What it says
`muon_family_rate`	Any update with alignment at least $\alpha\|M\|_*$ and spectral norm at most β reaches Muon's $O(1/\sqrt{T})$ rate; only constants change.
`ht_muon_rate`	Echelon reaches it with $\beta = \sqrt{2}$ and α set by the tail loss.
`htDir_norm_le_sqrt_two`	$\|O\|_2 \le \sqrt{2}$ for any orthonormal frame.
`polished_ht_norm_le_one`	One cubic step gives $\|O\|_2 \le 1$, robust to τ low by up to $\sqrt{3}$.
`muon_norm_le`	Muon's five-step output never exceeds spectral norm 1.25.
`speedup_4096_16`	At least 1000× fewer flops than Newton–Schulz for a 4096×4096 layer at rank 16.
`ns5_spiked_kappa`	On a spiked 4096×4096 spectrum Muon's update scores at most 0.57, while rank-1 Echelon scores 1.

4.1 Setting and definitions

Matrices are real; $\|\cdot\|$ is Mathlib's spectral (ℓ^2-operator) norm. For a frame $U \in \mathbb{R}^{s\times r}$ with $U^\top U = I$, a head factor $P \in \mathbb{R}^{r\times l}$, a scale $\tau > 0$ and $X \in \mathbb{R}^{s\times l}$:

$$\mathrm{tail}(U, X) = X - U(U^\top X), \qquad \mathrm{htDir}(U, P, \tau, X) = UP + \tau^{-1}\,\mathrm{tail}(U, X),$$

$\mathrm{frob}(A, B) = \mathrm{tr}(A^\top B)$, nuc is the nuclear norm (defined as the dual of the spectral norm), $\mathrm{nsP}(x) = 3.4445x - 4.7750x^3 + 2.0315x^5$, $\mathrm{nsStep}(X) = 3.4445X - 4.7750\,XX^\top X + 2.0315\,XX^\top XX^\top X$, $\mathrm{cubicStep}(X) = 1.5X - 0.5\,XX^\top X$, and the cost models $\mathrm{nsCost}(s, l, k) = k(4s^2l + 2s^3)$, $\mathrm{htCost}(s, l, r) = 6slr + 6(s+l)r^2 + 20r^3 + 10sl$, $\mathrm{svdCost}(m, n) = 4mn^2 + 8n^3$ (Golub–Reinsch [8]).

The convergence results are stated for an abstract geometry G (a pairing $\langle\cdot,\cdot\rangle$, a primal norm N and its dual D), instantiated with $N = \|\cdot\|_2$ and $D = \mathrm{nuc}$. Momentum is the exponential average $m_{t+1} = \mu m_t + (1-\mu)\hat{g}_{t+1}$, $m_0 = \hat{g}_0$, with stochastic gradients $\hat{g}_t$ whose error is bounded in the dual norm by σ (worst-case bounded noise), and the iterate is $x_{t+1} = x_t - \eta d_t$.

4.2 Main theorems

Theorem 1 (Certified-direction rate)**.** (Lean: `FastMuon.Geometry.muon_family_rate`.) *Let f be L-smooth with respect to N, bounded below by f_{low}. Let $\eta, \alpha > 0$, $\beta \ge 0$, $0 \le \mu < 1$. If every direction satisfies $\alpha D(m_t) \le \langle m_t, d_t\rangle$ and $N(d_t) \le \beta$, and $D(\hat{g}_t - \nabla f(x_t)) \le \sigma$, then for every $T > 0$*

$$\frac{1}{T}\sum_{t<T} D(\nabla f(x_t)) \;\le\; \frac{f(x_0) - f_{\mathrm{low}}}{\alpha\eta T} + \frac{L\eta\beta^2}{2\alpha} + \frac{\alpha+\beta}{\alpha}\Big(\frac{\mu L\eta\beta}{1-\mu} + \sigma\Big).$$

We call a direction rule (α, β)-*certified* when it satisfies these two hypotheses. With $\eta \propto 1/\sqrt{T}$ the bound is the $O(1/\sqrt{T})$ rate (plus the noise floor) of Muon; Muon's exact polar factor is $(1, 1)$-certified. The rate's constants depend on α and β; the measured scores $\kappa = \langle M, O\rangle/(\|M\|_*\|O\|_2)$ in Table 15 (0.70–0.82 for Echelon, 0.72–0.75 for Muon on saved momentum) are the corresponding ratio α/β per step.

Theorem 2 (Rate for Echelon)**.** (Lean: `FastMuon.Spectral.ht_muon_rate`.) *In the spectral geometry, let $d_t = \mathrm{htDir}(U_t, P_t, \tau_t, m_t)$ with $U_t^\top U_t = I$, $\|P_t\| \le 1$, $\alpha_h\,\mathrm{nuc}(U_t^\top m_t) \le \mathrm{frob}(U_t^\top m_t, P_t)$ (the head is α_h-aligned), $\tau_t > 0$, $\|\mathrm{tail}(U_t, m_t)\| \le \tau_t$, and the tail loss satisfies*

$$\alpha_h\,\mathrm{nuc}(R_t) - \tau_t^{-1}\,\mathrm{frob}(R_t, R_t) \le (\alpha_h - \alpha)\,\mathrm{nuc}(m_t), \qquad R_t = \mathrm{tail}(U_t, m_t),$$

with $0 < \alpha \le \alpha_h$. Then the bound of Theorem 1 holds with $\beta = \sqrt{2}$.

Theorem 3 (Stability under any frame)**.** (Lean: `FastMuon.HeadTail.htDir_norm_le_sqrt_two`.) *If* $U^\top U = I$, $\|P\| \le 1$, $\tau > 0$ *and* $\|\mathrm{tail}(U, X)\| \le \tau$, *then* $\|\mathrm{htDir}(U, P, \tau, X)\| \le \sqrt{2}$. *No assumption is made on how well* U *matches the singular vectors of* X.

The hypotheses are exactly those in the statement: an exactly orthonormal U, any head factor with $\|P\| \le 1$ (it need not be a polar factor), $\tau > 0$ and $\|\mathrm{tail}(U, X)\| \le \tau$, in exact real arithmetic. Table 4 compares them with the code.

Theorem 4 (Polishing)**.** (Lean: `FastMuon.Polish.polished_ht_norm_le_one`, `polished_ht_norm_le_one_robust`.) *Under the hypotheses of Theorem 3,* $\|\mathrm{cubicStep}(\mathrm{htDir}(U, P, \tau, X))\| \le 1$. *The same conclusion holds under the weaker* $\|\mathrm{tail}(U, X)\| \le \sqrt{3}\,\tau$.

Theorem 5 (Alignment given up)**.** (Lean: `FastMuon.Spectral.ht_alignment_lower`.) *If* $U^\top U = I$, $\alpha_h \ge 0$ *and* $\alpha_h \,\mathrm{nuc}(U^\top X) \le \mathrm{frob}(U^\top X, P)$, *then, with* $T = \mathrm{tail}(U, X)$,

$$\alpha_h \,\mathrm{nuc}(X) - \big(\alpha_h \,\mathrm{nuc}(T) - \tau^{-1}\mathrm{frob}(T, T)\big) \le \mathrm{frob}\big(X, \mathrm{htDir}(U, P, \tau, X)\big).$$

With $\alpha_h = 1$ *this is* (6).

Theorem 6 (Muon's overshoot)**.** (Lean: `FastMuon.NSMatrix.muon_norm_le`, `FastMuon.NSPoly.muon_five_steps`.) *For every real matrix* M, $\|\mathrm{nsStep}^5(M/\|M\|_F)\| \le 1.25$. *On scalars, every* x *with* $0.6/243 \le x \le 1.25$ *satisfies* $0.6 \le \mathrm{nsP}^5(x) \le 1.25$.

So Muon itself is only $(\alpha, \beta \le 1.25)$-certified in the sense of Theorem 1.

Theorem 7 (Flop counts)**.** (Lean: `FastMuon.Cost.speedup`, `speedup_unbounded`, `speedup_4096_16`, `speedup_4096_16384_32`.) *For* $1 \le r \le s \le l$: $5s \cdot \mathrm{htCost}(s, l, r) \le 12r \cdot \mathrm{nsCost}(s, l, 5)$; *if moreover* $3Kr \le s$ *then* $K \cdot \mathrm{htCost}(s, l, r) \le \mathrm{nsCost}(s, l, 5)$. *In particular* $1000 \cdot \mathrm{htCost}(4096, 4096, 16) \le \mathrm{nsCost}(4096, 4096, 5)$ *(exact ratio 1,151×) and* $400 \cdot \mathrm{htCost}(4096, 16384, 32) \le \mathrm{nsCost}(4096, 16384, 5)$ *(exact ratio 452×).*

Theorem 8 (Spiked-spectrum separation)**.** (Lean: `FastMuon.Separation.ns5_spiked_kappa`, `ht_spiked_kappa`.) *Consider a* 4096×4096 *spectrum with one singular value 1 and 4095 equal to* $\varepsilon = 4 \cdot 10^{-4}$, *and any Frobenius scale* $1 \le F \le 1/0.99$. *The score of Muon's five-step map,* $\sum_i \sigma_i \phi(\sigma_i) / (\sum_i \sigma_i \cdot \max_i \phi(\sigma_i))$ *with* $\phi = \mathrm{nsP}^5(\cdot/F)$, *is at most 0.57; rank-1 Echelon with the exact top direction and* $\tau = \varepsilon$ *has score exactly 1.*

The bridge from these scalar spectral maps to matrices (spectral calculus) is assumed in this theorem, not formalized.

4.3 Pinned statements

A proof is only as good as its statement, and it is easy to prove something slightly weaker than what you meant. So we pinned seven headline statements in the lake comparator format, which checks that the theorem that compiles is literally the statement a reader sees. It accepts all seven. As a control, we weakened one statement from 1000× to 100× and the comparator rejected it. The comparator format comes from OpenAI's math [17] and ten-proofs [18] releases.

Each challenge module imports only Mathlib, restates the needed definitions word for word, and states the headline theorems; `lake comparator` checks that every named theorem in the real module has exactly the challenge's statement, that every definition the statements mention is the same constant in both, that the proofs use no axiom outside the three permitted ones, and that Lean's kernel accepts the exported proofs. The seven pinned challenges are `muon_family_rate`, `ht_muon_rate`, `htDir_norm_le_sqrt_two`, `speedup` together with `speedup_4096_16`, `muon_norm_le`, `claimed_contraction_false` together with `ns_vs_svd`, and `ns5_spiked_kappa`. The comparator was run on 2026-11-08.

4.4 Three refuted claims about Newton–Schulz

The library also refutes three claims about Muon's Newton–Schulz step from a popular write-up, which stated them without proof:

- "Each step shrinks the error on [0.5, 1.5]." False at the most basic point: a singular value already equal to 1 is sent to 0.701 (`claimed_contraction_false`).
- "Five steps land within 0.05 of 1 on [0.5, 1.5]." False: from 1.5 the iteration diverges, and even from exactly 1 five steps end between 0.65 and 0.90. Muon's quintic is built to be fast, not accurate (`claimed_five_step_false`, `five_steps_from_one_not_close`).
- "Five steps cost fewer flops than an SVD once the matrix is 4096 or larger." False even with the write-up's own cost formulas (8192×4096 is a counterexample). Counted correctly, Newton–Schulz does 2.5 to 5 times the arithmetic of a classical SVD. It wins on GPUs because matrix products parallelize, not because it does less work (`claimed_efficiency_false`, `ns_vs_svd`: $\frac{5}{2}\,\mathrm{svdCost}(l, s) \le \mathrm{nsCost}(s, l, 5) \le 5\,\mathrm{svdCost}(l, s)$ for $s \le l$).

4.5 Pilot: informal optimizer claims tested in Lean

A separate pilot, which tests informal claims about optimizers one at a time in Lean, added a few more facts about Muon:

- No odd polynomial below degree 116 can squeeze $[0.01, 1]$ into $[0.9, 1.1]$. The best quintic misses by 0.919. Since stacking k quintic steps gives degree 5^k, fewer than three steps can never equalize a spectrum with condition number 100 to within 10%.
- More steps are not always better: for singular values $(1, 0.3)$, three Newton–Schulz steps score above 0.80 and five below 0.73.
- Muon does not dominate gradient descent: on 2×2 quadratics it wins on some and loses on others.
- Echelon's alignment never drops below $1/\sqrt{\rho}$, where $\rho = (\sum \sigma_i)^2 / \sum \sigma_i^2$ is the momentum's nuclear rank, for any rank r, and that floor is tight.
- Modeling Muon as exact spectral steepest descent, with diminishing steps and no momentum, its implicit bias on separable data is the spectral-norm max-margin solution.

In total the pilot tested 20 claims: 17 were refuted, 2 proved, and 1 proved with conditions. The pilot also refutes the claim that the score alone sets one-step progress on a quadratic (`kappa_does_not_determine_decrease`) and proves that the best one-step decrease equals $\kappa^2 \|G\|_*^2 / (2\,\mathrm{srank}(d))$, where srank is the stable rank of the direction (`decrease_eq_kappa_srank`); Section 7 uses this.

4.6 What the proofs do not cover

What Lean does not give you is just as important. The rate theorem assumes the tail loss is small, which we measure rather than prove. It also assumes a head factor with spectral norm at most one, which an exact polar factor satisfies, while the GPU code used in training takes five Newton–Schulz steps on the head, which can overshoot to 1.25; and every bound assumes $\tau \ge \|T\|_2$, which two power iterations with a 1.05 margin do not guarantee (Table 4). The proofs cover exponential-average momentum and worst-case bounded noise, while the code uses Nesterov momentum. And Lean proves flop counts, not wall-clock times. Those we had to measure. Section 9.11 collects these points with the rest of the limitations.

5 Where Echelon fits: the cost model

5.1 A faster optimizer saves only the optimizer's share

An optimizer speedup only matters in proportion to how much of the training step the optimizer takes. If Muon's optimizer is a fraction f of each step and Echelon makes it S times faster, the time saved per step is

$$\text{saving} = f\left(1 - \frac{1}{S}\right). \tag{7}$$

However large S is, the saving cannot pass f (Figure 10). GPU time is what one pays for, so the same fraction comes off the compute bill. Two things decide the share f: how wide each matrix is, and how many tokens each matrix sees per step. We measure f as Muon's optimizer time over its step time, and S as Muon's optimizer time over Echelon's.

5.2 Why wider layers

For a square $d \times d$ layer trained on n_{tok} tokens per step, the forward and backward matmuls cost about $6n_{\text{tok}}d^2$ flops, while Muon's Newton–Schulz costs $30d^3$. Their ratio is

$$\frac{\text{orthogonalization}}{\text{training compute}} \approx \frac{30d^3}{6n_{\text{tok}}d^2} = \frac{5d}{n_{\text{tok}}}. \tag{8}$$

So Muon's overhead grows linearly with width at a fixed batch. Echelon replaces $30d^3$ with about $6d^2r$, and the rank it needs grows much more slowly than the width: in our measurements on real momentum it was a quarter of the width at 256 and between a sixteenth and an eighth at 1024 (Section 6.8). Both effects point the same way, which is why Figure 1 climbs from 2.0× (1.4× against batched Muon) to 10.5× as the width goes from 768 to 8192.

5.3 Why mixture-of-experts

A mixture-of-experts (MoE) model replaces each feed-forward block with many expert feed-forward blocks and routes each token to only a few of them. That is what makes MoEs attractive: the model holds many parameters but each token only pays for the few experts it visits, so it gets the capacity of a large model at the per-token compute of a small one.

The optimizer does not get that discount. Every expert's weight matrices are updated every step, so the optimizer's cost scales with the total parameter count, while the training compute scales with the active count. With E experts and k active per token, each expert sees only about kn_{tok}/E tokens per step, and the ratio above becomes roughly

$$\frac{\text{orthogonalization}}{\text{training compute}} \propto \frac{E}{k} \cdot \frac{\min(d, d_{\text{ff}})}{n_{\text{tok}}}. \tag{9}$$

With 256 experts and 8 active, as in several current open models, $E/k = 32$: the orthogonalization share is tens of times larger than in a dense model of the same width. Expert matrices in today's MoEs are also wide in both directions, for example 5120×2560 in a model that holds 21× more total than active parameters. That is where a faster optimizer step turns into end-to-end time.

Three more properties line up with Echelon specifically:

- *Low-rank gradients.* A layer's gradient is a sum of one outer product per token it saw, so an expert that sees fewer tokens per micro-batch has lower-rank gradients, and a small head rank captures more of its momentum.
- *Many identical shapes.* Hundreds of experts with the same shape are exactly what the batched implementation wants: one call per shape group, so launch latency disappears.
- *Bounded even when stale.* Rarely routed experts update their tracked frame less usefully, and the $\sqrt{2}$ bound means a stale frame can cost score but never stability.

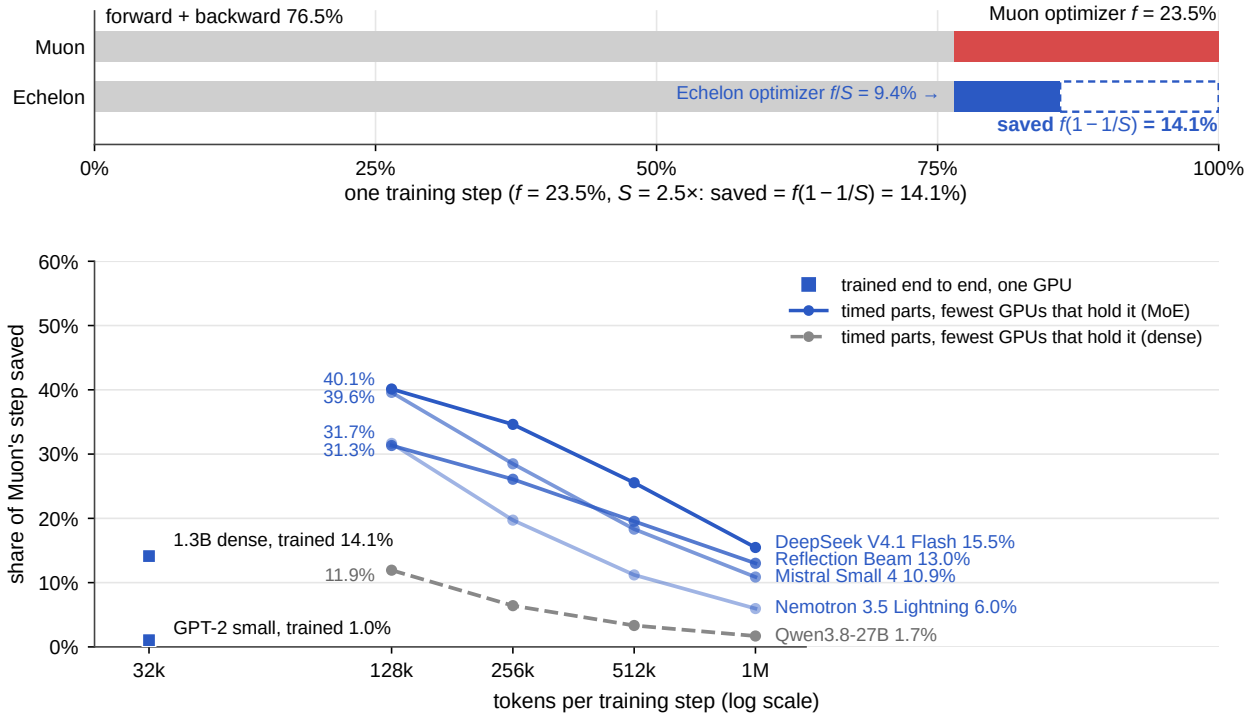


Figure 10. A faster optimizer saves only the optimizer's share. Top: one training step at $f = 23.5\%$, $S = 2.5\times$ (the 1.3B run). Bottom: measured share of the step saved as tokens per step grow, for the two trained runs and the five timed models. (Details: Appendix A.12.)

5.4 Which runs benefit

From the measurements of Section 6:

- Good fit: fine-tuning and RL policy updates of large mixture-of-experts models, at roughly 128k to 1M tokens per optimizer step. That is where the optimizer is a large share of the step: 31–40% saved at 128k, 13–15% at 1M for DeepSeek (16 B300s) and Beam (43 RTX PRO 6000s), each on the fewest GPUs that hold it, rank 32, partly computed outside the measured tokens per GPU (Section 6.3).
- Small dense models at small batches: the 1.3B run saved 14% at 32k tokens per step.
- Little to gain: dense models at large batches (Qwen3.8-27B on 3 RTX PRO 6000s: 6.4% at 256k, 1.7% at 1M), and any model at 4M tokens per step or more (4.6% at most). Pretraining at tens of millions of tokens per step would save less still.
- RL caveat: Echelon speeds up policy updates, not rollout generation, which is usually most of an RL run's GPU time. Against a whole RL run, the saving is the update share times the numbers above.

6 Experiments

We first give the shared setup, then the large-model timings and their dependence on batch size, the optimizer benchmarks and the fused kernel, the training runs, and finally wall-clock time and cost.

6.1 Setup and conventions

Hardware and software. Every timing comes from one NVIDIA RTX PRO 6000 Blackwell Server Edition (an AWS g7e.2xlarge instance), except the first per-layer implementation and the GPT-2-small AdamW baseline, which ran on an NVIDIA L40S. Software: PyTorch 2.14.0 (CUDA 13.0 build), Triton 3.8.0 for the fused kernel, `torch.compile` on the model forward (not the optimizer), bf16 autocast, TF32 matmuls allowed.

Timing conventions. In training runs, *optimizer ms* is the CUDA-synchronized time of all optimizer `step()` calls in a training step (the Muon-family step on hidden matrices plus the AdamW step on the other parameters), averaged over steps 10 to the end; *step ms* is forward, backward (all accumulation micro-steps) and optimizer, averaged the same way, excluding evaluation. The optimizer share is f = optimizer ms / step ms. Optimizer benchmarks time the orthogonalization of every hidden matrix with CUDA events after warm-up calls and report a median.

Loss conventions. Final validation loss is the mean over 50 random validation batches after training; curve points are means over 20 batches, so the two differ slightly (for the 1.3B Muon run, final 4.3880 against a last curve point of 4.3673).

Seeds. The 1.3B comparison of Muon and Echelon has three seeds per optimizer (seeds 0, 1 and 2; seed 1 and 2 change the initialization and the data order); AdamW has one. The GPT-2-small comparison of Muon and Echelon at rank 96 has three seeds each; the polished and AdamW rows have one. The character-model results quoted with ± have three seeds; the sweeps have one. Section 9 lists what this does and does not support.

Models. *GPT-2 small*: width 768, 12 layers, 12 heads, context 1024, untied head, 163M parameters, 1,500 steps of 32,768 tokens. *1.3B*: width 2048, 24 layers, 16 heads, context 1024, untied head, 1.42B parameters (1.21B in hidden matrices), 2,000 steps of 32,768 tokens (16 sequences of 1,024, gradient accumulation 2). Both are GPT-2-style (LayerNorm, GELU MLP of width $4d$, learned positional embedding) trained on FineWeb [19] GPT-2 tokens with a 100-step linear warmup and cosine decay to 0.1×. *Character model*: TinyShakespeare, width 256, 4 layers, context 64, batch 32, 1,000 steps, fp32 on CPU.

6.2 Model-shape timings: five current models

To check the argument of Section 5 on models people train today, we took four mixture-of-experts models and one dense model and timed both optimizers over every matrix Muon would update, on one NVIDIA RTX PRO 6000 Blackwell. We also timed one real layer's forward and backward pass in bf16 at several batch sizes. For DeepSeek V4.1 Flash we repeated both timings on one NVIDIA B300, with all 384 experts allocated. We combined the parts into an estimated training step with each model on the fewest GPUs that hold its training state: DeepSeek V4.1 Flash on 16 B300s, timed on the B300, and the other four on RTX PRO 6000s, timed there. The optimizer work is split evenly over those GPUs, each GPU processes its share of the tokens, and nothing is communicated (Section 6.3). Echelon runs at rank 32: our fused kernel on the RTX PRO 6000 (Section 6.7); on the B300 the faster of the fused kernel and the batched implementation for each shape. That is below the width/16, at least 64, that we suggest for training (Table 5); Section 6.4 gives other ranks. Table 7 gives the parts and Table 8 the resulting shares f, speedups S and savings, next to the two trained runs. The pattern is the one the cost model predicts: the dense 27B model saves the least, and the more experts a model has per active parameter, the more it saves.

Table 7. Measured on real model shapes, each model on the fewest GPUs that hold its training state, timed on that GPU type. Optimizer columns: seconds per GPU per step, whole model split over those GPUs, Echelon at rank 32. The other columns: share of step time saved at that many tokens per optimizer step, summed over all GPUs; ≈: the forward and backward time is computed outside the measured tokens per GPU (extended below the smallest measured batch, or accumulated above the largest). Timed parts without communication (Section 6.3), not training runs. (Details: Appendix A.13.)

Model	GPUs	Muon opt.	Echelon opt.	128k	256k	1M	4M
DeepSeek V4.1 Flash 552B-A16B	16 B300	2.13 s	0.57 s	40.1%	34.6%	≈15.5%	≈4.6%
Reflection Beam 501B-A23B	43 RTX PRO	2.67 s	0.23 s	≈31.3%	≈26.1%	13.0%	4.4%
Mistral Small 4 119B-A6.5B	10 RTX PRO	2.34 s	0.23 s	≈39.6%	28.5%	10.9%	≈3.0%
Nemotron 3.5 Lightning 30B-A3B	3 RTX PRO	2.28 s	0.20 s	31.7%	19.7%	≈6.0%	≈1.6%
Qwen3.8-27B (dense)	3 RTX PRO	3.03 s	0.17 s	11.9%	6.4%	≈1.7%	≈0.4%

Table 8. What it saves: training time and cost. f is Muon's optimizer share of the step, S the optimizer speedup, and Saved = $f(1 - 1/S)$. The GPT-2-small speedup, 1.1×, is measured inside training at rank 96, which is why it differs from the 2.0× and 1.4× benchmark numbers of Figure 1. DeepSeek V4.1 Flash's speedup is the B300's; on the RTX PRO 6000 it is 10.7× (Table 9). (Details: Appendix A.4.)

Model	GPUs	Share f	Speedup S	Saved
Trained end to end, one GPU, 32k tokens per step				
GPT-2 small	1 RTX PRO	10.9%	1.1×	1.0%
1.3B dense	1 RTX PRO	23.5%	2.5×	14.1%
Timed parts, rank 32, 256k tokens per step, each on the fewest GPUs that hold it				
Qwen3.8-27B (dense)	3 RTX PRO	6.8%	17.6×	6.4%
Nemotron 3.5 Lightning 30B-A3B	3 RTX PRO	21.6%	11.5×	19.7%
Mistral Small 4 119B-A6.5B	10 RTX PRO	31.5%	10.4×	28.5%
Reflection Beam 501B-A23B	43 RTX PRO	28.5%	11.7×	≈26.1%
DeepSeek V4.1 Flash 552B-A16B	16 B300	47.3%	3.7×	34.6%

Figure 11 splits each model's step into the forward and backward pass and the optimizer at three batch sizes. Per GPU, DeepSeek V4.1 Flash's step on 16 B300s goes from 3.89 s to 2.33 s at 128k tokens per step, from 4.50 s to 2.94 s at 256k, and from 10.08 s to 8.52 s at 1M. The forward and backward pass was timed on one real MoE or MLP layer with routing forced uniform; on the RTX PRO 6000 it used 91 of DeepSeek's 384 and 81 of Beam's 256 experts and was scaled up (Appendix A.2), and on the B300 all 384 of DeepSeek's experts. The caveats that apply to these rows are collected in Section 9.1.

6.3 What the timed-parts step assumes

The model-shape rows rest on three assumptions. The optimizer work of the whole model, timed on one GPU, is split evenly over N GPUs, as an expert- or ZeRO-sharded optimizer with perfect balance would split it. Each GPU's forward and backward pass is the compute for its $1/N$ of the tokens through the whole model. And nothing is communicated. N is the fewest GPUs that hold the training state. DeepSeek V4.1 Flash has about 554B parameters (549B in the matrices Muon updates), so its bf16 weights alone take 1.11 TB. With bf16 gradients and the fp32 momentum our optimizer timings used, 8 bytes per parameter, the training state is about 4.43 TB: at least 16 B300s (288 GB each) or 47 RTX PRO 6000s (96 GB each), before any activations; with fp32 master weights it is about 6.7 TB.

Table 9. Optimizer speedup on the whole DeepSeek V4.1 Flash model, Echelon against Muon, and the share of a 256k-token training step saved, by rank. Optimizer: whole model on one GPU, seconds per step (median of three; chunks of up to 3 GB scaled to the full count), the faster Echelon implementation per GPU (fused: fused kernel; batched: the batched implementation; f+b: fused on some shapes). Saved: equation (10) on the fewest B300s that hold the model (16; 16k tokens per GPU), a more realistic 24 B300s (10.7k), and the fewest RTX PRO 6000s (47; 5.4k, F extrapolated, ≈). Rank 320 is width/16 (DeepSeek's width is 5120). The ceiling is the saving of an optimizer that cost nothing. Same-day timings on both GPUs.

	B300			RTX PRO 6000		Saved at 256k		
Rank	Echelon		S	Echelon	S	16 B300	24 B300	47 RTX PRO
Muon	34.07 s			115.18 s				
32	9.12 s	f+b	3.7×	10.72 s fused	10.7×	34.6%	30.9%	≈26.6%
64	9.89 s	batched	3.4×	15.77 s fused	7.3×	33.5%	29.9%	≈25.3%
128	12.08 s	batched	2.8×	38.50 s batched	3.0×	30.5%	27.2%	≈19.5%
256	16.95 s	batched	2.0×	51.00 s batched	2.3×	23.8%	21.2%	≈16.3%
320 (width/16)	20.34 s	batched	1.7×	58.52 s batched	2.0×	19.1%	17.0%	≈14.4%
512	28.88 s	batched	1.2×	81.03 s batched	1.4×	7.2%	6.4%	≈8.7%
Ceiling						47.3%	42.1%	≈29.3%

In such an N-GPU run every GPU holds $1/N$ of the optimizer state, does $1/N$ of the optimizer work and processes $1/N$ of the tokens, and the share of a step saved is

$$\text{saved} = \frac{(t_{\text{Muon}} - t_{\text{Echelon}})/N}{F(n_{\text{tok}}/N) + t_{\text{Muon}}/N}, \tag{10}$$

with t the whole-model optimizer time on one GPU and $F(T)$ the forward and backward time for T tokens per GPU. If F grew in proportion to T, N would cancel. It does not: on the B300, DeepSeek V4.1 Flash's F is 1.76 s at 8,192 tokens per GPU, 1.95 s at 10,923, 2.37 s at 16,384 and 3.98 s at 32,768, so part of it does not shrink with the batch, and more GPUs at the same global batch save less. The B300 points cover 16 and 24 GPUs at 256k tokens per step. On the RTX PRO 6000 we did not time F below 16k tokens per GPU; rows there extend the straight line through the two smallest measured points and are extrapolations. No row counts communication: expert-parallel and FSDP traffic lengthen both steps and lower the saving, while Muon's gathering of whole matrices for Newton–Schulz would favour Echelon.

The speedup depends on the hardware. Table 9 times the same optimizer step on both GPUs. Muon's Newton–Schulz step is bound by arithmetic, and the B300 has far more bf16 throughput: it runs 3.4× faster there (34.07 s against 115.18 s). Echelon is bound by memory traffic and gains only 1.2× at rank 32 to 2.9× at rank 320 (measured copy bandwidth 6,294 GB/s on the B300 against 1,461 GB/s on the RTX PRO 6000). So the speedup at rank 32 falls from 10.7× on the RTX PRO 6000 to 3.7× on the B300, and to 1.7× at width/16. The fused kernel was tuned for the RTX PRO 6000; on the B300 it is slower than the batched implementation on the expert matrices (0.204 against 0.179 ms per 2304×5120 matrix at rank 32, 0.320 against 0.201 ms at rank 64) and wins only on the 32768×1280 projections at rank 32. The step saving holds up better than the speedup, because on a 16- to 24-GPU B300 run Muon is still 42–47% of the step at 256k tokens: Echelon at ranks 32 to 128 saves 27–35% of it, and 17–19% at width/16. We have not trained an MoE with Echelon or timed a multi-GPU step.

Table 10 applies the same accounting to all five models at ranks 32 and 64, each on the fewest GPUs that hold its training state, at 256k tokens per step: DeepSeek V4.1 Flash saves 34.6% on 16 B300s, Reflection Beam about 26% on 43 RTX PRO 6000s, Mistral Small 4 28.5% on 10, Nemotron 3.5 Lightning 19.7% on 3 and the dense Qwen3.8-27B 6.4% on 3.

Table 10. Each model on the fewest GPUs that hold its training state (bf16 weights and gradients and fp32 momentum, 8 bytes per parameter; activations not counted), 256k tokens per step, from equation (10). DeepSeek V4.1 Flash on B300s (288 GB), timed there; the others on RTX PRO 6000s (96 GB). Rows marked ≈ extrapolate F below 16k tokens per GPU; the others use measured or interpolated points. Tokens per GPU with k = 1,024.

					Saved	
Model	Params	State	GPUs	Tokens/GPU	$r = 32$	$r = 64$
DeepSeek V4.1 Flash	554B	4.43 TB	16 B300	16k	34.6%	33.5%
Reflection Beam	507B	4.06 TB	43 RTX PRO	6.0k	≈26.1%	≈25.0%
Mistral Small 4	119B	0.96 TB	10 RTX PRO	25.6k	28.5%	26.8%
Nemotron 3.5 Lightning	32B	0.25 TB	3 RTX PRO	85.3k	19.7%	18.7%
Qwen3.8-27B (dense)	33B	0.26 TB	3 RTX PRO	85.3k	6.4%	6.3%

Table 11. The model-shape estimate at other ranks, RTX PRO 6000 (same-day re-timing; DeepSeek V4.1 Flash is in Table 9). Speedup S of the faster Echelon implementation over Muon, whole model on one GPU, and the share of the 256k-token step saved on the fewest RTX PRO 6000s that hold the model, from equation (10). Width/16 is 320 for Beam and Qwen3.8-27B, 256 for Mistral Small 4 and 168 for Nemotron 3.5 Lightning. At rank 32 the re-timing is within 0.1 points of Table 10.

		Speedup S			Saved at 256k		
Model	GPUs	$r = 32$	$r = 64$	width/16	$r = 32$	$r = 64$	width/16
Reflection Beam	43	11.8×	8.2×	2.3×	≈26.2%	≈25.1%	≈15.9%
Mistral Small 4	10	10.4×	6.7×	2.0×	28.5%	26.8%	16.1%
Nemotron 3.5 Lightning	3	11.5×	7.6×	2.6×	19.7%	18.7%	13.3%
Qwen3.8-27B (dense)	3	17.7×	13.5×	4.6×	6.4%	6.3%	5.3%

6.4 The rank of the large-model timings

The timings above use rank 32. Table 9 gives DeepSeek V4.1 Flash at ranks 32 to 512 on both GPUs, and Table 11 the other four models at ranks 32, 64 and width/16, re-timed on the RTX PRO 6000 on the same day. Rank 64 costs the fused kernel well under twice the time of rank 32, because the kernel is bound by memory traffic, which does not depend on the rank. On DeepSeek's shapes, from rank 128 up, the fused kernel is slower than the batched implementation, or does not fit in shared memory, and the batched implementation's cost grows with the rank. At width/16 the speedup of these four models falls to 2.0–4.6× and the saving to 5.3–16.1%, against 6.4–28.5% at rank 32. The quality evidence for the rank is in Sections 6.8 and 6.9; we have no quality measurement at these models' scale.

6.5 Batch-size dependence

The optimizer's work is the same at every batch size, while the forward and backward pass grows with the tokens per step, so the saving falls roughly as one over the tokens per step (Figure 12). At 1M tokens per step the savings in Table 8's order are 1.7%, 6.0%, 10.9%, 13.0% and 15.5%; at 128k they are 11.9%, 31.7%, 39.6%, 31.3% and 40.1%; at 4M they are at most 4.6%. At 1M tokens per step Muon's optimizer share is 1.8%, 6.5%, 12.0%, 14.2% and 21.1% for the same five models. The MoE models keep the largest share because the optimizer updates every expert while each token pays for only a few.

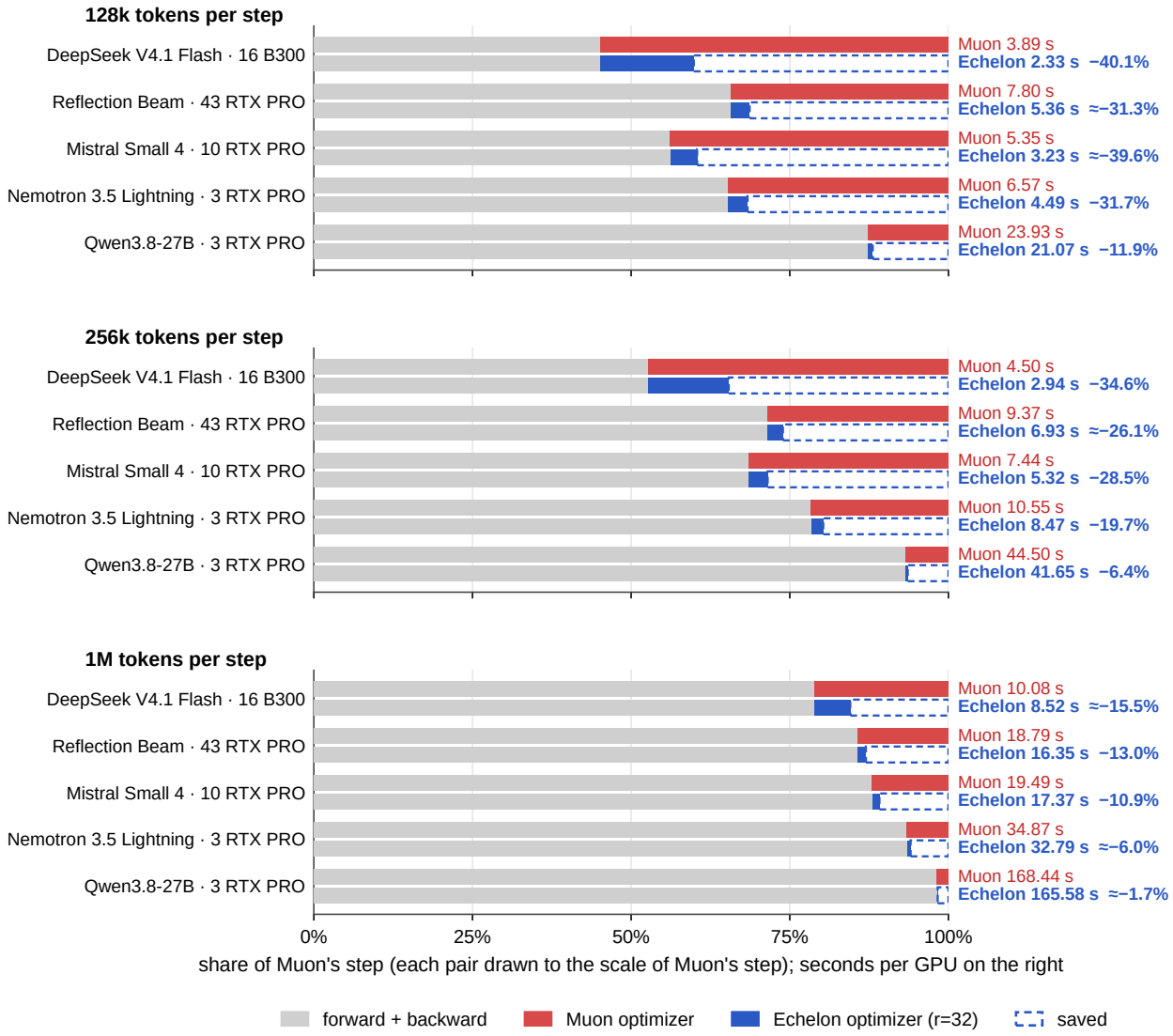


Figure 11. Where the time goes, by batch size (estimated). One training step per GPU, each model on the fewest GPUs that hold it (DeepSeek V4.1 Flash on 16 B300s, the others on RTX PRO 6000s), forward and backward (grey) and optimizer (colour), Muon against Echelon at rank 32, at 128k, 256k and 1M tokens per step; ≈: forward and backward computed outside the measured tokens per GPU. (Details: Appendix A.14.)

6.6 Speed in detail: per-layer optimizer time and the flop count

Lean proves at least 1000× fewer flops for a 4096×4096 layer at rank 16. Wall-clock is a different story: on a GPU the step is bound by memory traffic and kernel launches, so the real saving is one to two orders of magnitude smaller than the flop count. Flop counts are a poor guide at small width in particular: Section 3.7 describes the per-matrix launch-latency floor that made the first implementation slower than Muon below width 2048, and the batching that removed it.

Table 12 and Figure 14 give the times. In the same benchmark, against batched Muon, Echelon at the larger rank 128 is 1.15×, 3.3×, 5.6× and 10.2× faster. Figure 13 sets the measured speedups next to the flop ratio; the measured gain is one to two orders of magnitude below it, for the reason given above.

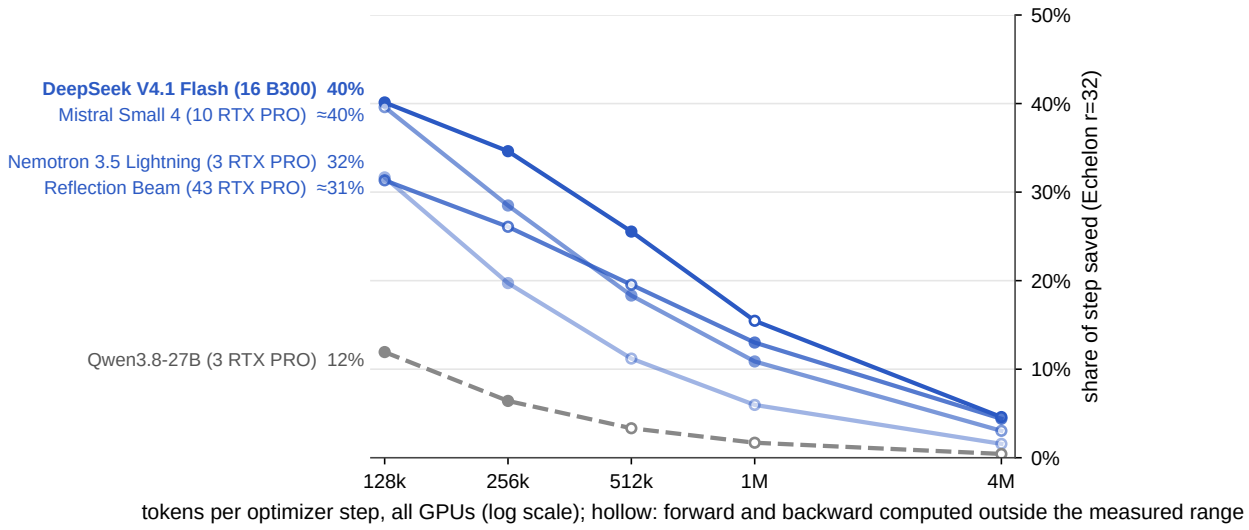


Figure 12. The saving is largest for big MoE models at small batches, and falls as the batch grows. Share of the estimated step saved (Echelon at rank 32), each model on the fewest GPUs that hold it, at 128k to 4M tokens per step; dashed: the dense model; hollow points: forward and backward computed outside the measured tokens per GPU. (Details: Appendix A.15.)

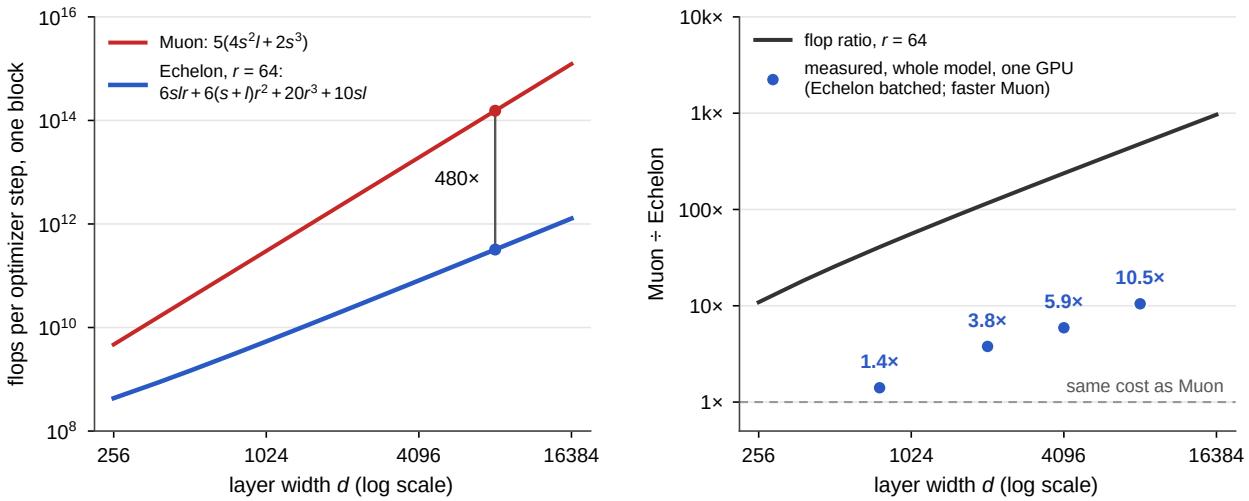


Figure 13. The flop count grows apart with the width; the GPU gain follows, further down. Left: flops per optimizer step for one transformer block, from the cost model. Right: the flop ratio (line) against the measured whole-model speedup on one GPU (dots). Both panels use log scales. (Details: Appendix A.17.)

The per-matrix variant. Version 1 of this report led with 13.2× (33.9 s to 2.56 s) at 70B-class shapes and 6.4× at 8B-class shapes. Those two numbers came from a per-matrix implementation, one call per matrix, that factors $XX^\top U$ with Cholesky-QR in a single power step. In this benchmark, with fp32 inputs, it is faster than the batched implementation at widths 4096 and 8192 (0.29 s against 0.31 s and 2.56 s against 3.23 s). We dropped it because no training run uses it: in training, that one-shot factorization lost positive definiteness and the per-matrix GPU version crashed within the first minute (Section 8). The batched implementation, with two orthonormalized half-steps, is the one behind every training result, so every speedup in this version is measured with it.

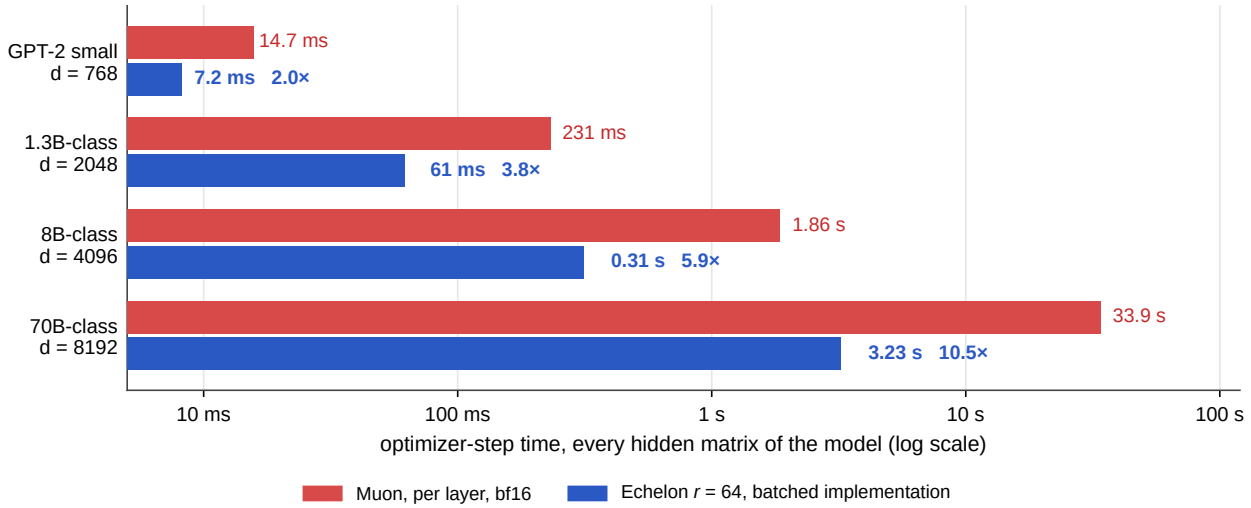


Figure 14. Optimizer-step time for every hidden matrix of a model with the given layer shapes, Muon per layer in bf16 against Echelon's batched implementation at rank 64, one RTX PRO 6000 Blackwell, log scale. (Details: Appendix A.18.)

Table 12. Optimizer-step time for every hidden matrix, one RTX PRO 6000 Blackwell (the benchmark of Figure 14; random inputs, orthogonalization only), Echelon's batched implementation. The first speedup column is against per-layer Muon, the second against the faster of per-layer and batched Muon (batched is faster only at $d = 768$); Table 18 uses the second. (Details: Appendix A.19.)

	Muon	Echelon	Speedup vs Muon		Echelon $r = 32$
Layer shapes	per layer	$r = 64$	per layer	faster	+ polish
GPT-2 small ($d = 768$)	14.7 ms	7.2 ms	2.0×	1.4×	8.9 ms
1.3B-class ($d = 2048$)	231 ms	61 ms	3.8×	3.8×	116 ms
8B-class ($d = 4096$)	1.86 s	0.31 s	5.9×	5.9×	0.73 s
70B-class ($d = 8192$)	33.9 s	3.23 s	10.5×	10.5×	9.8 s

How much of this reaches end-to-end training depends on how large a share of the step the optimizer takes. At GPT-2-small scale it is small: Newton–Schulz was 34 ms of a 252 ms step on an L40S, so even an infinitely fast orthogonalizer would make training at most about 15% faster there. The case for Echelon is wide layers.

6.7 A fused kernel

On the real model shapes above, our first batched implementation was only 3.6–6.5× faster than Muon, far from what its flop count allows. Newton–Schulz is limited by arithmetic, which GPUs do fast; that Echelon was limited by memory traffic, moving about 78 bytes per matrix element in some 20 passes. So we wrote a fused Triton kernel that does the same math in 6 passes and 20 bytes per element. It runs at 86–92% of the GPU's measured copy bandwidth on expert-sized matrices (Table 13).

The kernel's output matches the first implementation to within about 1%, and its largest observed spectral norm on expert matrices was 1.02. It has not been used in a training run yet: every loss in this report comes from the first implementation. At rank 64 the fused kernel is 6.7–13.5× faster than Muon on the same shapes (Tables 9 and 11). These are RTX PRO 6000 numbers; on the B300 the speedups are smaller (Table 9).

The kernel also fixes a numerical weak spot. In the ill-conditioned bf16 stress test (rank half the width), the first implementation reached $\|O\|_2 = 1.47$ on the GPU, above $\sqrt{2}$, because it rounds the output of its

Table 13. Fused kernel: whole model on one RTX PRO 6000, rank 32. (Details: Appendix A.20.)

Whole model, one GPU, $r = 32$	Muon	First version	Fused	vs Muon
DeepSeek V4.1 Flash	117.3 s	30.8 s	10.8 s	10.8×
Reflection Beam 501B	115.0 s	27.9 s	9.9 s	11.7×
Mistral Small 4	23.4 s	6.6 s	2.3 s	10.4×
Nemotron 3.5 Lightning 30B	6.8 s	1.7 s	0.60 s	11.5×
Qwen3.8-27B (dense)	9.1 s	1.4 s	0.51 s	17.6×

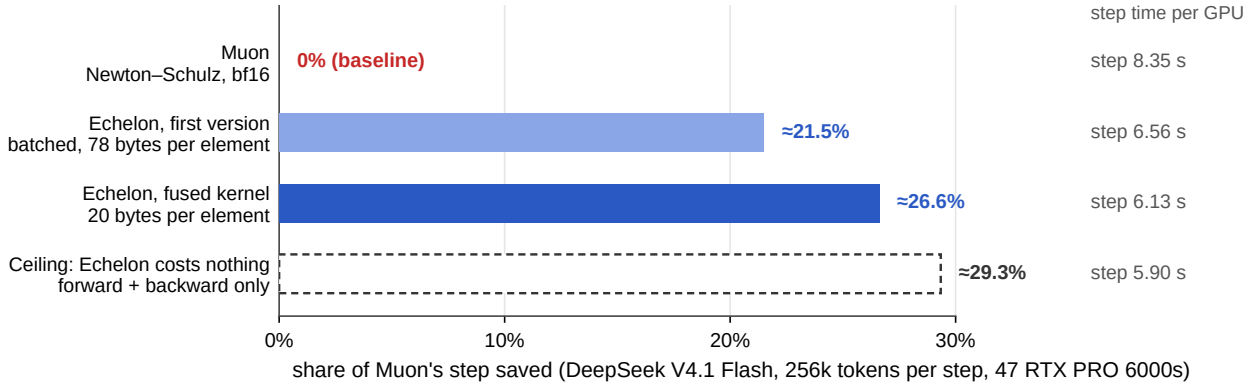


Figure 15. How close to free. DeepSeek V4.1 Flash, 256k tokens per step, on the 47 RTX PRO 6000s that hold it: share of Muon's step saved by each version of the Echelon kernel at rank 32 (estimated; forward and backward extrapolated). (Details: Appendix A.21.)

skinny products to bf16 before Cholesky-QR and so drops weak head directions into the tail. The fused kernel keeps those products in fp32 and stayed at 1.02–1.13 with fp32 X and 1.00–1.09 with bf16 X.

Figure 15 shows what the kernel does to a training step on the GPU it was built for. For DeepSeek V4.1 Flash at 256k tokens per step on the 47 RTX PRO 6000s that hold it (same-day timings; forward and backward extrapolated below the measured 16k tokens per GPU), the first version saves about 21.5% of Muon's step and the fused kernel about 26.6%. An optimizer that cost nothing would save about 29.3%, because the forward and backward pass stays. A kernel that reads and writes each matrix only once, 8 bytes per element, is not built. On the B300 the fused kernel brings little: there the batched implementation is faster on the expert matrices (Section 6.3).

6.8 Training at small width: GPT-2 small and a character model

We trained a GPT-2-small-shaped model (width 768, 12 layers; 163M parameters with an untied output head) on FineWeb for 1,500 steps of 32k tokens (Table 14).

At rank 96, one eighth of the width, Echelon keeps 87% of Muon's improvement over AdamW. It is worse than Muon on all three seeds, by 0.065 on average. On a much smaller 4-layer character model of width 256, averaged over three seeds, rank 64 keeps about two thirds (Muon 1.561 ± 0.012, Echelon 1.587 ± 0.007, AdamW 1.639 ± 0.009). Across our three trained comparisons the gap is smaller at larger width, but those runs differ in more than width (Section 9.9), so this is not a controlled width trend.

One reason to expect the gap to shrink with width is that the rank needed grows much more slowly than the width. We saved real momentum from Muon training runs of a small model at widths 256, 512 and 1024 and scored each method against an exact SVD. Echelon matches Muon's score at rank 64 for widths 256 and 512, a quarter and an eighth of the width, and needs between 64 and 128 at width 1024, a sixteenth to an eighth. Over the same range the momentum's effective rank grew only about 1.2× per doubling of width.

Table 14. GPT-2 small: how much of Muon it keeps. Batched optimizers on Blackwell. The polished row is compared with Muon's seed 0 (4.706). The AdamW baseline is from the matching L40S run. Muon at twice the learning rate diverged. (Details: Appendix A.22.)

Optimizer	Val loss	Share of Muon's gain over AdamW
Muon (3 seeds)	4.6975 ± 0.0024	100%
Echelon, $r = 96$ (3 seeds)	4.7624 ± 0.0027	87%
Echelon, $r = 32$ + one polish step (1 seed)	4.801	81%
AdamW (best of two learning rates, 1 seed)	5.199	0%

Table 15. Rank needed against width, on momentum saved from CPU Muon runs (4 layers). Score = mean of $\langle M, O\rangle/(\|M\|_*\|O\|_2)$ over 6 samples (minimum in parentheses); exact polar factor = 1. Source: `results/cert_offline*.txt` (`data/cert_offline.csv` in the data bundle).

Width	Effective rank ρ (three layers)	Muon score	Best Echelon score
256	94.1 / 61.3 / 97.0	0.750 (0.724)	$r = 64$: 0.768 (0.700)
512	119.6 / 75.7 / 117.7	0.739 (0.724)	$r = 64$: 0.737 (0.647)
1024	144.3 / 82.7 / 133.6	0.735 (0.721)	$r = 128$: 0.824 (0.676); $r = 64$: 0.705 (0.579)

Table 15 gives the measurements: the effective (nuclear) rank $\rho = (\sum \sigma_i)^2/\sum \sigma_i^2$ of the last saved momentum of three layers, and the mean score (2) of Muon and of Echelon with one fresh subspace step, over six samples per method.

6.9 Training at 1.3B

At 1.3B parameters the gap was within seed noise. We trained a GPT with width 2048 and 24 layers on FineWeb for 2,000 steps of 32k tokens (about 65M tokens, three seeds per optimizer, one Blackwell GPU with gradient accumulation); Table 16 has the results. AdamW here runs on all parameters at learning rate 6e-4, which we did not tune at this size.

Rank 128 is one sixteenth of the width. Over three seeds Echelon's final validation loss is 4.3751 ± 0.0115 and Muon's 4.3757 ± 0.0109; paired by seed, Echelon minus Muon is −0.0006 ± 0.0017 (+0.0003, +0.0005 and −0.0025 for seeds 0, 1 and 2), against a seed-to-seed standard deviation of about 0.011. We read this as a match within seed noise over three seeds, not as a measured share of Muon's gain. Echelon reaches AdamW's last curve point (5.168) at step 681.5 ± 2.9, after 11.33 ± 0.04 minutes of training, Muon at step 677.0 ± 6.8 after 13.11 ± 0.13 minutes, against 30.5 minutes for AdamW's whole run: roughly 2.7× less wall-clock time and GPU cost than AdamW (Figures 17 and 18 show seed 0). Because AdamW's learning rate was not tuned and AdamW has one seed, read that 2.7× as an upper bound. These are short runs, about 0.2% of a compute-optimal token budget for this size, so they are early evidence rather than proof. It is consistent with the rank measurements, but the comparison with the smaller runs is confounded (Section 9.9). The optimizer step drops from 273 ms to 111 ms (2.5×; it includes the AdamW step on the other parameters, the momentum update and the stacking), and the full step from 1,163 ms to 999 ms (1.16×, 14% less time and GPU cost; Figure 3); over three seeds the optimizer is 2.46× faster and the step 14.2% shorter.

Table 16. 1.3B on one RTX PRO 6000 Blackwell. Muon and Echelon: mean ± sample standard deviation over seeds 0, 1 and 2 (seed 0 in brackets, the run of Figures 3, 16 and 18); AdamW: seed 0. Tokens per second is 32,768 divided by the step time (seed 0). (Details: Appendix A.23.)

Optimizer	Val loss	Optimizer ms/step	Full step ms	Tokens/s
Muon	4.3757 ± 0.0109 (4.3880)	273.0 ± 0.3	1,161.8 ± 1.4	28,170
Echelon, $r = 128$	4.3751 ± 0.0115 (4.3883)	110.8 ± 0.1	997.2 ± 1.6	32,806
AdamW (one learning rate)	5.184	28	915	35,796

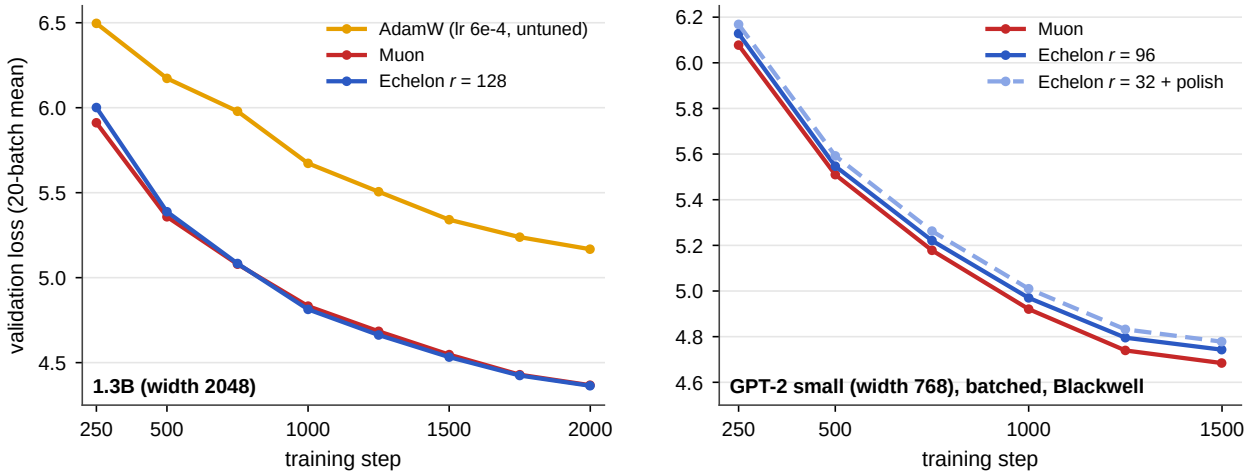


Figure 16. Validation loss during training, from step 250, seed 0 of each optimizer; 20-batch evaluations every 250 steps. Left: 1.3B. Right: GPT-2 small (batched, Blackwell). (Details: Appendix A.24.)

6.10 Wall-clock time and cost

Steps are not what one pays for. Plotted against wall-clock time, Echelon's cheaper step moves its whole curve to the left: at 1.3B it finishes the same 2,000 steps 5.5 minutes sooner than Muon at the same loss (Figure 17). On one GPU, cost is wall-clock time multiplied by the hourly price, so the same picture is also a cost curve; the figures use an illustrative $3 per GPU-hour. In GPU-hours the 2,000 steps took 0.555 for Echelon and 0.646 for Muon (2,000 times the mean step time on one GPU, without evaluation). Figure 18 gives the time to reach AdamW's final curve point (5.168), for seed 0: 11.3 minutes for Echelon, 13.0 for Muon and 30.5 for AdamW (11.33 ± 0.04 and 13.11 ± 0.13 over three seeds), about 2.7× sooner than AdamW and 13% sooner than Muon.

Estimated curves for the five large models. Panels (c) and (d) of Figure 17 are estimates, built as follows.

- *Loss per step*: the measured 1.3B Muon and Echelon validation curves, unchanged (the 8 points at steps 250 to 1999 of the 2,000-step run in Figure 16).
- *Time per step*: that model's estimated training-step time at 256k tokens per step on the fewest GPUs that hold it (Section 6.3; DeepSeek V4.1 Flash on 16 B300s, the others on RTX PRO 6000s), the timings of Figures 2 and 11. Muon: the timed forward and backward pass plus Muon per GPU. Echelon: the same pass plus Echelon at rank 32 per GPU.
- x = step × step time; cost = GPUs × hours × the price per GPU-hour.

The estimate assumes Echelon keeps tracking Muon step for step at that scale, and the loss is the 1.3B run's, not the model's: the estimated curves show time and cost, not that model's loss. Table 17 gives the end points.

Table 17. Estimated 2,000-step runs at 256k tokens per step and $3 per GPU-hour, each model on the fewest GPUs that hold it (method in Section 6.10). Step times per GPU from the timed parts of Table 7.

Model	GPUs	Muon step	Echelon step	Muon run	Echelon run	Saved
DeepSeek V4.1 Flash	16 B300	4.504 s	2.944 s	2.50 h, $120	1.64 h, $79	0.87 h (34.6%)
Reflection Beam	43 RTX PRO	9.373 s	6.928 s	5.21 h, $672	3.85 h, $497	1.36 h (≈26.1%)
Mistral Small 4	10 RTX PRO	7.439 s	5.319 s	4.13 h, $124	2.96 h, $89	1.18 h (28.5%)
Nemotron 3.5 Lightning	3 RTX PRO	10.548 s	8.467 s	5.86 h, $53	4.70 h, $42	1.16 h (19.7%)
Qwen3.8-27B	3 RTX PRO	44.503 s	41.647 s	24.72 h, $223	23.14 h, $208	1.59 h (6.4%)

Table 18. Projected share of the training step saved, $f(1 - 1/S)$, for optimizer shares f and the measured optimizer speedups S at each width. Projections, not measurements. (Details: Appendix A.5.)

Share f	$S = 1.4\times$ ($d = 768$)	$3.8\times$ ($d = 2048$)	$5.9\times$ ($d = 4096$)	$10.5\times$ ($d = 8192$)
10%	3%	7%	8%	9%
25%	7%	18%	21%	23%
50%	14%	37%	42%	45%

For DeepSeek V4.1 Flash, 2,000 steps on 16 B300s take an estimated 1.6 h ($79) with Echelon and 2.5 h ($120) with Muon, at the same illustrative $3 per GPU-hour for both GPU types.

6.11 Savings projections and the cost calculator

Table 18 applies the same formula to other optimizer shares and to the speedups measured at each width. Wider layers raise both numbers, and mixture-of-experts models raise f further, since the optimizer updates every expert while each token uses only a few (Section 5.3). These cells are projections, not measurements.

The speedups in the column headers are whole-model optimizer benchmarks of the batched implementation at rank 64 on one RTX PRO 6000, against the faster of per-layer and batched Muon; on a B300 they are smaller (Table 9). Inside the 1.3B training run, at rank 128, the measured speedup was 2.5×.

The blog post that accompanies this report has a calculator that applies (7) to a GPU-hour budget, a price, a share f and a speedup S. The calculator opens on DeepSeek V4.1 Flash on 16 B300s with an illustrative 100,000 GPU-hour budget, which saves 34,516 GPU-hours, or $103,549. That figure is an estimate from timed parts, and it holds only if Echelon's loss stays level with Muon's. It did at 1.3B. At smaller widths Echelon trails Muon slightly, and closing that gap would eat into the saving. (Inputs: $f = 47.3\%$, $S = 3.7\times$, $3 per GPU-hour; the run then takes about 65,500 instead of 100,000 GPU-hours.) The 35% behind it carries the assumptions of Section 6.3.

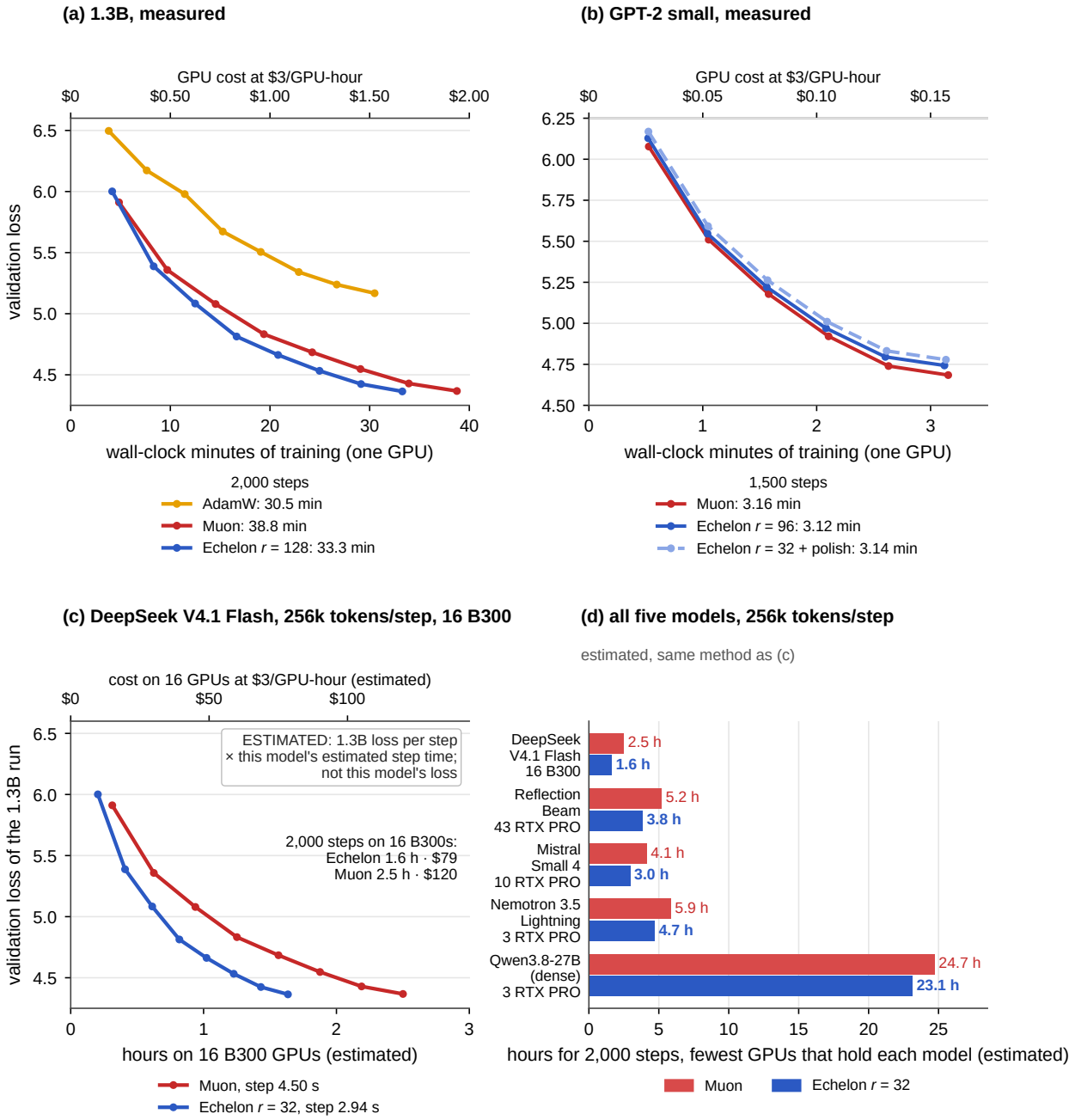


Figure 17. The same runs against wall-clock time. (a, b) Measured: 1.3B and GPT-2 small on one GPU, minutes of training (bottom) and GPU cost at $3 per GPU-hour (top). (c) Estimated: DeepSeek V4.1 Flash, the 1.3B run's loss per step × that model's estimated step time at 256k tokens per step on 16 B300s (Section 6.3), hours and cost; not that model's loss. (d) Estimated end times for all five models, each on the fewest GPUs that hold it, same method. (Details: Appendix A.25.)

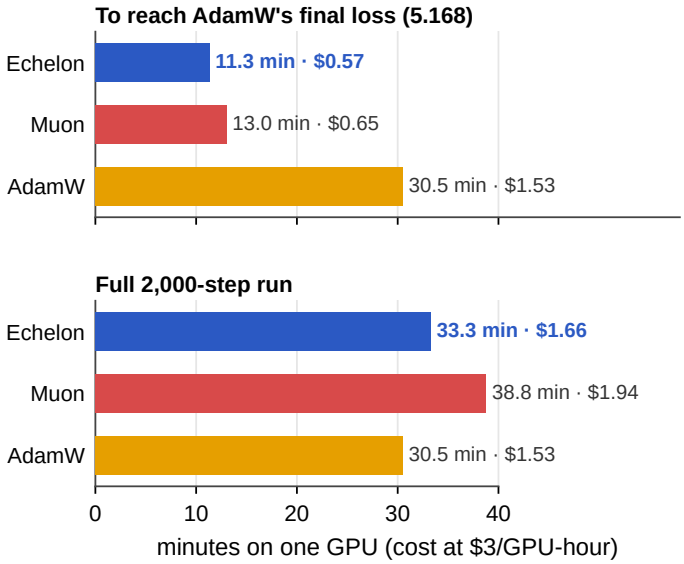


Figure 18. Time and cost at 1.3B. Top: time to reach the loss AdamW ends with. Bottom: the full 2,000-step run, one GPU, cost at $3 per GPU-hour. (Details: Appendix A.26.)

7 What the theory misses

Our convergence theorem rewards one number, the score: alignment divided by spectral norm. By that number, a rank-16 Echelon polished twice (0.81) is better than Muon (0.75). In training it is clearly worse: 1.593 against 1.548 on the character model.

What tracks training loss is how flat the update's spectrum is.

Measure the update's stable rank, $\|O\|_F^2/\|O\|_2^2$, divided by width. Muon's is 0.48. Echelon at rank 64 and twice-polished rank 16 are both 0.28, and plain rank 16 is 0.12. That is the same order as their losses. Figure 7 shows the mechanism: Muon's polynomial lifts even the weakest directions into its band near one, while Echelon preserves the tail's decay unless you polish it.

Worst-case rate theory cannot see this. A direction with a tiny singular value adds almost nothing to $\langle M, O\rangle$, so raising it spends spectral-norm budget and buys no certified progress. Training rewards it anyway, which suggests the small directions of the gradient carry signal that the per-step bound treats as noise. The Lean pilot pins down the one-step version: it refutes the claim that the score alone sets one-step progress and shows the update's stable rank decides it. Turning that into a rate is open. To us this is the most useful thing the project found, and it points at a concrete design: a method that costs $O(mnr)$ and flattens the tail the way Newton–Schulz does, instead of preserving its shape.

The one-step statement is `decrease_eq_kappa_srank` (Section 4.5): on a quadratic, the best decrease along a direction d is $\kappa^2\|G\|_*^2/(2\,\mathrm{srank}(d))$, so two directions with the same score κ can make different progress (the counterexample in `kappa_does_not_determine_decrease` uses $f = \frac{1}{2}\|W\|_F^2$, $G = \mathrm{diag}(1,0)$, and $d_1 = \mathrm{diag}(1,0)$, $d_2 = I$, both with score 1 and best decreases $1/2$ and $1/4$).

8 What we tried that didn't work

Not every variant helped. These are the ones worth knowing about, with the numbers that ruled them out. The first two items are from the 4-layer character model (width 256, 1,000 steps, seed 0), where Muon ends at 1.548.

- *Dropping the tail.* Keeping only the head's polar factor, at rank 16, ended at 1.788, against 1.629 for Echelon at the same rank and 1.629 for AdamW. The tail carries signal, as the alignment formula above says it should (equation (6)).
- *Cheaper updates with no head at all.* Dividing the momentum by its spectral norm, which is Echelon at rank 0, reached 1.729. Alternating row and column normalization, a SinkGD-style update, reached 1.846. Dion at rank 16 reached 1.715 (1.781 at twice the learning rate).
- *A higher-order Echelon (HO-4).* It keeps Echelon's tracked head, gives it an exact limiter, and runs a polar iteration in Gram space on the deflated tail. On the character model it beat Muon on all six seeds, by 0.005 on average. On GPT-2-small it lost on all three seeds, by 0.0435 ± 0.0054, and its fp32 small-matrix iteration made the optimizer 2.3× slower than bf16 Muon (31.3 against 13.8 ms per step).

Engineering failures that shaped the implementation:

- *One-shot subspace step.* Factoring $XX^\top U$ with Cholesky-QR in one go squares the singular values twice; in fp32 the Gram matrix lost positive definiteness and the per-matrix GPU version crashed within the first minute of training. Splitting the power step into two orthonormalized half-steps fixed it (Section 3.7).
- *Per-matrix calls.* One call per matrix hit a launch-latency floor of about 1.6 ms per matrix on an L40S; at GPT-2-small scale per-layer Echelon at rank 96 took 135.0 ms per optimizer step against Muon's 33.6 ms (full step 349.3 against 251.6 ms). Only batching made it faster than Muon.
- *Muon at twice the learning rate* (0.04) diverged on GPT-2 small (final validation loss 6.441).
- *Polishing at scale.* Echelon at rank 32 with one polishing step costs 9.8 s at 70B-class shapes against 3.1 s without it, because the cubic step needs full-size products (Table 12).

9 Limitations and measurement notes

This section collects, word for word, every limitation, caveat and measurement note from the pre-release draft of the companion blog post [12], and organizes them by topic. These passages were moved into this report when the published post was shortened. In this section "the post" means that draft, and quoted passages are set as they appeared in it ("this post" inside a quote is the post). In this version the passages that gave model-shape numbers in the post's 8-GPU accounting are reworded with the numbers of Section 6.3, each model on the fewest GPUs that hold it; those passages are marked. Each subsection adds what the report's own sections establish.

9.1 The model-shape savings: four caveats

The savings table (Table 8) mixes two kinds of rows: two trained runs and five models built from timed parts. The post stated (numbers re-based in version 2):

> The pattern is the one the cost model predicts: the dense 27B model saves the least, and the more experts a model has per active parameter, the more it saves. The lower rows are a different kind of measurement from the top two, so read them with these limits in mind:

- They are timings, not training runs. For each model we timed every matrix shape Muon updates on one RTX PRO 6000 Blackwell, Muon against Echelon's fused kernel at rank 32, and timed one real MoE or MLP layer's forward and backward pass. Then we combined the parts into a training step on the fewest GPUs that hold each model; DeepSeek V4.1 Flash was also timed on a B300, and its step is the B300's. We measured no loss on these models, and rank 32 is lower than the rank 128 that matched Muon's loss at 1.3B.
- A real step would save less. The model leaves out expert-parallel and sharding communication, the attention core and activation recomputation, which all make a real step longer. It also leaves out Muon's own optimizer communication, which would favour Echelon.
- Batch size matters. 256k tokens per step is typical of a post-training update. At 1M tokens per step the savings fall to 1.7%, 6.0%, 10.9%, 13.0% and 15.5%, in the table's order; at 128k they rise to 11.9%, 31.7%, 39.6%, 31.3% and 40.1%.
- The kernels differ. The two training runs used our earlier, unfused kernel. With that kernel the big-model speedups on the RTX PRO 6000 would be 3.6× to 6.5×; for DeepSeek V4.1 Flash on 47 RTX PRO 6000s it saves about 21.5% of the step where the fused kernel saves about 26.6% (Figure 15). On the B300 the unfused kernel is the faster of the two on the expert matrices.

9.2 Limits, in one place

The post's "Limits" section:

> Most of these are stated where they apply. Here they are in one place.
>
> - One seed, short runs. The 1.3B result is one seed of 2,000 steps at 32,768 tokens per step, about 65M tokens [version 2: three seeds per optimizer, Section 9.3]. The GPT-2-small comparison has three seeds per optimizer, and most character-model sweeps have one.
> - One GPU type. Every timing in this post comes from one NVIDIA RTX PRO 6000 Blackwell, except the first per-layer implementation and the GPT-2-small AdamW baseline, which ran on an L40S.
> - No MoE, RL or multi-node training. Echelon has trained only the dense models above, each on one GPU. The sharded version has been checked against the single-process one (Using Echelon) but has not trained a model across GPUs or nodes.
> - Proofs and code differ in the momentum. The Lean proofs cover heavy-ball (exponential-average) momentum; the code uses Nesterov momentum. The proofs also assume an exact head polar factor and an exact τ, which the code only approximates.
> - The gap at small width. At GPT-2-small width Echelon at rank 96 is worse than Muon on all three seeds, by 0.065 on average (4.7624 ± 0.0027 against 4.6975 ± 0.0024). One 1.3B seed is our evidence that it closes with width.
> - Big-model numbers are timed parts. The savings for the five large models combine optimizer timings with one timed layer's forward and backward pass. They are not training runs, and they leave out communication, the attention core and recomputation.
> - The fused kernel has not trained anything. Every loss in this post comes from the first implementation.
> - AdamW at 1.3B is untuned. It ran at one learning rate, 6e-4, so read the 2.7× against it as an upper bound.

9.3 Scope of evidence: seeds and run length

The 1.3B comparison (Section 6.9) was one seed per optimizer in the post and in version 1 of this report; this version adds seeds 1 and 2. In the post's words: "This is one seed and a short run, about 0.2% of a compute-optimal token budget for this size, so it is early evidence rather than proof." Figure 18's note adds: "These runs are short; a compute-optimal run of this model is roughly 400 times longer, and if the per-step times held, every bar and dollar figure would scale by that factor." The 1.3B final losses differ by 0.0003 (4.3883 against 4.3880) on a 50-batch evaluation and by −0.0033 on the last 20-batch curve point; with one seed each this is a tie, not a win, and we report it as a match within seed noise rather than as a share of Muon's gain. Over three seeds the picture is the same: 4.3751 ± 0.0115 for Echelon against 4.3757 ± 0.0109 for Muon, a paired difference of −0.0006 ± 0.0017, still a match within seed noise. Three seeds of 65M tokens are still a short run, and AdamW has one seed. The GPT-2-small Muon and rank-96 rows have three seeds each; the polished row, the AdamW rows and most character-model sweeps have one ("The polished row is compared with Muon's seed 0 (4.706)."). The negative results of Section 8 on the character model are seed 0 only, except where seeds are stated.

Optimizer benchmarks report the median of 5 timed runs after 2 warm-ups (Figures 1, 13, 14) or of 3–5 timed runs (model shapes); "the spread was not recorded". They use random inputs (`torch.randn`), not real gradients: "Random inputs; no loss was measured."

9.4 Timings versus training runs, and communication

The five large-model results (Tables 7 and 8, Figures 2, 11, 12, 15, and the estimated panels of Figure 17) are timed parts combined into a step on the fewest GPUs that hold each model, not training runs. The post stated, under the real-shapes table:

> The limits listed under the savings table apply here too: these are timings, not training runs, and a real multi-GPU step would save less. The MoE layer was timed with routing forced uniform, and for DeepSeek and Beam with a quarter to a third of the experts, scaled up. We have not trained an MoE with Echelon yet.

and, in the notes: "these are timings, not training runs, and a real multi-GPU step would save less"; "Routing is uniform, and communication, the attention core and recompute are left out; those make real steps longer, so real savings are lower." On the B300, DeepSeek's layer was timed with all 384 experts. The step divides the optimizer work evenly over its GPUs and does not model the communication any sharded optimizer needs; neither Muon's (which would favour Echelon) nor expert-parallel or FSDP traffic (which lengthens the step for both and so lowers the saving). The 4M-token column is not a single timed batch: each GPU processes its tokens as micro-batches of the largest measured size, and the timings are combined; the same holds for any column above the largest measured tokens per GPU, and columns below the smallest extend the measured line (both marked ≈). Cross-node behaviour is unmeasured. The GPU counts are the fewest that hold each model's training state; a real run would usually use more, and Table 9 shows that more GPUs at the same global batch save less.

The estimated wall-clock curves (Section 6.10) add one more assumption. They pair the 1.3B run's loss at each step with that model's estimated step time at 256k tokens per step on the fewest GPUs that hold it (Figures 2 and 11), in hours and cost on those GPUs: they show time and cost, not that model's loss, and assume Echelon keeps tracking Muon step for step at that scale.

9.5 Batch-size dependence

Every large-model saving depends on the tokens per optimizer step, because the optimizer's work is fixed while the forward and backward pass grows with the batch (Section 6.5). The headline 35% (34.6% on 16 B300s) is at 256k tokens per step; DeepSeek V4.1 Flash saves 40.1% at 128k, about 15.5% at 1M and about 4.6% at 4M. The post's guidance on which runs benefit included (numbers re-based in version 2):

> Little to gain: dense models at large batches (Qwen3.8-27B: 6.4% at 256k, 1.7% at 1M), and any model at 4M tokens per step or more (4.6% at most). Pretraining at tens of millions of tokens per step would save less still.

9.6 Kernel differences

Two implementations appear in this report. Every training run (every loss) used the first, unfused batched implementation (`ht_batched`); every large-model saving used the fused kernel. The post: "It has not been used in a training run yet: every loss in this post comes from the first implementation." With the first implementation the large-model speedups on the RTX PRO 6000 are 3.6–6.5× (Table 13); on the B300 it is faster than the fused kernel on DeepSeek's expert matrices, and Table 9 uses whichever is faster per shape (Section 9.1). The fused kernel matches the first implementation to within about 1%, and its largest observed spectral norm on expert matrices was 1.02; on the ill-conditioned bf16 stress test the first implementation reached $\|O\|_2 = 1.47$, above $\sqrt{2}$ (Section 6.7). Muon in all comparisons is Newton–Schulz in bf16, per layer or batched as stated; a kernel at 8 bytes per element is not built. The configurations also differ: the training runs used two tail iterations and no cubic step on the head, while the fused-kernel benchmarks (Table 13) run the guarded configuration, four tail iterations and one cubic step on the head, for both the first version and the fused kernel, and the same-day re-timing (Tables 9 and 11, Figure 15) runs four tail iterations for both implementations.

9.7 Ranks

Speedups depend on the rank, and the report uses several: rank 64 for the whole-model width benchmarks (Figure 1, Table 12); rank 32 for the large models and the fused kernel, with ranks up to 512 in Tables 9 and 11; rank 96 at GPT-2 small; rank 128 at 1.3B. "We measured no loss on these models, and rank 32 is lower than the rank 128 that matched Muon's loss at 1.3B." At rank 128, against batched Muon, Echelon is 1.15×, 3.3×, 5.6× and 10.2× faster at the four widths, below the rank-64 numbers. The post's note under the projected table: "The speedups in the column headers are whole-model optimizer benchmarks at rank 64, against the faster of per-layer and batched Muon. Inside the 1.3B training run, at rank 128, the measured speedup was 2.5×." The rank needed at width 1024 is not pinned down: at 1024 we measured only ranks 64 (score below Muon's) and 128 (above), so it lies between them (Table 15). For the large models, Tables 9 and 11 give ranks 32 to 512. At width/16 (rank 320) DeepSeek V4.1 Flash's optimizer step is 1.7× faster than Muon's on the B300 and 2.0× on the RTX PRO 6000, and the step saving on 16 B300s is 19.1%, against 34.6% at rank 32. For the other four models width/16 gives 2.0–4.6× and 5.3–16.1% on the RTX PRO 6000. We have measured no loss at these ranks on these models.

9.8 The small-width gap

Echelon trails Muon at small width. From the post: "At smaller scale it trails Muon slightly, and working out why was the most interesting thing we learned." and, for GPT-2 small, "It is worse than Muon on all three seeds, by 0.065 on average." On the character model rank 64 keeps about two thirds of Muon's gain over AdamW. The cost calculator's saving carries the same condition: "That figure is an estimate from timed parts, and it holds only if Echelon's loss stays level with Muon's. It did at 1.3B. At smaller widths Echelon trails Muon slightly, and closing that gap would eat into the saving." Section 7 gives our explanation, which is a finding, not a solved problem.

Table 19. The three trained comparisons behind the width trend. Sources: Sections 6.1, 6.8 and 6.9.

Run	Width	Layers	Rank	Rank/width	Steps	Seeds	Data	Hardware
Character model	256	4	64	1/4	1,000	3	TinyShakespeare	CPU, fp32
GPT-2 small	768	12	96	1/8	1,500	3	FineWeb	Blackwell
1.3B	2048	24	128	1/16	2,000	1	FineWeb	Blackwell

9.9 The width trend is confounded

The claim that Echelon's gap to Muon shrinks with width rests on three trained comparisons, and between them every other setting changes too (Table 19): depth, the rank as a fraction of the width, run length, data, hardware and the number of seeds. The trend may come from width, from rank or from length, or from none of them. The offline rank measurement of Table 15 holds depth and data fixed, but it measures the score of one step on saved momentum, not training loss. A controlled test would vary width alone at a fixed rank fraction, depth, token budget and seed count.

9.10 Baselines

AdamW ran at one learning rate at 1.3B and at the better of two at GPT-2 small (Section 9.12), so the comparisons against it are upper bounds. Every Muon comparison uses Muon's standard five-step quintic Newton–Schulz in bf16, per layer or batched, in eager PyTorch. We did not compare against cheaper Muon variants (fewer iterations, Gram-space or symmetric-product Newton–Schulz, fused Newton–Schulz kernels) or against implementations that shard the orthogonalization across GPUs; any of them would shrink Echelon's speedups. Dion was compared only on the 4-layer character model (Section 8). The learning rates of Muon and Echelon were not tuned beyond Muon's default of 0.02.

9.11 Scope of the proofs

What the Lean development covers is stated in Section 4; what it does not is the following.

- *Momentum.* The proofs cover exponential-average (heavy-ball) momentum; the code uses Nesterov momentum. In the post's related-work paragraph: Peng, Wang and Yu "cover Nesterov momentum, which we do not."
- *Head factor and τ.* The bounds assume a head factor with $\|P\| \le 1$ (the rate theorem also a given head alignment; an exact polar factor gives both) and $\tau \ge \|T\|_2$; neither has to be exact; the GPU code uses five Newton–Schulz steps on the head, which can overshoot to 1.25, and a power-iteration estimate of τ with a 1.05 margin. The post's numerics paragraph, in full: "In practice the bound has two conditions: the head's polar factor is exact, and τ is at least the tail's largest singular value. Our code bends both. Five Newton–Schulz steps on the head can overshoot to 1.25, and two power iterations can underestimate τ; the 1.05 margin helps but does not guarantee it. In a stress test with ill-conditioned matrices, bf16 products and rank half the width, the update's spectral norm reached 1.43 and 1.56. A cheap guard, one cubic step on the small head and four tail iterations instead of two, brought those to 1.33 and 1.36. On momentum saved from real training runs, the variant that refreshes the frame before using it, which is what we train with, stayed at or below 1.25; without that refresh it reached 1.52. Polishing the full update is the robust fix, but it costs full-size products, so we leave it off at scale." The quoted "exact" is stronger than needed: the theorems ask only for $\|P\| \le 1$ (Table 4). Measured inside two 1.3B training runs, neither hypothesis holds most of the time: $\tau < \|T\|_2$ in about 92% of updates, $\|P\|_2$ up to 1.20, and the update exceeds $\sqrt{2}$ in about 9% of updates, mostly in the first 50 steps (Section 3.5). The $\sqrt{2}$ and convergence theorems are therefore statements about an idealised Echelon, not guarantees for the code we trained with.

- *Tail loss.* "The rate theorem assumes the tail loss is small, which we measure rather than prove."
- *Noise model.* Worst-case bounded noise in the dual norm, not a stochastic model.
- *Real arithmetic.* The theorems are about exact real matrices; nothing is proved about bf16 or fp32 rounding.
- *Flops, not time.* "And Lean proves flop counts, not wall-clock times. Those we had to measure."
- *Spectral calculus.* The spiked-spectrum separation (Theorem 8) is proved for the scalar maps; the step from scalar maps to matrices is assumed, not formalized.
- *Pilot conditions.* The implicit-bias result models Muon as exact spectral steepest descent, with diminishing steps and no momentum.
- *Comparator sandbox.* The comparator normally runs its checks inside a bubblewrap sandbox; ours ran without it (`--inadvisably-no-sandbox`), because the sandbox was unavailable on the machine.

The post also pointed at this list from the introduction: "A later section covers what that buys and what it doesn't."

9.12 Other measurement notes

- *AdamW baselines.* "AdamW here runs on all parameters at learning rate 6e-4, which we did not tune at this size." "Because AdamW's learning rate was not tuned, read that 2.7× as an upper bound." At GPT-2 small the AdamW row is the better of two learning rates (1.5e-3 and 6e-4), one seed, on an L40S, and "The AdamW baseline is from the matching L40S run."
- *Projections.* The projected table (Table 18): "These cells are projections, not measurements."
- *RL.* "RL caveat: Echelon speeds up policy updates, not rollout generation, which is usually most of an RL run's GPU time. Against a whole RL run, the saving is the update share times the numbers above."
- *Sharding.* "So far it has run only on CPU." The sharded version matches single-process Echelon on 4 processes but has not trained a model across GPUs or nodes. A two-GPU data-parallel smoke test at GPT-2-small scale (500 steps) showed no optimizer speedup there (17.37 against 16.23 ms).
- *One GPU type and instance.* All Blackwell timings come from single-GPU instances; timings on other GPUs (H100, B200) and under multi-tenant load are not measured.
- *Times exclude evaluation and compilation.* Training-run step times are means over steps 10 to the end and exclude evaluation; wall-clock curves exclude compilation and evaluation.
- *Muon learning rate.* Muon at twice the learning rate diverged at GPT-2 small; we did not tune Muon's or Echelon's learning rate beyond Muon's default 0.02.
- *Tokens.* The 1.3B result is about 65M tokens; the GPT-2-small runs about 49M (1,500 × 32,768).

10 Related work

Muon and its geometry. Muon [13] orthogonalizes the momentum of hidden layers with a Newton–Schulz quintic; Liu et al. [15] show it scales to LLM training; Bernstein [3] develops the manifold view. Bernstein and Newhouse [4] interpret Muon and related methods as steepest descent under operator norms, and Pethick et al. [21] as linear-minimization oracles over norm balls, which is the view behind equation (1). Faster polynomial schedules for the polar factor are studied in The Polar Express [2]; classical background on matrix sign and polar iterations is in Higham [11].

Low-rank and clipped relatives. Echelon is a new combination of mostly published parts, and several close relatives appeared within weeks of this work. Dion [1] tracks a warm-started subspace but sends the tail to error feedback; Orth-Dion [16] revisits its geometry in the distributed low-rank setting. He, Deng and Lu [10] take the exact polar factor of a sketched head and discard the tail. Spectral clipping methods (Yukhimchuk et al. [26], MuCon [25], and the concurrent Spectrally Targeted Muon [9]) clip at a tuned threshold. The concurrent PMRC [24] adds the residual back scaled by a fixed constant, where Echelon divides it by its own spectral norm.

Convergence. Kim and Oh [14] analyze Muon with Newton–Schulz iterations. Peng, Wang and Yu [20] prove, on paper, a convergence theorem close to our certificate theorem and the same 1.25 bound, and they cover Nesterov momentum, which we do not.

What is new here. What we think is new: the rank-anchored, scale-free clip at $O(mnr)$ with the $\sqrt{2}$ guarantee under a stale frame, the polishing step, the spiked-spectrum separation, a batched GPU implementation that makes the saving real at small widths, the flatness observation (an open finding, not a solved problem), and machine-checked proofs of the bounds, the rate and the separation. As far as we can find, this is the first formal convergence proof for Muon or any steepest-descent method in the spectral norm.

Numerical building blocks. The frame is orthonormalized with CholeskyQR2 [6] with a diagonal shift [7]; the fused kernel is written in Triton [23]; the proofs use Lean 4 [5] and Mathlib [22], with statements pinned in the comparator format of OpenAI's math [17] and ten-proofs [18] releases.

Further reading. By topic: Muon: Jordan et al., *Muon: an optimizer for hidden layers* [13]; Liu et al., *Muon is scalable for LLM training* [15]; Bernstein, *Modular Manifolds* [3]. Geometry: Bernstein and Newhouse, *Old optimizer, new norm* [4]; Pethick et al., *Training deep learning models with norm-constrained LMOs* [21]. Low-rank and clipped relatives: Dion [1]; Orth-Dion [16]; He, Deng and Lu [10]; Yukhimchuk et al. [26]; MuCon [25]; Spectrally Targeted Muon [9]; PMRC [24]. Convergence: Kim and Oh, *Convergence of Muon with Newton–Schulz* [14]; Peng, Wang and Yu, *Convergence of practical Muon* [20]. Faster polar factors: Amsel et al., *The Polar Express* [2].

11 Directions for future work

- A theory that sees flatness. A rate that rewards lifting weak directions would explain Muon's edge and could be checked in the same Lean library.
- A cheap flat tail. Replace T/τ with an $O(mn)$ map that pushes tail singular values towards one.
- Sharded optimizers. How the $O(mnr)$ saving composes with distributed Muon and Dion-style communication.
- Scale. Multiple seeds and longer runs at 1B parameters and beyond, and a test that varies width alone.
- Stronger baselines. A tuned AdamW at 1.3B, and cheaper or fused Muon variants as the reference for every speedup.
- Rank. Large-model timings and quality at width/16.
- Nesterov momentum in Lean. Close the gap between what is proved and what the code runs.

Two further items follow directly from Section 9: training runs with the fused kernel, including an MoE run, and a measured multi-GPU, multi-node training step in place of the timed-parts model.

12 Conclusion and release

Echelon replaces Muon's cubic-cost orthogonalization with a rank-r head–tail update that is a spectral clip when its frame is exact and stays within spectral norm $\sqrt{2}$ when it is not. Its cost is $O(mnr)$, its stability and convergence are machine-checked, and on one RTX PRO 6000 its batched implementation turns the optimizer step for 70B-class layer shapes from 33.9 s into 3.23 s. In 1.3B runs (three seeds each) its loss was within seed noise of Muon's with a 14% shorter training step, and on the real shapes of large mixture-of-experts models the timings point to larger savings: for DeepSeek V4.1 Flash, timed on a B300, an estimated 35% per step on the 16 B300 GPUs that can hold the model and 31% on 24, at 256k tokens per step, rank 32 and without communication (Section 6.3). The optimizer speedup depends on the hardware: 10.7× on the RTX PRO 6000 and 3.7× on the B300 for the same model at rank 32. The remaining gap to Muon at small width points at spectral flatness, which the current theory does not capture.

On DeepSeek V4.1 Flash, our estimate from timed parts is about 35% off each training step on 16 B300s (about 31% on a more realistic 24). A frontier lab can absorb Muon's overhead. A university group, a startup or an open-source project often can't: for them, 100,000 versus about 65,500 GPU-hours can decide whether a run happens at all, or how many ablations fit in the budget. Echelon's update is a few thin matrix products whose cost grows with the rank, not the cube of the width, so the saving is largest on big mixture-of-experts models like DeepSeek V4.1 Flash.

Release. The Lean library, with its comparator challenges and the pilot result, is available at `https://github.com/evolving-machines-lab/echelon-lean` under Apache-2.0 (Appendix D). The code (the reference implementation, the batched implementation, the fused Triton kernel, the training and benchmark scripts and the result files named in Appendix A) will be released on GitHub soon, together with a package.

Acknowledgements

This research was done on compute provided through AWS credits for startups and Daytona credits. We thank AWS and Daytona for compute credits.

References

[1] Kwangjun Ahn, Byron Xu, Natalie Abreu, Ying Fan, Gagik Magakyan, Pratyusha Sharma, Zheng Zhan, and John Langford. Dion: Distributed Orthonormalized Updates. *arXiv preprint arXiv:2504.05295*, 2025.

[2] Noah Amsel, David Persson, Christopher Musco, and Robert M. Gower. The Polar Express: Optimal Matrix Sign Methods and Their Application to the Muon Algorithm. *arXiv preprint arXiv:2505.16932*, 2025.

[3] Jeremy Bernstein. Modular Manifolds. Thinking Machines Lab blog, 2025. URL https://thinkingmachines.ai/blog/modular-manifolds/.

[4] Jeremy Bernstein and Laker Newhouse. Old Optimizer, New Norm: An Anthology. *arXiv preprint arXiv:2409.20325*, 2024.

[5] Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021.

[6] Takeshi Fukaya, Yuji Nakatsukasa, Yuka Yanagisawa, and Yusaku Yamamoto. CholeskyQR2: A Simple and Communication-Avoiding Algorithm for Computing a Tall-Skinny QR Factorization on a Large-Scale Parallel System. In *2014 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, pages 31–38, 2014.

[7] Takeshi Fukaya, Ramaseshan Kannan, Yuji Nakatsukasa, Yusaku Yamamoto, and Yuka Yanagisawa. Shifted Cholesky QR for Computing the QR Factorization of Ill-Conditioned Matrices. *SIAM Journal on Scientific Computing*, 42(1):A477–A503, 2020.

[8] Gene H. Golub and Christian Reinsch. Singular Value Decomposition and Least Squares Solutions. *Numerische Mathematik*, 14:403–420, 1970.

[9] Vishrut Goyal and Rohan Ramkumar. Spectrally Targeted Muon. *arXiv preprint arXiv:2610.10965*, 2026.

[10] Chuan He, Zhanwang Deng, and Zhaosong Lu. Low-rank Orthogonalization for Large-scale Matrix Optimization with Applications to Foundation Model Training. *arXiv preprint arXiv:2509.11983*, 2025.

[11] Nicholas J. Higham. *Functions of Matrices: Theory and Computation*. SIAM, 2008.

[12] Tushar Jain. The Echelon Optimizer. Evolving Machines blog, 2026. URL https://evolvingmachines.ai/blog/echelon.

[13] Keller Jordan, Yuchen Jin, Vlado Boza, Jiacheng You, Franz Cesista, Laker Newhouse, and Jeremy Bernstein. Muon: An optimizer for hidden layers in neural networks. Blog post, 2024. URL https://kellerjordan.github.io/posts/muon/.

[14] Gyu Yeol Kim and Min-hwan Oh. Convergence of Muon with Newton-Schulz. *arXiv preprint arXiv:2601.19156*, 2026. Accepted at ICLR 2026.

[15] Jingyuan Liu, Jianlin Su, Xingcheng Yao, Zhejun Jiang, Guokun Lai, Yulun Du, Yidao Qin, Weixin Xu, Enzhe Lu, Junjie Yan, et al. Muon is Scalable for LLM Training. *arXiv preprint arXiv:2502.16982*, 2025.

[16] Tatsuhiro Nakamori, Laura Gomezjurado Gonzalez, Ganesh Talluri, Ansh Tiwari, Hideyuki Kawashima, Ioannis Mitliagkas, Guillaume Rabusseau, and Hiroki Naganuma. Orth-Dion: Eliminating Geometric Mismatch in Distributed Low-Rank Spectral Optimization. *arXiv preprint arXiv:2605.16341*, 2026.

[17] OpenAI. `openai/math`. GitHub repository, 2026. URL https://github.com/openai/math. Accessed November 2026.

[18] OpenAI. `openai/ten-proofs`. GitHub repository, 2026. URL https://github.com/openai/ten-proofs. Accessed November 2026.

[19] Guilherme Penedo, Hynek Kydlíček, Loubna Ben Allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro von Werra, and Thomas Wolf. The FineWeb Datasets: Decanting the Web for the Finest Text Data at Scale. *arXiv preprint arXiv:2406.17557*, 2024.

[20] Hanyng Peng, Hui Wang, and Yue Yu. Convergence of Practical Muon with Finite Newton-Schulz Iterations and Nesterov Momentum. *arXiv preprint arXiv:2609.39595*, 2026.

[21] Thomas Pethick, Wanyun Xie, Kimon Antonakopoulos, Zhenyu Zhu, Antonio Silveti-Falls, and Volkan Cevher. Training Deep Learning Models with Norm-Constrained LMOs. *arXiv preprint arXiv:2502.07529*, 2025.

[22] The mathlib Community. The Lean Mathematical Library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020)*, pages 367–381. ACM, 2020.

[23] Philippe Tillet, H. T. Kung, and David Cox. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*, pages 10–19, 2019.

[24] Sifan Yang, Hongyi Tao, Dingzhi Yu, Haomin Bai, Wei Jiang, Silin Gao, Pi Bu, and Lijun Zhang. Memory-Efficient Subspace Optimization with Residual Correction: From Momentum SGD to Muon. *arXiv preprint arXiv:2610.19844*, 2026.

[25] Albert Yi. MuCon: Clipped Muon Updates for LLM Training. *arXiv preprint arXiv:2605.26459*, 2026.

[26] Alexander Yukhimchuk, Mladen Kolar, Martin Takáč, and Sayantan Choudhury. Gradient Clipping Beyond Vector Norms: A Spectral Approach for Matrix-Valued Parameters. *arXiv preprint arXiv:2605.11838*, 2026.

A Sources and measurement details

The setup for each figure and table: hardware, inputs, repeats, seeds, rank, kernel and source file. Paths are relative to the code release. Entry A.*n* is note *n* of the "Notes on the figures and tables" in the pre-release draft of the companion blog post [12], kept word for word except the notes on the model-shape timings and the speed benchmark, which version 2 rewrites for its new accounting; "this post" there means that draft. The notes were moved here when the published post was shortened. Figure numbers in the notes are the draft's, which are this report's. The published post keeps the draft's Figures 1–12 under the same numbers, and shows this report's Figures 16, 17 and 18 as its Figures 13, 14 and 15. Several figures are interactive in the post; their notes describe the controls ("Drag", "Press"), and this report shows the default state as a static figure. Bracketed remarks are ours. Each figure and table caption in this report points to its entry.

A.1 Figure 1

Echelon's batched implementation (`py/batched.py`, the one every training run used) at rank 64 against Muon (per layer, bf16) over every hidden matrix of a model with the given layer shapes, one NVIDIA RTX PRO 6000 Blackwell (g7e.2xlarge), random inputs. The gain grows with width, from 2.0× at GPT-2-small shapes to 10.5× at 70B-class shapes (33.9 s down to 3.23 s per step); 3.8× at 2048 and 5.9× at 4096. At the smallest width Muon also gains from batching, which shrinks the gap there to 1.4×; from width 2048 up it does not. Random fp32 inputs (torch.randn, TF32 allowed), the four hidden shapes of each block (qkv $3d \times d$, proj $d \times d$, fc $4d \times d$, fc2 $d \times 4d$); orthogonalization only, without the momentum update. CUDA events, 2 warm-up calls, median of 5 timed runs; the spread was not recorded. The 8B- and 70B-class rows timed 8 and 2 layers and scaled by 4 and 40. Source: `results/gpu3/bench_batched.json`, rows "HT-Muon r=64, batched". [The 2.0× is against per-layer Muon and the 1.4× against batched Muon, both in this benchmark; inside the trained GPT-2-small run, at rank 96, the optimizer step was 1.1× faster (Table 8). Version 1 plotted the per-matrix variant (rows "HT-Muon r=64, per layer") at widths 4096 and 8192; Section 6.6 says why it was dropped. On a B300 the speedups are smaller (Table 9).]

A.2 Figure 2

Each pair of bars is one training step at 256k tokens per step, each model on the fewest GPUs that hold its training state (bf16 weights and gradients and fp32 momentum, 8 bytes per parameter; 288 GB per B300, 96 GB per RTX PRO 6000), scaled so Muon's step is 100%: grey is the forward and backward pass, colour is the optimizer (Muon above, Echelon below). Echelon shrinks only the coloured part, so the bigger the optimizer's share of the step, the more it saves: 34.6% for DeepSeek V4.1 Flash on 16 B300s, 6.4% for the dense Qwen3.8-27B on 3 RTX PRO 6000s. These are timings, not training runs, and a real multi-GPU step would save less (Section 9). DeepSeek V4.1 Flash: optimizer and forward and backward both timed on one B300 (all 384 experts allocated; forward and backward measured at 8,192, 10,923, 16,384 and 32,768 tokens per GPU); Echelon at rank 32 is the faster of the fused kernel and the batched implementation per shape. Other models: every matrix shape Muon would update was stacked and timed on one RTX PRO 6000 Blackwell (g7e.2xlarge, torch 2.14): Muon's Newton–Schulz in bf16 (`ns5_batched`) against Echelon's fused kernel (`kernels/echelon_fused.py`) at rank 32 from a warm state, medians of 3–5 timed runs after 2 warm-ups, on chunks of up to 3 GB and scaled to the full matrix count; forward and backward: one real MoE or MLP layer in bf16 with routing forced uniform (Beam with 81 of 256 experts, scaled up), times the layer count, plus the other parameters at the measured bf16 matmul rate, at 16k to 128k tokens per GPU. In both cases the whole-model optimizer time is divided by the GPU count, and the forward and backward time is taken at that count's tokens per GPU: measured, interpolated between measured points, extended below the smallest (≈) or accumulated above the largest (≈). No communication. Random inputs; no loss was measured. Sources: Re-timing of 2026-11-16 (`bench_rank.json` and `bench_shapes.json` for each GPU, written by `py/bench_shapes.py` and a rank sweep built on it): one NVIDIA B300 SXM6 (288 GB, torch 2.14.0, Triton 3.8.0) and one RTX PRO

6000 Blackwell, the same day. RTX PRO 6000 rows of the other models: `results/shapes/shapes.csv`, `results/shapes/REPORT.md`, `results/fused/bench_fused.json`. [Version 1 split every model over 8 GPUs; see Appendix C.]

A.3 Figure 3

Measured inside a 1.3B-parameter training run (width 2048, 24 layers, 2,000 steps, one seed). The optimizer step drops from 273 ms to 111 ms, but it is only part of the step, so the full training step drops from 1,163 ms to 999 ms: 1.16× faster, 14% less time and GPU cost. Validation loss: 4.3883 for Echelon, 4.3880 for Muon, 5.184 for AdamW. At GPT-2-small scale, where the optimizer is a much smaller share of the step, the end-to-end gain is only about 1%. The more of the step the optimizer takes, the more of Figure 1 shows up here; see where it fits best [Section 5]. One training run per optimizer, seed 0, on one RTX PRO 6000 Blackwell: a GPT-2-style model of width 2048 and 24 layers (1.42B parameters, untied head) on FineWeb, 2,000 steps of 32,768 tokens (16 sequences of 1,024, gradient accumulation 2), bf16 autocast, torch.compile. Hidden matrices on Muon or Echelon (rank 128, `py/batched.py`, two tail iterations, no polishing) at learning rate 0.02 with Nesterov momentum 0.95; AdamW at 6e-4 on everything else. Times are means over steps 10 to 2,000 and exclude evaluation. Loss: mean of 50 validation batches. Source: `results/gpu13/*.json`.

A.4 Savings table

Top two rows: the trained runs of the GPT-2-small and 1.3B tables below (`results/gpu3`, `results/gpu13`); f is the optimizer's time over the step time, and S is Muon's optimizer time over Echelon's. Lower five rows: the timed parts of Figure 2's note at 256k tokens per step, each model on the fewest GPUs that hold it.

A.5 Projected savings

Projections from $f(1 - 1/S)$, not measurements. The speedups are the whole-model, rank-64 times of Echelon's batched implementation in Figure 1's note against the faster of per-layer and batched Muon (`results/gpu3/bench_batched.json`).

A.6 Figure 4

Left: a 2×2 matrix M turns the unit circle into the large outlined ellipse, whose half-axes are σ_1 and σ_2. A candidate update O with the same singular directions turns it into the red ellipse, and $\|O\|_2 \le 1$ means that ellipse has to stay inside the dashed circle. Right: the same candidate as a point (o_1, o_2), the stretch it applies along each of M's singular directions. Here the spectral-norm ball is the square. Alignment is $\langle M, O \rangle = \sigma_1 o_1 + \sigma_2 o_2$, so it grows along the arrow labelled M, and the tinted part of the square is every update that beats yours. Drag O until nothing is tinted. You end at the corner $(1, 1)$, both stretches equal to one, which is polar(M), and the alignment there is $\sigma_1 + \sigma_2 = \|M\|_*$. Now make M lopsided: the corner does not move. The small grey dot is the best step under a Frobenius-norm budget, $M/\|M\|_F$; it follows the arrow and puts almost nothing on the weak direction. The square is exact, not a sketch: for any O with $\|O\|_2 \le 1$, the two numbers $o_i = u_i^\top O v_i$ lie in it, and the alignment depends on nothing else. Computed in your browser from the formulas in the caption: a 2×2 matrix and its singular value decomposition in closed form. No measurement. (In this report's static Figure 4 the candidate is drawn in purple and computed in Python.)

A.7 Figure 5

The red curve is one step, $p(x) = 3.4445x - 4.7750x^3 + 2.0315x^5$. Each dot is one singular value of a synthetic spectrum of eight values. Press Step: every dot moves up or down to the curve, then across to the diagonal, and that is its new value. Small values grow about 3.4 times per step until they reach the shaded band from 0.68 to 1.20. Inside the band they keep bouncing and never settle at 1. Muon first divides the matrix by its Frobenius norm, which puts every singular value in $(0, 1]$; the row under the axis shows that rescaling. With the box ticked, dragging the largest value changes nothing above the axis: normalization removes the scale. Untick the box and drag again. At exactly 1, a value that was already perfect is sent to 0.701 and sits at 0.70 after five steps. At 1.5 it blows up. Our Lean library proves that every normalized value of at least 0.6/243 ends in $[0.6, 1.25]$ after five steps. Computed in your browser: Muon's quintic with coefficients $(3.4445, -4.7750, 2.0315)$ applied to eight synthetic singular values in a geometric sequence (ratio 0.55). The Lean facts quoted are from the library in `FastMuon/` of the Lean repository. No measurement.

A.8 Figure 6

Data flows from top to bottom for a stack of same-shaped momenta. Steps 1 and 2 are one power step on the warm frame, split into two half-steps so that no Gram matrix squares the singular values twice. The head (blue) [the "Head" rows of Figure 6] gets Muon's treatment on a matrix only r rows tall; the tail is touched only through products with X, so T is never stored. Tags give the precision, the cost and the pass of the fused kernel that does the work. The training runs in this post used two tail iterations; the guard described below [Section 3.5] adds two more and one cubic step on the head. The data flow of `ht_batched` in `py/batched.py`, the implementation behind every loss in this post, and of `kernels/echelon_fused.py`, which does the same math in the 6 passes A–F. Precision tags are the reference's: bf16 for the two half-step products, fp32 for Cholesky-QR (two passes, diagonal shift 1e-6 of the mean diagonal), the head, the tail and the output. In the fused kernel's bf16 mode the head, tail and output read one bf16 copy of X with fp32 accumulation. Flop count: the cost model in `py/ortho.py`; bytes per element: `results/fused/REPORT.md`, section 1.

A.9 Figure 7

A synthetic width-4096 momentum with $\sigma_i \propto i^{-p}$, optionally with a boosted top direction. Left: the output singular value of each direction. The r directions left of the blue line are the head and all go to one, as in Muon's ideal. The rest are the tail: they keep their shape and are divided by σ_{r+1}, so the largest of them is exactly one. Right: the same thing as a function of the input singular value. Echelon's map is $\min(1, \sigma/\sigma_{r+1})$, a straight clip whose corner follows the rank you set. Muon's map is five Newton–Schulz steps after dividing by $\|M\|_F$: it lifts anything above a fixed fraction of $\|M\|_F$ into its band and does not know r. Try "spiked, rank 1", one direction at 1 and 4,095 at $4 \cdot 10^{-4}$, the example in our Lean library. Muon's normalization leaves the bulk far below one and its score is 0.55, while rank-1 Echelon scores exactly 1. On a smooth power law the order flips. Exact frame and exact τ are assumed here; Figure 8 drops the first and Figure 9 the second. Computed in your browser: a synthetic spectrum of width 4096 with $\sigma_i \propto i^{-p}$. Echelon's curve assumes an exact frame and an exact τ; Muon's is five Newton–Schulz steps after dividing by the Frobenius norm. No measurement. (This report's static Figure 7 shows the default $p = 0.40$, $r = 64$, where Muon scores 0.71 with flatness 0.48 and Echelon 0.40 with flatness 0.12.)

A.10 Figure 8

A synthetic 24×24 momentum with $\sigma_i \propto i^{-p}$ and a rank-r frame Q. Left: the angle of the blue ray is θ, how far Q has turned from the true top-r subspace toward a random set of directions orthogonal to it (every principal angle equals θ). Along the ray, the blue curve gives $\|O\|_2$ and the thinner neutral curve gives the alignment. Right: one dot for each singular direction x of the update $O = QP + T/\tau$. Its horizontal position is $\|QPx\|$, the head part, which lies inside $\mathrm{span}(Q)$. Its vertical position is $\|Tx\|/\tau$, the tail part, which lies outside. Each is at most one, so every dot stays in the unit square, and since the two parts are perpendicular a dot's distance from the origin is that direction's singular value. No point of the square is further than $\sqrt{2}$ from the origin. With an exact frame the dots sit on the two axes and the norm is one. Turn the frame and some head directions pick up tail as well: they drift toward the corner and the norm climbs. With the starting settings it is 1.37 at 25° and peaks at 1.39, short of $\sqrt{2} = 1.414$. Alignment rises slightly at first, because a longer update collects more of it, then falls from 0.77 to 0.51 by 90°. Alignment per unit of norm, the score our theorem uses, falls from the first degree: 0.77 with an exact frame, 0.40 at 75°. Make the spectrum steeper and the norm sits almost on the ceiling while the raw alignment grows with it; the score still falls. The head polar factor and $\tau = \|T\|_2$ are exact here. Computed in your browser: a synthetic 24×24 momentum with $\sigma_i \propto i^{-p}$, a frame turned so that every principal angle equals θ, and a Jacobi eigensolver for the singular values. The head polar factor and τ are exact. No measurement. (Defaults: $r = 4$, $p = 0.70$; the static figure reproduces these numbers, with the peak at 46°.)

A.11 Figure 9

The blue curve is the polish step acting on one singular value, $x \mapsto 1.5x - 0.5x^3$. The dots on the horizontal axis are the singular values of a synthetic Echelon update before polishing: the head at 1 and a decaying tail whose top value is the handle. Follow each dot to the curve to read its value after. Drag the handle to the right to make τ too small by that factor. The top of the tail now sits above 1 and the curve folds it back down. Everything in $[0, \sqrt{3}]$ lands in $[0, 1]$, the shaded box. With a wrong frame the largest singular value can be as large as $\sqrt{1 + c^2}$, the open circle, where c is the tail norm divided by τ. For c up to $\sqrt{3}$ that is at most 2, and the curve stays between -1 and 1 up to there, which is the bound our Lean library proves. Two things to notice. Past $\sqrt{3}$ a value comes out with its sign flipped, and past 2 the bound is lost. And on $(0, 1)$ the curve lies above the dashed diagonal, so polishing also lifts the tail. Computed in your browser: a synthetic update with its head at 1 and a 12-value tail decaying as $i^{-0.7}$. The bound quoted is the Lean theorem polished_ht_norm_le_one. No measurement. (The static figure uses the widget's default handle, $c = 1.30$.)

A.12 Figure 10

Top: one training step with Muon and the same step with Echelon. If Muon's optimizer takes a fraction f of the step and Echelon's is S times faster, the step gets shorter by $f(1 - 1/S)$. However large S is, the saving cannot pass f. Move f and S, or pick a measured run. Bottom: the measured saving for each run. GPT-2 small and 1.3B dense are real training runs on one GPU at 32k tokens per step. The five large models are not end-to-end training: every Muon-eligible matrix shape of each model was timed on one RTX PRO 6000 Blackwell (Muon `ns5_batched` in bf16 against the fused Echelon kernel at $r = 32$; DeepSeek V4.1 Flash also on one B300), one real MoE or MLP layer was timed forward and backward, and the two were combined into a step on the fewest GPUs that hold each model (DeepSeek V4.1 Flash timed on a B300). Routing is uniform, and communication, the attention core and recompute are left out; those make real steps longer, so real savings are lower. The optimizer's cost per step is fixed while forward and backward grow with the batch, so every line falls as tokens per step go from 128k to 1M. The MoE models keep the largest share because the optimizer updates every expert while each token pays for only a few. Top: the formula $f(1 - 1/S)$. Bottom: the measured savings of the savings table: the two trained runs (`results/gpu3`, `results/gpu13`) and the timed parts of Figure 2's note from 128k to 1M tokens per step.

A.13 Real model shapes table

The timed parts of Figure 2's note, each model on the fewest GPUs that hold it. Tokens per GPU are the tokens per step divided by the GPU count. Forward and backward was measured at 8k–32k tokens per GPU on the B300 and 16k–128k on the RTX PRO 6000; columns outside those ranges are marked ≈ (extended below, or micro-batches of the largest measured size above).

A.14 Figure 11

One training step per GPU, split into the forward and backward pass (grey) and the optimizer (colour), for Muon and for Echelon at rank 32, each model on the fewest GPUs that hold it. Each pair is drawn to the scale of Muon's step, with seconds per GPU on the right. The optimizer's work stays the same while the forward and backward pass grows with the tokens, so the coloured part, and with it the saving, shrinks from 128k to 1M tokens per step. Setup of Figure 2's note.

A.15 Figure 12

Share of a step saved by Echelon (rank 32) against Muon, each model on the fewest GPUs that hold it (DeepSeek V4.1 Flash on 16 B300s, the others on RTX PRO 6000s), built from timed parts. The optimizer's work is the same at every batch size while the forward and backward pass grows with it, so the saving falls roughly as one over the tokens per step. Dashed: the dense model. Hollow points: forward and backward computed outside the measured tokens per GPU. Setup of Figure 2's note, at 128k, 256k, 512k, 1M and 4M tokens per step.

A.16 Theorems table

Lean 4 v4.35.0 with Mathlib at the matching tag, lake build with no sorry; #print axioms on all 35 theorems lists only propext, Classical.choice and Quot.sound. Comparator: 7 pinned statements, run on 2026-11-08 (`COMPARATOR.md` in the Lean repository, `results/comparator2.log`). Pilot: `pilot/items/*/REPORT.md`.

A.17 Figure 13

Left: flops for one optimizer step from the cost model, for an $s \times l$ matrix with $s \le l$. Muon's five Newton–Schulz steps cost $5(4s^2l + 2s^3)$ and Echelon costs $6slr + 6(s + l)r^2 + 20r^3 + 10sl$. On these log scales Muon's line has slope 3 and, at a fixed rank, Echelon's has slope 2, so the gap widens in proportion to the width. Move the rank up and Echelon's line rises until, near $r = s$, it costs about as much as Muon. Right: the ratio. The line is what the flop count predicts. The blue dots are what we measured for a whole model with these block shapes on one NVIDIA RTX PRO 6000 Blackwell, random inputs, median of five runs: the faster of per-layer and batched Muon in bf16, divided by Echelon at rank 32, 64 or 128 in its faster mode (per layer or batched; per layer was timed at rank 64 only). The measured gain is one to two orders of magnitude below the flop ratio, because on a GPU the step is bound by memory traffic and kernel launches, not by arithmetic. Two presets reproduce Lean theorems: at least 1000× for a 4096×4096 layer at rank 16 (the exact ratio is 1,151×) and at least 400× for 4096×16384 at rank 32 (452×). A transformer block here is the four hidden matrices qkv ($3d \times d$), proj ($d \times d$), fc ($4d \times d$) and fc2 ($d \times 4d$). Left, and the line on the right: the flop formulas in the caption [the left-panel description above], computed in your browser. Dots: the whole-model times of Figure 1's note (`results/gpu3/bench_batched.json`), the faster of per-layer and batched Muon in bf16 over the faster of per-layer and batched Echelon at rank 32, 64 or 128 (per layer was timed at rank 64 only); random inputs, median of 5 runs. (The static figure shows rank 64 with Echelon's batched implementation, where the dots are 1.4×, 3.8×, 5.9× and 10.5×.) Panel titles in the post: "Flops per optimizer step, from the cost model. Log scales." and "How many times cheaper Echelon is. Flop count against a real GPU. Log scales." Interactive controls in the post: width d (default 8192) and rank r (default 64); "What is being orthogonalized": one $d \times d$, one $d \times 4d$, a transformer block; presets "Lean: 4096×4096, $r = 16$", "Lean: 4096×16384, $r = 32$" and "timed: block, $r = 64$".

A.18 Figure 14

Optimizer-step time for every hidden matrix of a model with the given layer shapes, one NVIDIA RTX PRO 6000 (Blackwell). Muon per layer in bf16; Echelon's batched implementation at rank 64, batched over same-shape matrices at every width. The 8B- and 70B-class rows time 8 and 2 layers and scale to 32 and 80. Random inputs, median of five runs. Log scale. The gap grows with width, as the flop count predicts. Against batched Muon, Echelon at the larger rank 128 is 1.15×, 3.3×, 5.6× and 10.2× faster. The setup of Figure 1's note: one RTX PRO 6000 Blackwell, random inputs, median of 5 runs after 2 warm-ups; Muon per layer in bf16, Echelon batched at rank 64. Source: `results/gpu3/bench_batched.json`.

A.19 Speed table

Same benchmark as Figure 14 (`results/gpu3/bench_batched.json`); the Echelon column is the batched implementation at rank 64 and the last column batched Echelon at rank 32 with one polishing step. (The GPT-2-small Muon time is 14.7457 ms in that file; this report rounds it to 14.7 ms.)

A.20 Fused-kernel table

One spot g7e.2xlarge (RTX PRO 6000 Blackwell Server Edition), torch 2.14.0, Triton 3.8.0. Every Muon-eligible matrix of each model at rank 32 from a warm state, in chunks of 64 same-shape matrices, scaled to the full count; whole model on one GPU. "First version" is `ht_batched` (`py/batched.py`). The ill-conditioned stress test uses rank half the width with bf16 products. Sources: `results/fused/REPORT.md`, `results/fused/bench_fused.json`.

A.21 Figure 15

Each rung is the same training step of DeepSeek V4.1 Flash at 256k tokens per step on the 47 RTX PRO 6000s that hold it, with Echelon at rank 32, as a share of Muon's step saved; the step time per GPU is on the right. The first version (the batched implementation, about 78 bytes per element) saves about 21.5%, the fused kernel (20 bytes per element) about 26.6%. An optimizer that cost nothing would save about 29.3%, because the forward and backward pass stays. The forward and backward time at 5,577 tokens per GPU extends the line through the measured 16k and 32k points. Rungs: Muon (Newton–Schulz, bf16): 0% (baseline), step 8.35 s; first version: ≈21.5%, step 6.56 s; fused kernel: ≈26.6%, step 6.13 s; ceiling: ≈29.3%, step 5.90 s. Source: Re-timing of 2026-11-16 (`bench_rank.json` and `bench_shapes.json` for each GPU, written by `py/bench_shapes.py` and a rank sweep built on it): one NVIDIA B300 SXM6 (288 GB, torch 2.14.0, Triton 3.8.0) and one RTX PRO 6000 Blackwell, the same day. The bytes per element are from `results/fused/REPORT.md`, section 1.

A.22 GPT-2-small table

Width 768, 12 layers, 163M parameters, FineWeb, 1,500 steps of 32,768 tokens, bf16 autocast. Muon and Echelon at rank 96: seeds 0, 1 and 2 on one spot g7e.2xlarge (RTX PRO 6000 Blackwell), batched bf16 path, learning rate 0.02, AdamW 1e-3 on the other parameters (`results/LEADERBOARD.md`, rows HG0 and HG1). Echelon at rank 32 with one polishing step: seed 0 (`results/gpu3`). AdamW on all parameters: seed 0 on an L40S, learning rate 1.5e-3, the better of 1.5e-3 and 6e-4 (`results/gpu`). Loss: mean of 50 validation batches.

A.23 1.3B table

The runs of Figure 3's note (`results/gpu13/*.json`): one seed, one RTX PRO 6000 Blackwell. Tokens per second is 32,768 divided by the step time.

A.24 Figure 16

Validation loss during training, from step 250, one seed each. Left: at 1.3B, Echelon (rank 128) tracks Muon from step 750 on and ends level; both finish far ahead of an untuned AdamW. Right: at GPT-2-small (batched, Blackwell), Echelon trails Muon by a margin that opens early and then holds. Curve points use a 20-batch evaluation, the tables a 50-batch final evaluation. Validation loss every 250 steps, each the mean of 20 batches (`data/curves_requested.csv` in the data bundle, from the run files `results/gpu13` and `results/gpu3`). One seed per curve.

A.25 Figure 17

The same runs against wall-clock time (bottom axis, minutes of training) and GPU cost (top axis, at the price above). Time is steps multiplied by each run's mean step time on one RTX PRO 6000, excluding evaluation. At GPT-2-small the optimizer is a small share of the step, so the three curves end within seconds of each other. The estimated tabs pair the 1.3B run's loss at each step with that model's estimated step time at 256k tokens per step on the fewest GPUs that hold it (Figures 2 and 11), in hours and cost on those GPUs: they show time and cost, not that model's loss, and assume Echelon keeps tracking Muon step for step at that scale. Time is the step number times the run's mean step time, without compilation or evaluation; cost is that time times the price you set. Runs: Figure 3's note (1.3B) and `results/gpu3` (GPT-2-small, batched, RTX PRO 6000 Blackwell). Data: `data/curves_wallclock_cost.csv`. (In this report the price is fixed at $3 per GPU-hour, and the estimated "tabs" are panels (c) and (d).)

A.26 Figure 18

Top: time to reach the loss AdamW ends with (5.168 on the same 20-batch evaluation), found by interpolating between evaluations every 250 steps. Echelon gets there about 2.7× sooner than AdamW and 13% sooner than Muon. Bottom: the full 2,000-step run, where Echelon saves 14% of Muon's time and cost and ends at the same loss. AdamW's learning rate was not tuned. These runs are short; a compute-optimal run of this model is roughly 400 times longer, and if the per-step times held, every bar and dollar figure would scale by that factor. Time to AdamW's final curve point (5.168, `results/gpu13`), interpolated linearly between the evaluations every 250 steps on the 20-batch curves. GPU-hours are 2,000 times the mean step time.

B Reproducibility

Paths are relative to the code release (Section 12), which will be published on GitHub soon; the layout below is the one the result files and notes refer to. Lean paths are relative to the root of the public `echelon-lean` repository, https://github.com/evolving-machines-lab/echelon-lean (Appendix D).

Code.

- `py/ortho.py`: reference oracles (Muon's `ns5`, the head–tail oracle with exact small polar factor, `cubic_polish`) and the flop model `flops`, mirrored in `FastMuon/Cost.lean` of the Lean repository.
- `py/batched.py`: the batched GPU implementation used by every training run on Blackwell (`ns5_batched`, `cholqr_batched`, `ht_batched`).
- `kernels/echelon_fused.py`: the fused Triton kernel (passes A–F) with the same contract as `ht_batched`; `kernels/test_fused.py` (correctness), `kernels/bench_fused.py` (GPU benchmark), `kernels/bytes_model.py` (bytes per element).

- py/gpt_gpu.py: GPU training (GPT-2 small and 1.3B; MuonBatched and per-layer MuonFamily optimizers); py/train_gpt.py: CPU character model; py/gpt_ddp.py: the two-GPU smoke test.
- py/bench_batched.py: whole-model optimizer benchmark (Figures 1, 13, 14); py/bench_shapes.py: real model shapes and timed forward/backward (Figures 2, 10–12, 15); py/cert_offline.py: scores on saved momentum (Table 15).
- The Lean library is the separate echelon-lean repository: FastMuon/*.lean, Axioms.lean, Pilot/O4_Certificate.lean, the comparator challenges (ComparatorChallenges/) and COMPARATOR.md at its root; pilot/items/*/ in the code release: the pilot's statements and reports.

Commands. Training (1.3B; GPT-2 small uses the script defaults --d 768 --layers 12 --heads 12 --batch 32 --accum 1 --steps 1500 --adam_lr 1e-3):

```
python py/gpt_gpu.py --opt htb --rank 128 --batched 1 --d 2048 --layers 24 --heads 16 \
  --batch 16 --accum 2 --steps 2000 --warmup 100 --lr 0.02 --adam_lr 6e-4 --seed 0
python py/gpt_gpu.py --opt ns5 --batched 1 ...          # Muon, same flags
python py/gpt_gpu.py --opt adamw --adam_lr 6e-4 ...     # AdamW on all parameters
```

Benchmarks and checks:

```
python py/bench_batched.py              # Figs 1, 13, 14 -> results/gpu3/
python py/bench_shapes.py out.json      # model shapes   -> results/shapes/
python kernels/bench_fused.py out.json  # fused kernel   -> results/fused/
TRITON_INTERPRET=1 python kernels/test_fused.py --quick   # kernel correctness on CPU
cd echelon-lean && lake exe cache get && lake build   # the Lean repository: proofs
lake env lean Axioms.lean                              # axioms of all 35 theorems
lake comparator --config ComparatorChallenges/<Name>.json # each of the 7 pinned statements
```

Result and data files.

- results/gpu13/ns5_r0_lr0.02_alr6e-4_b1.json, results/gpu13/htb_r128_lr0.02_alr6e-4_b1.json, results/gpu13/adamw_r0_lr0.02_alr6e-4_b0.json: the 1.3B runs (arguments, opt_ms, step_ms, final_val, validation curve).
- results/gpu3/: GPT-2 small on Blackwell and bench_batched.json; results/gpu/: GPT-2 small on L40S (AdamW baselines, per-layer runs); results/LEADERBOARD.md: the three-seed GPT-2-small and character-model rows.
- results/shapes/shapes.csv and REPORT.md: every large-model timing (columns fwdbwd_ms, muon_ms_per_gpu, fused32_ms_per_gpu, step_muon_ms, saved_pct_r32, saved_pct_fused, saved_pct_ceiling).
- results/fused/REPORT.md and bench_fused.json: the fused kernel (bytes per element, speed, correctness, the bf16 stress test).
- results/cert_offline*.txt: scores on saved momentum; results/comparator2.log: comparator run.
- Data bundle data/: curves_requested.csv (validation curves), curves_wallclock_cost.csv (loss against time and cost), bench_batched_blackwell.csv, cert_offline.csv, flop_model.csv, runs.csv (all run arguments).

C Changes from version 1

- The 1.3B result is reported as a match within seed noise, not as a share of Muon's gain, now over three seeds per optimizer (Table 16).
- Section 3.5 reports how often the $\sqrt{2}$ theorem's hypotheses fail inside training (about 92% of updates) and how often the bound is exceeded (about 9%), and Table 3 tests larger tail-estimate settings.
- Every model-shape estimate now puts each model on the fewest GPUs that hold its training state, and states what that step assumes (Section 6.3). Version 1 split the optimizer work of every model over 8 GPUs, which cannot hold the larger models for training, and gave DeepSeek V4.1 Flash an estimated 44.5% shorter step at 256k tokens per step in that accounting. DeepSeek V4.1 Flash is now timed on a B300 (optimizer and forward and backward, all 384 experts): about 35% on 16 B300s and 31% on 24 (Tables 7, 8 and 10, Figures 2, 11, 12, 15 and 17).
- New: the whole-model DeepSeek V4.1 Flash optimizer step on a B300 and an RTX PRO 6000 at ranks 32 to 512 (Table 9), the other four models at ranks 32, 64 and width/16 (Table 11), and the dependence of the speedup on the hardware: Muon runs 3.4× faster on the B300 while Echelon is bound by memory traffic, so the speedup at rank 32 is 3.7× there against 10.7× on the RTX PRO 6000. Every headline timing carries its rank.
- The optimizer-speed headline is the batched implementation that the training runs use: 10.5× (33.9 s to 3.23 s) at 70B-class shapes, rank 64, instead of version 1's 13.2× (2.56 s) from a per-matrix variant that no training run uses (Section 6.6). Figure 1 and Tables 12 and 18 use the batched implementation at every width (5.9× at 8B-class shapes instead of 6.4×).
- Appendix D prints the complete Lean sources.
- The width-768 speedups (2.0×, 1.4×, 1.15×, 1.1×) are each labelled by what they measure.
- One notation throughout (Table 2): S for the speedup, n_{tok} for tokens per step, K for the stack size, $\bar{M}$ for the momentum buffer, and τ for the margin-scaled tail estimate.
- Section 3.5 sets each hypothesis of the $\sqrt{2}$ theorem against the implementation; the bounds need $\|P\| \le 1$, not an exact polar factor, and the abstract and introduction now state their conditions.
- The Lean library is public at `https://github.com/evolving-machines-lab/echelon-lean` (Apache-2.0); the code and a package are described as coming soon on GitHub, with no release date.
- The width trend is described as confounded (Section 9.9), and the baselines' limits are stated (Section 9.10).

D Lean sources

This appendix prints the complete Lean library behind Section 4: 21 files, 2,293 lines. It builds with Lean 4 (`leanprover/lean4:v4.35.0`) and Mathlib at tag `v4.35.0` (commit `56a005855ea1`); `lake-manifest.json` pins every dependency. To check it, with `elan` installed:

```
lake exe cache get        # prebuilt Mathlib
lake build                # the library and the pilot; fails on any error
lake env lean Axioms.lean # #print axioms for the 35 audited theorems
lake comparator --config ComparatorChallenges/<Name>.json   # each pinned statement
```

The build fails on any error, and a `sorry` shows up as a warning and in `#print axioms`, which for every audited theorem lists only `propext`, `Classical.choice` and `Quot.sound`. The comparator challenges restate the headline statements with `sorry` on purpose: `lake comparator` checks that the real theorems

prove exactly those statements (Section 4.3); they are not part of the default build. The same tree, with `lakefile.toml`, `lean-toolchain`, `lake-manifest.json` and a README, is public at https://github.com/evolving-machines-lab/echelon-lean, and a copy is in the `lean/` folder of this report's source archive. The library is licensed under Apache-2.0 (the report itself is CC BY 4.0); `FastMuon/MatrixFacts.lean` adapts two lemmas from OpenAI's Lean repository, also Apache-2.0, as its header says. A clean copy of this tree, without build artifacts, builds with the four commands above (`lake build` completes with no errors, and all 35 `#print axioms` lines list only the three standard axioms). Files, in order, with line counts:

- Library: `FastMuon.lean` (18), `FastMuon/Certificate.lean` (237), `FastMuon/Cost.lean` (81), `FastMuon/HeadTail.lean` (155), `FastMuon/MatrixFacts.lean` (126), `FastMuon/MuonVsSVD.lean` (136), `FastMuon/NSMatrix.lean` (142), `FastMuon/NSPoly.lean` (101), `FastMuon/OptimalTail.lean` (44), `FastMuon/Polish.lean` (122), `FastMuon/Separation.lean` (99), `FastMuon/Spectral.lean` (158).
- Pilot: `Pilot/O4_Certificate.lean` (482).
- Axioms: `Axioms.lean` (37).
- Comparator challenges: `ComparatorChallenges/Certificate.lean` (52), `ComparatorChallenges/Cost.lean` (27), `ComparatorChallenges/HeadTail.lean` (34), `ComparatorChallenges/MuonVsSVD.lean` (42), `ComparatorChallenges/NSMatrix.lean` (46), `ComparatorChallenges/Separation.lean` (27), `ComparatorChallenges/Spectral.lean` (127).

Library

`FastMuon.lean` (18 lines)

```
import FastMuon.Certificate
import FastMuon.HeadTail
import FastMuon.Cost
import FastMuon.NSPoly
import FastMuon.Spectral
import FastMuon.MuonVsSVD
import FastMuon.Separation
import FastMuon.OptimalTail
import FastMuon.MatrixFacts
import FastMuon.NSMatrix
import FastMuon.Polish

/-!
# FastMuon

Lean proofs for **Echelon**, a rank-`r` replacement for Muon's Newton–Schulz step (its update is a
head on a tracked subspace plus a scaled tail; identifiers keep the older `HeadTail`/`ht_`/`htDir` prefix).
-/
```

`FastMuon/Certificate.lean` (237 lines)

```
import Mathlib

/-!
# Certified steepest descent: when does a cheap orthogonaliser keep Muon's rate?

Muon updates a weight matrix with `W ← W - η · polar(M)`, where `polar(M)` is the steepest-descent
direction for the spectral norm.  We abstract that setting:

* `E` is the parameter space (e.g. `Matrix m n ℝ`),
* `ip` is the pairing between gradients and updates (the Frobenius inner product),
* `N` is the primal seminorm (the spectral norm), `D` the dual seminorm (the nuclear norm),
  related by Hölder's inequality `|ip x y| ≤ D x * N y`.

A *direction oracle* (exact polar factor, Newton–Schulz, or the cheap head–tail oracle) only has to
return a direction `d` with an `(α, β)`-*certificate*

  `ip m d ≥ α * D m`   and   `N d ≤ β`

for the matrix `m` it was given.  The exact polar factor has `α = β = 1`.  We prove that any certified
```

```
oracle keeps the nonconvex `O(1/√T)` stationarity rate of Muon, with the constants degraded only through
`α` and `β`, including heavy-ball momentum and bounded gradient noise.

Nothing about matrices is used here: the theorems hold for any pair of seminorms satisfying Hölder.
-/

namespace FastMuon

open Finset

/-- The geometry: a bilinear pairing and two seminorms related by Hölder's inequality. -/
structure Geometry (E : Type*) [AddCommGroup E] [Module ℝ E] where
  ip : E →ₗ[ℝ] E →ₗ[ℝ] ℝ
  N : Seminorm ℝ E
  D : Seminorm ℝ E
  holder : ∀ x y, |ip x y| ≤ D x * N y

variable {E : Type*} [AddCommGroup E] [Module ℝ E]

namespace Geometry

variable (G : Geometry E)

/-- An `L`-smooth objective in the geometry `(N, D)`: the descent-lemma upper bound and a
`D`-Lipschitz gradient.  (For a differentiable `f` both follow from the second one; we assume both.) -/
structure Smooth (f : E → ℝ) (grad : E → E) (L : ℝ) : Prop where
  L_nonneg : 0 ≤ L
  upper : ∀ x y, f y ≤ f x + G.ip (grad x) (y - x) + L / 2 * G.N (y - x) ^ 2
  lip : ∀ x y, G.D (grad y - grad x) ≤ L * G.N (y - x)

/-- **Perturbation lemma.** A direction certified for a proxy `m` (momentum) is still a descent
direction for the true gradient `g`, losing `(α + β) · D (g - m)`. -/
theorem certificate_transfer {α β : ℝ} (hα : 0 ≤ α) {g m d : E}
    (hcert : α * G.D m ≤ G.ip m d) (hN : G.N d ≤ β) :
    α * G.D g - (α + β) * G.D (g - m) ≤ G.ip g d := by
  have hsplit : G.ip g d = G.ip m d + G.ip (g - m) d := by
    have : g = m + (g - m) := by abel
    conv_lhs => rw [this]
    simp
  have h1 : -(G.D (g - m) * G.N d) ≤ G.ip (g - m) d := by
    have := G.holder (g - m) d
    linarith [neg_abs_le (G.ip (g - m) d)]
  have h2 : G.D (g - m) * G.N d ≤ G.D (g - m) * β :=
    mul_le_mul_of_nonneg_left hN (apply_nonneg _ _)
  have htri : G.D g ≤ G.D m + G.D (g - m) := by
    have := map_add_le_add G.D m (g - m)
    simpa using this
  have h3 : α * G.D g ≤ α * G.D m + α * G.D (g - m) := by
    have := mul_le_mul_of_nonneg_left htri hα
    linarith
  nlinarith [h1, h2, h3, hsplit, hcert]

/-- **One step.** If `d` is certified for the gradient up to an error `ε`, then the step
`x - η • d` decreases `f` by at least `η (α D(∇f) - ε) - L η² β² / 2`. -/
theorem one_step {f : E → ℝ} {grad : E → E} {L : ℝ} (hf : G.Smooth f grad L)
    {x d : E} {η α β ε : ℝ} (hη : 0 ≤ η)
    (hcert : α * G.D (grad x) - ε ≤ G.ip (grad x) d) (hN : G.N d ≤ β) :
    f (x - η • d) ≤ f x - η * (α * G.D (grad x) - ε) + L / 2 * η ^ 2 * β ^ 2 := by
  have h := hf.upper x (x - η • d)
  have hsub : x - η • d - x = -(η • d) := by abel
  rw [hsub] at h
  have hip : G.ip (grad x) (-(η • d)) = -(η * G.ip (grad x) d) := by simp
  have hN' : G.N (-(η • d)) = η * G.N d := by
    rw [map_neg_eq_map, map_smul_eq_mul, Real.norm_eq_abs, abs_of_nonneg hη]
  rw [hip, hN'] at h
  have hNd : 0 ≤ G.N d := apply_nonneg _ _
  have hsq : (η * G.N d) ^ 2 ≤ η ^ 2 * β ^ 2 := by
    rw [mul_pow]
    exact mul_le_mul_of_nonneg_left (pow_le_pow_left₀ hNd hN 2) (sq_nonneg η)
  have hL : L / 2 * (η * G.N d) ^ 2 ≤ L / 2 * (η ^ 2 * β ^ 2) :=
    mul_le_mul_of_nonneg_left hsq (by linarith [hf.L_nonneg])
  have hdesc : η * (α * G.D (grad x) - ε) ≤ η * G.ip (grad x) d :=
    mul_le_mul_of_nonneg_left hcert hη
  nlinarith [h, hL, hdesc]

/-- **Telescoped descent.** For iterates `x (t+1) = x t - η • d t` with per-step certificates
(`α`, `β`, errors `ε t`), the weighted sum of dual gradient norms is bounded. -/
theorem sum_bound {f : E → ℝ} {grad : E → E} {L : ℝ} (hf : G.Smooth f grad L)
    {x d : ℕ → E} {ε : ℕ → ℝ} {η α β : ℝ} (hη : 0 ≤ η)
    (hstep : ∀ t, x (t + 1) = x t - η • d t)
    (hcert : ∀ t, α * G.D (grad (x t)) - ε t ≤ G.ip (grad (x t)) (d t))
```

```
    (hN : ∀ t, G.N (d t) ≤ β) (T : ℕ) :
    η * α * ∑ t ∈ range T, G.D (grad (x t)) ≤
      f (x 0) - f (x T) + η * ∑ t ∈ range T, ε t + T * (L / 2 * η ^ 2 * β ^ 2) := by
  induction T with
  | zero => simp
  | succ T ih =>
    have hs := G.one_step hf hη (hcert T) (hN T)
    rw [← hstep T] at hs
    rw [sum_range_succ, sum_range_succ]
    push_cast
    nlinarith [hs, ih]

/-- **Rate.** With `f ≥ f_low`, the average dual gradient norm over `T` steps is at most
`(f x₀ - f_low)/(α η T) + L η β²/(2α) + (average error)/α`. -/
theorem avg_bound {f : E → ℝ} {grad : E → E} {L : ℝ} (hf : G.Smooth f grad L)
    {x d : ℕ → E} {ε : ℕ → ℝ} {η α β flow : ℝ} (hη : 0 < η) (hα : 0 < α)
    (hstep : ∀ t, x (t + 1) = x t - η • d t)
    (hcert : ∀ t, α * G.D (grad (x t)) - ε t ≤ G.ip (grad (x t)) (d t))
    (hN : ∀ t, G.N (d t) ≤ β) (hlow : ∀ y, flow ≤ f y) {T : ℕ} (hT : 0 < T) :
    (∑ t ∈ range T, G.D (grad (x t))) / T ≤
      (f (x 0) - flow) / (α * η * T) + L * η * β ^ 2 / (2 * α) +
        (∑ t ∈ range T, ε t) / (α * T) := by
  have hs := G.sum_bound hf hη.le hstep hcert hN T
  have hfT := hlow (x T)
  have hTpos : (0 : ℝ) < T := by exact_mod_cast hT
  have hden : 0 < α * η * T := by positivity
  rw [div_le_iff₀ hTpos]
  have key : η * α * ∑ t ∈ range T, G.D (grad (x t)) ≤
      (f (x 0) - flow) + η * ∑ t ∈ range T, ε t + T * (L / 2 * η ^ 2 * β ^ 2) := by
    linarith
  have e1 : (f (x 0) - flow) / (α * η * T) * T = (f (x 0) - flow) / (α * η) := by
    field_simp
  have e2 : L * η * β ^ 2 / (2 * α) * T = (T * (L / 2 * η ^ 2 * β ^ 2)) / (α * η) := by
    field_simp
  have e3 : (∑ t ∈ range T, ε t) / (α * T) * T = (η * ∑ t ∈ range T, ε t) / (α * η) := by
    field_simp
  rw [add_mul, add_mul, e1, e2, e3, ← add_div, ← add_div, le_div_iff₀ (by positivity)]
  linarith

/-- **Momentum error.** Heavy-ball EMA momentum `m (t+1) = μ m t + (1-μ) ĝ (t+1)` of noisy gradients
`ĝ t = g t + ξ t` with `D ξ ≤ σ`, where consecutive true gradients move by at most `δ` in `D`,
stays within `μ δ / (1-μ) + σ` of the true gradient. -/
theorem momentum_error {g ghat m : ℕ → E} {μ δ σ : ℝ} (hμ0 : 0 ≤ μ) (hμ1 : μ < 1) (hδ : 0 ≤ δ)
    (hm0 : m 0 = ghat 0) (hm : ∀ t, m (t + 1) = μ • m t + (1 - μ) • ghat (t + 1))
    (hnoise : ∀ t, G.D (ghat t - g t) ≤ σ) (hdrift : ∀ t, G.D (g t - g (t + 1)) ≤ δ) :
    ∀ t, G.D (m t - g t) ≤ μ * δ / (1 - μ) + σ := by
  have h1μ : 0 < 1 - μ := by linarith
  have hB : 0 ≤ μ * δ / (1 - μ) := div_nonneg (mul_nonneg hμ0 hδ) h1μ.le
  intro t
  induction t with
  | zero => rw [hm0]; linarith [hnoise 0]
  | succ t ih =>
    have hdecomp : m (t + 1) - g (t + 1) =
        μ • (m t - g t) + μ • (g t - g (t + 1)) + (1 - μ) • (ghat (t + 1) - g (t + 1)) := by
      rw [hm t]
      simp only [smul_sub, sub_smul, one_smul]
      abel
    rw [hdecomp]
    have tri := (map_add_le_add G.D (μ • (m t - g t) + μ • (g t - g (t + 1)))
      ((1 - μ) • (ghat (t + 1) - g (t + 1))))
    have tri2 := map_add_le_add G.D (μ • (m t - g t)) (μ • (g t - g (t + 1)))
    simp only [map_smul_eq_mul, Real.norm_eq_abs, abs_of_nonneg hμ0, abs_of_pos h1μ] at tri tri2
    have a1 : μ * G.D (m t - g t) ≤ μ * (μ * δ / (1 - μ) + σ) := mul_le_mul_of_nonneg_left ih hμ0
    have a2 : μ * G.D (g t - g (t + 1)) ≤ μ * δ := mul_le_mul_of_nonneg_left (hdrift t) hμ0
    have a3 : (1 - μ) * G.D (ghat (t + 1) - g (t + 1)) ≤ (1 - μ) * σ :=
      mul_le_mul_of_nonneg_left (hnoise (t + 1)) h1μ.le
    have alg : μ * (μ * δ / (1 - μ) + σ) + μ * δ + (1 - μ) * σ = μ * δ / (1 - μ) + σ := by
      field_simp
      ring
    linarith

/-- **Main theorem (Muon family with momentum).** Run `x (t+1) = x t - η • d t` where the oracle sees
only the momentum `m t` and returns a direction with an `(α, β)` certificate for `m t`. If gradients
are observed with noise at most `σ` (in `D`), then

`avg_t D(∇f(x t)) ≤ (f x₀ - f_low)/(α η T) + L η β²/(2α) + (α+β)/α · (μ L η β/(1-μ) + σ)`.

For the exact polar factor (`α = β = 1`) this is Muon's deterministic rate; a cheap oracle with a
certificate `(α, β)` pays only these constant factors. -/
theorem muon_family_rate {f : E → ℝ} {grad : E → E} {L : ℝ} (hf : G.Smooth f grad L)
    {x d m ghat : ℕ → E} {η α β μ σ flow : ℝ} (hη : 0 < η) (hα : 0 < α) (hβ : 0 ≤ β)
```

```
    (hμ0 : 0 ≤ μ) (hμ1 : μ < 1)
    (hstep : ∀ t, x (t + 1) = x t - η • d t)
    (horacle : ∀ t, α * G.D (m t) ≤ G.ip (m t) (d t)) (hN : ∀ t, G.N (d t) ≤ β)
    (hm0 : m 0 = ghat 0) (hm : ∀ t, m (t + 1) = μ • m t + (1 - μ) • ghat (t + 1))
    (hnoise : ∀ t, G.D (ghat t - grad (x t)) ≤ σ)
    (hlow : ∀ y, flow ≤ f y) {T : ℕ} (hT : 0 < T) :
    (∑ t ∈ range T, G.D (grad (x t))) / T ≤
      (f (x 0) - flow) / (α * η * T) + L * η * β ^ 2 / (2 * α) +
        (α + β) / α * (μ * (L * η * β) / (1 - μ) + σ) := by
  -- drift of true gradients: D(g t - g (t+1)) ≤ L N(x(t+1) - x t) = L η N(d t) ≤ L η β
  have hdrift : ∀ t, G.D (grad (x t) - grad (x (t + 1))) ≤ L * η * β := by
    intro t
    have h := hf.lip (x (t + 1)) (x t)
    have hx : x t - x (t + 1) = η • d t := by rw [hstep t]; abel
    rw [hx, map_smul_eq_mul, Real.norm_eq_abs, abs_of_pos hη] at h
    have : L * (η * G.N (d t)) ≤ L * (η * β) :=
      mul_le_mul_of_nonneg_left (mul_le_mul_of_nonneg_left (hN t) hη.le) hf.L_nonneg
    linarith
  have hδ : 0 ≤ L * η * β := mul_nonneg (mul_nonneg hf.L_nonneg hη.le) hβ
  have herr := G.momentum_error (g := fun t => grad (x t)) hμ0 hμ1 hδ hm0 hm hnoise hdrift
  set B := μ * (L * η * β) / (1 - μ) + σ with hBdef
  have hcert : ∀ t, α * G.D (grad (x t)) - (α + β) * B ≤ G.ip (grad (x t)) (d t) := by
    intro t
    have h := G.certificate_transfer hα.le (g := grad (x t)) (horacle t) (hN t)
    have he : G.D (grad (x t) - m t) ≤ B := by
      have := herr t
      rw [← map_neg_eq_map, neg_sub]
      simpa [hBdef, mul_assoc] using this
    have : (α + β) * G.D (grad (x t) - m t) ≤ (α + β) * B :=
      mul_le_mul_of_nonneg_left he (by linarith)
    linarith
  have h := G.avg_bound hf (ε := fun _ => (α + β) * B) hη hα hstep hcert hN hlow hT
  have hTpos : (0 : ℝ) < T := by exact_mod_cast hT
  have hsum : (∑ t ∈ range T, (fun _ => (α + β) * B) t) / (α * T) = (α + β) / α * B := by
    simp only [sum_const, card_range, nsmul_eq_mul]
    field_simp
  rw [hsum] at h
  exact h

end Geometry

/-! ## The hypotheses are satisfiable

Euclidean geometry (`N = D = ‖·‖`, Cauchy–Schwarz) is an instance; for matrices with the spectral and
nuclear norms, Hölder is von Neumann's trace inequality (standard; not re-proved here). -/

/-- Any real inner-product space gives a `Geometry` with `N = D = ‖·‖`. -/
noncomputable def euclideanGeometry (F : Type*) [NormedAddCommGroup F] [InnerProductSpace ℝ F] :
    Geometry F where
  ip := inner, F
  N := normSeminorm ℝ F
  D := normSeminorm ℝ F
  holder x y := by
    simpa using abs_real_inner_le_norm x y

end FastMuon
```

FastMuon/Cost.lean (81 lines)

```
import Mathlib

/-!
# Cost model: head–tail vs Newton–Schulz

Flop counts for one oracle call on an `s × l` matrix with `s ≤ l` (a multiply–add counts as 2 flops).
These are the formulas used in `py/ortho.py :: flops`.  The theorems below are exact statements about
these counting functions; whether wall-clock time follows the flop count is an empirical question
(see the benchmark in the write-up).

* Newton–Schulz with `k` quintic steps: each step forms `A = X Xᵀ` (`2 s² l`), `A²` (`2 s³`),
  and `B X` (`2 s² l`): `k (4 s² l + 2 s³)`.
* Head–tail with rank `r`: `H = Uᵀ X`, `U (P - H/τ)`, `X Hᵀ` (`6 s l r`), the small polar factor and the
  QR of the next frame (`6 (s + l) r² + 20 r³`, generous), and the tail-scale power iteration plus the
  elementwise work (`10 s l`).
-/

namespace FastMuon.Cost

/-- Flops of `k` Newton–Schulz quintic steps on an `s × l` matrix (`s ≤ l`). -/
def nsCost (s l k : ℕ) : ℕ := k * (4 * s ^ 2 * l + 2 * s ^ 3)
```

```

/-- Flops of one head-tail call with rank `r`. -/
def htCost (s l r : ℕ) : ℕ := 6 * s * l * r + 6 * (s + l) * r ^ 2 + 20 * r ^ 3 + 10 * s * l

/-- The head-tail cost is linear in each of `s`, `l`, `r` up to a constant: at most `48 s l r`. -/
theorem htCost_le (s l r : ℕ) (hr : 1 ≤ r) (hrs : r ≤ s) (hsl : s ≤ l) :
    htCost s l r ≤ 48 * s * l * r := by
  unfold htCost
  have h1 : (s + l) * r ^ 2 ≤ 2 * s * l * r := by
    have : (s + l) * r ≤ 2 * l * s := by nlinarith
    calc (s + l) * r ^ 2 = ((s + l) * r) * r := by ring
      _ ≤ (2 * l * s) * r := Nat.mul_le_mul_right _ this
      _ = 2 * s * l * r := by ring
  have h2 : r ^ 3 ≤ s * l * r := by
    have : r * r ≤ s * l := Nat.mul_le_mul hrs (hrs.trans hsl)
    calc r ^ 3 = (r * r) * r := by ring
      _ ≤ (s * l) * r := Nat.mul_le_mul_right _ this
  have h3 : s * l ≤ s * l * r := Nat.le_mul_of_pos_right _ hr
  nlinarith

/-- Five Newton-Schulz steps cost at least `20 s² l`. -/
theorem nsCost_ge (s l : ℕ) : 20 * s ^ 2 * l ≤ nsCost s l 5 := by
  unfold nsCost; nlinarith [Nat.zero_le (s ^ 3)]

/-- **Speed-up guarantee.** For `1 ≤ r ≤ s ≤ l`, Muon's 5-step Newton-Schulz costs at least
`(5/12)·(s/r)` times the head-tail oracle:  `5 s · htCost ≤ 12 r · nsCost₅`.
For `s = 4096, r = 16` this is a ≥ 106× flop reduction even with these loose constants;
the exact formulas give 1151× (see `speedup_4096_16`). -/
theorem speedup (s l r : ℕ) (hr : 1 ≤ r) (hrs : r ≤ s) (hsl : s ≤ l) :
    5 * s * htCost s l r ≤ 12 * r * nsCost s l 5 := by
  have h1 := htCost_le s l r hr hrs hsl
  have h2 := nsCost_ge s l
  calc 5 * s * htCost s l r ≤ 5 * s * (48 * s * l * r) := Nat.mul_le_mul_left _ h1
    _ = 12 * r * (20 * s ^ 2 * l) := by ring
    _ ≤ 12 * r * nsCost s l 5 := Nat.mul_le_mul_left _ h2

/-- Concrete instance: a 4096 × 4096 layer, rank 16 — more than 1000× fewer flops. -/
theorem speedup_4096_16 : 1000 * htCost 4096 4096 16 ≤ nsCost 4096 4096 5 := by
  unfold htCost nsCost; norm_num

/-- Concrete instance: a 4096 × 16384 MLP layer, rank 32 — more than 400× fewer flops (exact ratio ≈ 452×). -/
theorem speedup_4096_16384_32 : 400 * htCost 4096 16384 32 ≤ nsCost 4096 16384 5 := by
  unfold htCost nsCost; norm_num

/-- The speed-up grows without bound: for any target factor `K`, every layer with `s ≥ 3 K r`
(and `1 ≤ r ≤ s ≤ l`) gets at least a `K`-fold flop reduction. -/
theorem speedup_unbounded (K s l r : ℕ) (hr : 1 ≤ r) (hrs : r ≤ s) (hsl : s ≤ l)
    (hK : 3 * K * r ≤ s) : K * htCost s l r ≤ nsCost s l 5 := by
  have h := speedup s l r hr hrs hsl
  have hs : 0 < s := lt_of_lt_of_le hr hrs
  -- 5 s · K · ht ≤ K · 12 r · ns ≤ (12/15) s · ns  ⇒  K · ht ≤ ns (with room to spare)
  have : 5 * s * (K * htCost s l r) ≤ 5 * s * nsCost s l 5 := by
    calc 5 * s * (K * htCost s l r) = K * (5 * s * htCost s l r) := by ring
      _ ≤ K * (12 * r * nsCost s l 5) := Nat.mul_le_mul_left _ h
      _ = 4 * (3 * K * r) * nsCost s l 5 := by ring
      _ ≤ 4 * s * nsCost s l 5 := Nat.mul_le_mul_right _ (Nat.mul_le_mul_left _ hK)
      _ ≤ 5 * s * nsCost s l 5 := Nat.mul_le_mul_right _ (by omega)
  exact Nat.le_of_mul_le_mul_left this (by omega)

end FastMuon.Cost
```

FastMuon/HeadTail.lean (155 lines)

```
import Mathlib

/-!
# The head-tail direction is a certified oracle

For a momentum matrix `X : s × l`, the head-tail oracle keeps an orthonormal frame `U : s × r`
(`Uᵀ U = 1`, a warm-started estimate of the top-`r` left singular subspace) and returns

  `O = U P + τ⁻¹ • T`,   `T = X − U (Uᵀ X)`   (the tail, outside the span of `U`),

where `P` is the polar factor of the small `r × l` matrix `H = Uᵀ X` and `τ ≈ ‖T‖₂`.
Everything costs `O(s l r)` instead of Newton-Schulz's `O(s² l)`.

We prove, for the spectral norm `‖·‖` (Mathlib's `L2` operator norm) and the Frobenius pairing:

* `tail_perp`: `Uᵀ T = 0` (the tail is orthogonal to the head),
* `gram_split`: `Oᵀ O = Pᵀ P + τ⁻² Tᵀ T` — no cross terms, **whatever `U` is**, as long as it is orthonormal,
```

```
* `htDir_norm_sq_le` / `htDir_norm_le_sqrt_two`: `‖O‖^2 ≤ ‖P‖^2 + ‖T‖^2/τ^2`, hence `‖O‖ ≤ √2` when
  `‖P‖ ≤ 1` and `τ ≥ ‖T‖` (the `β` of the certificate),
* `head_isometry`: `‖U P‖ = ‖P‖`,
* `align`: `⟪X, O⟫ = ⟪Uᵀ X, P⟫ + ‖T‖_F^2 / τ`, so `O` is always a descent direction (`align_nonneg`).

The remaining certificate constant `α` (how much of the nuclear norm `⟪X, O⟫` captures) depends on the
spectrum of `X` and on how well `U` tracks the top-`r` subspace; we measure it, we do not prove it.
-/

set_option linter.unusedSectionVars false

namespace FastMuon.HeadTail

open Matrix
open scoped Matrix.Norms.L2Operator

variable {s l r : Type*} [Fintype s] [Fintype l] [Fintype r]
  [DecidableEq s] [DecidableEq l] [DecidableEq r]

/-- The tail of `X` outside the column span of `U`. -/
def tail (U : Matrix s r ℝ) (X : Matrix s l ℝ) : Matrix s l ℝ := X - U * (Uᵀ * X)

/-- The head–tail direction `U P + τ⁻¹ • tail U X`. -/
noncomputable def htDir (U : Matrix s r ℝ) (P : Matrix r l ℝ) (τ : ℝ) (X : Matrix s l ℝ) :
    Matrix s l ℝ :=
  U * P + τ⁻¹ • tail U X

/-- Frobenius pairing `⟪A, B⟫ = tr (Aᵀ B)`. -/
def frob (A B : Matrix s l ℝ) : ℝ := (Aᵀ * B).trace

theorem tail_perp {U : Matrix s r ℝ} (hU : Uᵀ * U = 1) (X : Matrix s l ℝ) :
    Uᵀ * tail U X = 0 := by
  simp [tail, Matrix.mul_sub, ← Matrix.mul_assoc, hU]

theorem tail_perp' {U : Matrix s r ℝ} (hU : Uᵀ * U = 1) (X : Matrix s l ℝ) :
    (tail U X)ᵀ * U = 0 := by
  have h := congrArg transpose (tail_perp hU X)
  simpa [transpose_mul] using h

/-- **No cross terms.** `Oᵀ O = Pᵀ P + τ⁻² Tᵀ T` for any orthonormal frame `U`. -/
theorem gram_split {U : Matrix s r ℝ} (hU : Uᵀ * U = 1) (P : Matrix r l ℝ) (τ : ℝ)
    (X : Matrix s l ℝ) :
    (htDir U P τ X)ᵀ * htDir U P τ X = Pᵀ * P + (τ⁻¹ * τ⁻¹) • ((tail U X)ᵀ * tail U X) := by
  set T := tail U X with hT
  have h1 : Uᵀ * T = 0 := tail_perp hU X
  have h2 : Tᵀ * U = 0 := tail_perp' hU X
  have expand : (htDir U P τ X)ᵀ * htDir U P τ X =
      Pᵀ * (Uᵀ * U) * P + τ⁻¹ • (Pᵀ * (Uᵀ * T)) + τ⁻¹ • (Tᵀ * U * P) +
        (τ⁻¹ * τ⁻¹) • (Tᵀ * T) := by
    simp only [htDir, ← hT, transpose_add, transpose_smul, transpose_mul, Matrix.add_mul,
      Matrix.mul_add, Matrix.smul_mul, Matrix.mul_smul, smul_add, smul_smul, Matrix.mul_assoc]
    abel
  rw [expand, hU, h1, h2]
  simp

/-- The head part is an isometric embedding of `P`: `‖U P‖ = ‖P‖`. -/
theorem head_isometry {U : Matrix s r ℝ} (hU : Uᵀ * U = 1) (P : Matrix r l ℝ) :
    ‖U * P‖ = ‖P‖ := by
  have h : ‖U * P‖ * ‖U * P‖ = ‖P‖ * ‖P‖ := by
    rw [← l2_opNorm_conjTranspose_mul_self, ← l2_opNorm_conjTranspose_mul_self,
      conjTranspose_eq_transpose_of_trivial, conjTranspose_eq_transpose_of_trivial,
      transpose_mul, Matrix.mul_assoc, ← Matrix.mul_assoc Uᵀ, hU, Matrix.one_mul]
  have h0 : 0 ≤ ‖U * P‖ := norm_nonneg _
  have h0' : 0 ≤ ‖P‖ := norm_nonneg _
  nlinarith [sq_nonneg (‖U * P‖ - ‖P‖), sq_nonneg (‖U * P‖ + ‖P‖)]

/-- **Spectral-norm bound (`β`).** `‖O‖^2 ≤ ‖P‖^2 + ‖T‖^2 / τ^2`. -/
theorem htDir_norm_sq_le {U : Matrix s r ℝ} (hU : Uᵀ * U = 1) (P : Matrix r l ℝ) (τ : ℝ)
    (X : Matrix s l ℝ) :
    ‖htDir U P τ X‖ * ‖htDir U P τ X‖ ≤
      ‖P‖ * ‖P‖ + (τ⁻¹ * τ⁻¹) * (‖tail U X‖ * ‖tail U X‖) := by
  rw [← l2_opNorm_conjTranspose_mul_self, conjTranspose_eq_transpose_of_trivial,
    gram_split hU P τ X]
  calc ‖Pᵀ * P + (τ⁻¹ * τ⁻¹) • ((tail U X)ᵀ * tail U X)‖
      ≤ ‖Pᵀ * P‖ + ‖(τ⁻¹ * τ⁻¹) • ((tail U X)ᵀ * tail U X)‖ := norm_add_le _ _
    _ = ‖P‖ * ‖P‖ + (τ⁻¹ * τ⁻¹) * (‖tail U X‖ * ‖tail U X‖) := by
      rw [norm_smul, ← conjTranspose_eq_transpose_of_trivial, l2_opNorm_conjTranspose_mul_self,
```

```
        ← conjTranspose_eq_transpose_of_trivial, l2_opNorm_conjTranspose_mul_self,
        Real.norm_eq_abs, abs_of_nonneg (mul_self_nonneg τ⁻¹)]

/-- **`β ≤ √2`.** If the head factor is a contraction (`‖P‖ ≤ 1`, true for an exact or clamped polar
factor) and the tail scale dominates the tail (`‖T‖ ≤ τ`), then `‖O‖ ≤ √2`. -/
theorem htDir_norm_le_sqrt_two {U : Matrix s r ℝ} (hU : Uᵀ * U = 1) {P : Matrix r l ℝ} {τ : ℝ}
    {X : Matrix s l ℝ} (hP : ‖P‖ ≤ 1) (hτ : 0 < τ) (hT : ‖tail U X‖ ≤ τ) :
    ‖htDir U P τ X‖ ≤ Real.sqrt 2 := by
  have h := htDir_norm_sq_le hU P τ X
  have hP2 : ‖P‖ * ‖P‖ ≤ 1 := by nlinarith [norm_nonneg P]
  have hT2 : (τ⁻¹ * τ⁻¹) * (‖tail U X‖ * ‖tail U X‖) ≤ 1 := by
    have hq : τ⁻¹ * ‖tail U X‖ ≤ 1 := by
      rw [inv_mul_le_iff₀ hτ]; linarith
    have hq0 : 0 ≤ τ⁻¹ * ‖tail U X‖ := mul_nonneg (inv_nonneg.mpr hτ.le) (norm_nonneg _)
    nlinarith
  have hsq : ‖htDir U P τ X‖ ^ 2 ≤ 2 := by nlinarith
  calc ‖htDir U P τ X‖ = Real.sqrt (‖htDir U P τ X‖ ^ 2) := by
        rw [Real.sqrt_sq (norm_nonneg _)]
    _ ≤ Real.sqrt 2 := Real.sqrt_le_sqrt hsq

/-- `⟪X, T⟫ = ‖T‖_F²`: the tail is the orthogonal projection of `X`. -/
theorem frob_tail {U : Matrix s r ℝ} (hU : Uᵀ * U = 1) (X : Matrix s l ℝ) :
    frob X (tail U X) = frob (tail U X) (tail U X) := by
  have h2 : (tail U X)ᵀ * U = 0 := tail_perp' hU X
  unfold frob
  congr 1
  conv_rhs => rw [show (tail U X)ᵀ = Xᵀ - (Uᵀ * X)ᵀ * Uᵀ by
    simp [tail, transpose_sub, transpose_mul]]
  rw [Matrix.sub_mul, Matrix.mul_assoc, tail_perp hU X, Matrix.mul_zero, sub_zero]

/-- **Alignment identity.** `⟪X, O⟫ = ⟪Uᵀ X, P⟫ + ‖T‖_F² / τ`. -/
theorem align {U : Matrix s r ℝ} (hU : Uᵀ * U = 1) (P : Matrix r l ℝ) (τ : ℝ)
    (X : Matrix s l ℝ) :
    frob X (htDir U P τ X) = frob (Uᵀ * X) P + τ⁻¹ * frob (tail U X) (tail U X) := by
  have hT := frob_tail hU X
  unfold frob at *
  simp only [htDir, Matrix.mul_add, Matrix.mul_smul, trace_add, trace_smul, smul_eq_mul]
  rw [hT, transpose_mul, transpose_transpose, Matrix.mul_assoc]

theorem frob_self_nonneg (A : Matrix s l ℝ) : 0 ≤ frob A A := by
  unfold frob
  simp only [trace, diag, mul_apply, transpose_apply]
  exact Finset.sum_nonneg fun j _ => Finset.sum_nonneg fun i _ => mul_self_nonneg (A i j)

/-- **Always a descent direction.** If the head factor is aligned with the head (`⟪H, P⟫ ≥ 0`, true for
any polar factor of `H`) and `τ > 0`, then `⟪X, O⟫ ≥ ‖T‖_F²/τ ≥ 0`. -/
theorem align_nonneg {U : Matrix s r ℝ} (hU : Uᵀ * U = 1) {P : Matrix r l ℝ} {τ : ℝ}
    {X : Matrix s l ℝ} (hP : 0 ≤ frob (Uᵀ * X) P) (hτ : 0 < τ) :
    τ⁻¹ * frob (tail U X) (tail U X) ≤ frob X (htDir U P τ X) ∧ 0 ≤ frob X (htDir U P τ X) := by
  rw [align hU P τ X]
  have h0 : 0 ≤ τ⁻¹ * frob (tail U X) (tail U X) :=
    mul_nonneg (inv_nonneg.mpr hτ.le) (frob_self_nonneg _)
  constructor <;> linarith

/-- Muon is the full-rank member of the family: with a square orthogonal frame the tail vanishes and
`O = U P`. -/
theorem full_rank_tail_zero {U : Matrix s s ℝ} (hU' : U * Uᵀ = 1) (X : Matrix s l ℝ) :
    tail U X = 0 := by
  simp [tail, ← Matrix.mul_assoc, hU']

end FastMuon.HeadTail
```

FastMuon/MatrixFacts.lean (126 lines)

```
import Mathlib

/-!
# Matrix facts for the matrix-level Newton–Schulz bound

All norms are Mathlib's spectral norm (`L2` operator norm, `open scoped Matrix.Norms.L2Operator`)
unless stated otherwise.

* `spectrum_mem_Icc_of_posSemidef`: the spectrum of a real PSD matrix lies in `[0, ‖A‖]`;
* `norm_aeval_le` (**polynomial norm bound for symmetric matrices**): for a real symmetric `A` and
  a real polynomial `p`, `‖p(A)‖ ≤ M` as soon as `|p| ≤ M` on the spectrum of `A`.  This goes through
  Mathlib's isometric continuous functional calculus on `Matrix n n ℝ`
  (`Matrix.instIsometricContinuousFunctionalCalculus`, `cfc_polynomial`, `norm_cfc_le`);
* `norm_aeval_le_of_posSemidef`: the same with the spectrum replaced by the interval `[0, ‖A‖]`;
```

```
* `psd_norm_le_trace`: `‖A‖ ≤ tr A` for real PSD `A`;
* `frobNorm`, `frobNorm_eq_frobenius_norm` (it is Mathlib's Frobenius norm) and
  `l2_opNorm_le_frobNorm`: `‖M‖₂ ≤ ‖M‖_F`.

## Provenance

`psd_norm_le_trace` is copied, with whitespace and naming adapted to Mathlib v4.35.0, from
`OAI/Probability/LogConcave/TensorGraphs/Vertex.lean` (namespace
`OAI.LogConcaveSampling.TraceMoment`, lemma `psd_norm_le_trace`) in OpenAI's Lean repository
(https://github.com/openai/math, directory `lean/`), which is licensed under the Apache License, Version 2.0
(http://www.apache.org/licenses/LICENSE-2.0).  Changes: the proof was re-indented and stated in this
file's namespace; no mathematical change.  Everything else in this file is original.
-/

set_option linter.unusedSectionVars false

namespace FastMuon.MatrixFacts

open Matrix Polynomial

section Frobenius

open scoped Matrix.Norms.Frobenius

variable {s l : Type*} [Fintype s] [Fintype l]

/-- The Frobenius norm `‖M‖_F = √(∑ᵢⱼ Mᵢⱼ²)`, written out explicitly. -/
noncomputable def frobNorm (M : Matrix s l ℝ) : ℝ := Real.sqrt (∑ i, ∑ j, M i j ^ 2)

/-- `frobNorm` is Mathlib's Frobenius norm (`open scoped Matrix.Norms.Frobenius`). -/
theorem frobNorm_eq_frobenius_norm (M : Matrix s l ℝ) : frobNorm M = ‖M‖ := by
  rw [frobenius_norm_def, frobNorm, Real.sqrt_eq_rpow]
  congr 1
  refine Finset.sum_congr rfl fun i _ => Finset.sum_congr rfl fun j _ => ?_
  rw [Real.norm_eq_abs, Real.rpow_two, sq_abs]

end Frobenius

open scoped Matrix.Norms.L2Operator

variable {n s l : Type*} [Fintype n] [DecidableEq n] [Fintype s] [Fintype l]
  [DecidableEq s] [DecidableEq l]

theorem norm_one_le : ‖(1 : Matrix n n ℝ)‖ ≤ 1 := by
  have h := l2_opNorm_conjTranspose_mul_self (1 : Matrix n n ℝ)
  rw [conjTranspose_one, Matrix.one_mul] at h
  nlinarith [norm_nonneg (1 : Matrix n n ℝ)]

/-- The spectrum of a real positive semidefinite matrix lies in `[0, ‖A‖]`. -/
theorem spectrum_mem_Icc_of_posSemidef {A : Matrix n n ℝ} (hA : A.PosSemidef) {x : ℝ}
    (hx : x ∈ spectrum ℝ A) : 0 ≤ x ∧ x ≤ ‖A‖ := by
  refine ⟨?_, ?_⟩
  · rw [hA.1.spectrum_real_eq_range_eigenvalues] at hx
    obtain ⟨i, rfl⟩ := hx
    exact hA.eigenvalues_nonneg i
  · have h := spectrum.norm_le_norm_mul_of_mem hx
    rw [Real.norm_eq_abs] at h
    calc x ≤ |x| := le_abs_self x
      _ ≤ ‖A‖ * ‖(1 : Matrix n n ℝ)‖ := h
      _ ≤ ‖A‖ * 1 := mul_le_mul_of_nonneg_left norm_one_le (norm_nonneg _)
      _ = ‖A‖ := mul_one _

/-- **Polynomial norm bound for real symmetric matrices.** If `|p(x)| ≤ M` for every `x` in the
spectrum of the symmetric matrix `A`, then `‖p(A)‖ ≤ M`. -/
theorem norm_aeval_le {A : Matrix n n ℝ} (hA : A.IsHermitian) (p : ℝ[X]) {M : ℝ} (hM : 0 ≤ M)
    (h : ∀ x ∈ spectrum ℝ A, |p.eval x| ≤ M) : ‖aeval A p‖ ≤ M := by
  rw [← cfc_polynomial p A hA.isSelfAdjoint]
  exact norm_cfc_le hM fun x hx => by simpa only [Real.norm_eq_abs] using h x hx

/-- **Polynomial norm bound for real PSD matrices.** If `|p(x)| ≤ M` on `[0, ‖A‖]`, then
`‖p(A)‖ ≤ M`. -/
theorem norm_aeval_le_of_posSemidef {A : Matrix n n ℝ} (hA : A.PosSemidef) (p : ℝ[X]) {M : ℝ}
    (hM : 0 ≤ M) (h : ∀ x : ℝ, 0 ≤ x → x ≤ ‖A‖ → |p.eval x| ≤ M) : ‖aeval A p‖ ≤ M :=
  norm_aeval_le hA.1 p hM fun x hx =>
    h x (spectrum_mem_Icc_of_posSemidef hA hx).1 (spectrum_mem_Icc_of_posSemidef hA hx).2

/-- `‖A‖ ≤ tr A` for a real PSD matrix.  Copied from OpenAI's Lean repository
(`OAI/Probability/LogConcave/TensorGraphs/Vertex.lean`, Apache-2.0); see the module docstring. -/
theorem psd_norm_le_trace {A : Matrix n n ℝ} (hA : A.PosSemidef) : ‖A‖ ≤ A.trace := by
  have hn : 0 ≤ ∑ i, hA.1.eigenvalues i := Finset.sum_nonneg (fun i _ => hA.eigenvalues_nonneg i)
  rw [hA.1.trace_eq_sum_eigenvalues]
```

```
  conv_lhs => rw [hA.1.spectral_theorem]
  simp only [Unitary.conjStarAlgAut_apply, ← Unitary.coe_star, CStarRing.norm_mul_coe_unitary,
    CStarRing.norm_coe_unitary_mul, l2_opNorm_diagonal]
  apply (pi_norm_le_iff_of_nonneg hn).mpr
  intro i
  simpa only [Function.comp_apply, RCLike.ofReal_real_eq_id, id_eq, Real.norm_eq_abs,
    abs_of_nonneg (hA.eigenvalues_nonneg i)] using
    (Finset.single_le_sum (fun j _ => hA.eigenvalues_nonneg j) (Finset.mem_univ i))

/-- `tr (Mᵀ M) = ∑ᵢⱼ Mᵢⱼ²`. -/
theorem trace_transpose_mul_self (M : Matrix s l ℝ) :
    (Mᵀ * M).trace = ∑ i, ∑ j, M i j ^ 2 := by
  simp only [trace, diag, mul_apply, transpose_apply]
  rw [Finset.sum_comm]
  simp [sq]

/-- **Spectral norm ≤ Frobenius norm**: `‖M‖₂ ≤ ‖M‖_F`. -/
theorem l2_opNorm_le_frobNorm (M : Matrix s l ℝ) : ‖M‖ ≤ frobNorm M := by
  have hpsd : (Mᵀ * M).PosSemidef := by
    simpa only [conjTranspose_eq_transpose_of_trivial] using posSemidef_conjTranspose_mul_self M
  have h1 : ‖M‖ * ‖M‖ ≤ ∑ i, ∑ j, M i j ^ 2 := by
    rw [← l2_opNorm_conjTranspose_mul_self, conjTranspose_eq_transpose_of_trivial,
      ← trace_transpose_mul_self]
    exact psd_norm_le_trace hpsd
  rw [frobNorm]
  calc ‖M‖ = Real.sqrt (‖M‖ * ‖M‖) := (Real.sqrt_mul_self (norm_nonneg _)).symm
    _ ≤ Real.sqrt (∑ i, ∑ j, M i j ^ 2) := Real.sqrt_le_sqrt h1

theorem frobNorm_nonneg (M : Matrix s l ℝ) : 0 ≤ frobNorm M := Real.sqrt_nonneg _

end FastMuon.MatrixFacts
```

`FastMuon/MuonVsSVD.lean` (136 lines)

```
import FastMuon.NSPoly
import FastMuon.Cost

/-!
# Checking three common claims about Muon's Newton–Schulz step

A popular write-up states three theorems about Muon's quintic `p(σ) = 3.4445σ − 4.7750σ³ + 2.0315σ⁵`
(left as `sorry` there).  Lean says:

1. "one step contracts the error `|p σ − 1| < |σ − 1|` on `[0.5, 1.5]`" — **false** (`claimed_contraction_false`):
   at `σ = 1` the error goes from `0` to `0.299`.
2. "five steps land within `0.05` of `1` on `[0.5, 1.5]`" — **false**, twice over:
   at `σ = 1.5` the iteration diverges (`claimed_five_step_false`), and even at `σ = 1`
   five steps end in `[0.65, 0.90]` (`five_steps_from_one`).  Muon's quintic is tuned to be fast, not
   accurate: the true guarantee is the band `[0.6, 1.25]` of `NSPoly.muon_five_steps`, after normalising
   by the Frobenius norm.
3. "five NS steps need fewer flops than SVD once `m, n ≥ 4096`" (with the write-up's own cost formulas) —
   **false** (`claimed_efficiency_false`, e.g. `m = 8192, n = 4096`).  With a correct NS count, five NS steps
   need **2.5×–5×** more flops** than a Golub–Reinsch SVD (`ns_vs_svd`).  Muon's speed comes from hardware
   (matmuls run at tensor-core peak in bf16; SVD is sequential), which no flop count can show.
   Head–tail (`Cost.htCost`) is cheaper than both by a factor that grows like `s / r` (`ht_vs_svd`).
-/

namespace FastMuon.MuonVsSVD

open FastMuon.NSPoly FastMuon.Cost

/-! ## 1. One step does not contract the error -/

theorem claimed_contraction_false :
    ¬ ∀ σ : ℝ, 0.5 ≤ σ → σ ≤ 1.5 → |nsP σ − 1| < |σ − 1| := by
  intro h
  have h1 := h 1 (by norm_num) (by norm_num)
  norm_num [nsP] at h1

/-! ## 2. Five steps do not converge to within 0.05 -/

theorem nsP_ge_two {x : ℝ} (hx : 2 ≤ x) : 2 ≤ nsP x := by
  unfold nsP
  have hx2 : 4 ≤ x ^ 2 := by nlinarith
  have hin : 1 ≤ 2.0315 * x ^ 4 - 4.7750 * x ^ 2 + 3.4445 := by
    nlinarith [mul_nonneg (sub_nonneg.mpr hx2) (sub_nonneg.mpr hx2)]
  nlinarith [mul_le_mul_of_nonneg_left hin (by linarith : (0 : ℝ) ≤ x)]

theorem iter_ge_two (k : ℕ) {x : ℝ} (hx : 2 ≤ x) : 2 ≤ nsP^[k] x := by
  induction k with
```

```
  | zero => exact hx
  | succ k ih => rw [Function.iterate_succ_apply']; exact nsP_ge_two ih

/-- Without Frobenius normalisation the iteration blows up: `σ = 1.5` is a counterexample. -/
theorem claimed_five_step_false :
    ¬ ∀ σ : ℝ, 0.5 ≤ σ → σ ≤ 1.5 → |nsP^[5] σ - 1| < 0.05 := by
  intro h
  have h1 := h 1.5 (by norm_num) (by norm_num)
  have hp : 2 ≤ nsP 1.5 := by norm_num [nsP]
  have h5 : 2 ≤ nsP^[5] 1.5 := by
    rw [Function.iterate_succ_apply]; exact iter_ge_two 4 hp
  have := (abs_lt.mp h1).2
  linarith

-- interval steps for the orbit of σ = 1: 1 ↦ 0.701 ↦ [1.05,1.15] ↦ [0.66,0.80] ↦ [0.95,1.18] ↦ [0.65,0.90]
lemma stepA {x : ℝ} (h0 : 0.69 ≤ x) (h1 : x ≤ 0.73) : 1.05 ≤ nsP x ∧ nsP x ≤ 1.15 := by
  unfold nsP
  have a := sub_nonneg.mpr h0; have b := sub_nonneg.mpr h1
  constructor <;> nlinarith [mul_nonneg a b, mul_nonneg (mul_nonneg a b) a, mul_nonneg (mul_nonneg a b) b,
    mul_nonneg (mul_nonneg a a) a, mul_nonneg (mul_nonneg b b) b, sq_nonneg (x - 0.71)]

lemma stepB {x : ℝ} (h0 : 1.05 ≤ x) (h1 : x ≤ 1.15) : 0.66 ≤ nsP x ∧ nsP x ≤ 0.80 := by
  unfold nsP
  have a := sub_nonneg.mpr h0; have b := sub_nonneg.mpr h1
  constructor <;> nlinarith [mul_nonneg a b, mul_nonneg (mul_nonneg a b) a, mul_nonneg (mul_nonneg a b) b,
    mul_nonneg (mul_nonneg a a) a, mul_nonneg (mul_nonneg b b) b, sq_nonneg (x - 1.05)]

lemma stepC {x : ℝ} (h0 : 0.66 ≤ x) (h1 : x ≤ 0.80) : 0.95 ≤ nsP x ∧ nsP x ≤ 1.18 := by
  unfold nsP
  have a := sub_nonneg.mpr h0; have b := sub_nonneg.mpr h1
  constructor <;> nlinarith [mul_nonneg a b, mul_nonneg (mul_nonneg a b) a, mul_nonneg (mul_nonneg a b) b,
    mul_nonneg (mul_nonneg a a) a, mul_nonneg (mul_nonneg b b) b, sq_nonneg (x - 0.73)]

lemma stepD {x : ℝ} (h0 : 0.95 ≤ x) (h1 : x ≤ 1.18) : 0.65 ≤ nsP x ∧ nsP x ≤ 0.90 := by
  unfold nsP
  have a := sub_nonneg.mpr h0; have b := sub_nonneg.mpr h1
  constructor <;> nlinarith [mul_nonneg a b, mul_nonneg (mul_nonneg a b) a, mul_nonneg (mul_nonneg a b) b,
    mul_nonneg (mul_nonneg a a) a, mul_nonneg (mul_nonneg b b) b, sq_nonneg (x - 1.05)]

/-- Even from an already-orthogonal direction (`σ = 1`), five Muon steps end in `[0.65, 0.90]`. -/
theorem five_steps_from_one : 0.65 ≤ nsP^[5] 1 ∧ nsP^[5] 1 ≤ 0.90 := by
  have e1 : nsP 1 = 0.701 := by norm_num [nsP]
  have hA := stepA (x := nsP 1) (by rw [e1]; norm_num) (by rw [e1]; norm_num)
  have hB := stepB hA.1 hA.2
  have hC := stepC hB.1 hB.2
  have hD := stepD hC.1 hC.2
  simpa [Function.iterate_succ_apply'] using hD

theorem five_steps_from_one_not_close : 0.05 < |nsP^[5] 1 - 1| := by
  have h := five_steps_from_one
  rw [abs_sub_comm, abs_of_pos (by linarith [h.2])]
  linarith [h.2]

/-! ## 3. Flops: NS is not cheaper than SVD; head–tail is cheaper than both -/

/-- The write-up's NS cost (one `m·n·m` matmul per step). -/
def claimedNsCost (m n steps : ℕ) : ℕ := steps * (2 * m * n * m)
/-- Golub–Reinsch SVD (`Σ` and `V`) of an `m × n` matrix, `m ≥ n`, as in the write-up. -/
def svdCost (m n : ℕ) : ℕ := 4 * m * n ^ 2 + 8 * n ^ 3

theorem claimed_efficiency_false :
    ¬ ∀ m n : ℕ, 4096 ≤ m → 4096 ≤ n → claimedNsCost m n 5 < svdCost m n := by
  intro h
  have := h 8192 4096 (by norm_num) (by norm_num)
  norm_num [claimedNsCost, svdCost] at this

/-- **NS vs SVD in flops.** For an `s × l` matrix with `s ≤ l`, five correctly-counted NS steps
(`Cost.nsCost`) need between 2.5× and 5× the flops of a Golub–Reinsch SVD. -/
theorem ns_vs_svd (s l : ℕ) (hsl : s ≤ l) :
    5 * svdCost l s ≤ 2 * nsCost s l 5 ∧ nsCost s l 5 ≤ 5 * svdCost l s := by
  unfold svdCost nsCost
  have : s ^ 3 ≤ s ^ 2 * l := by
    calc s ^ 3 = s ^ 2 * s := by ring
      _ ≤ s ^ 2 * l := Nat.mul_le_mul_left _ hsl
  constructor <;> nlinarith

/-- **Head–tail vs SVD.** `s · htCost ≤ 12 r · svdCost`: the head–tail oracle is at least
`s / (12 r)` times cheaper than an SVD, in flops. -/
theorem ht_vs_svd (s l r : ℕ) (hr : 1 ≤ r) (hrs : r ≤ s) (hsl : s ≤ l) :
    s * htCost s l r ≤ 12 * r * svdCost l s := by
  have h1 := speedup s l r hr hrs hsl
```

```
  have h2 := (ns_vs_svd s l hsl).2
  have : 5 * (s * htCost s l r) ≤ 5 * (12 * r * svdCost l s) := by
    calc 5 * (s * htCost s l r) = 5 * s * htCost s l r := by ring
      _ ≤ 12 * r * nsCost s l 5 := h1
      _ ≤ 12 * r * (5 * svdCost l s) := Nat.mul_le_mul_left _ h2
      _ = 5 * (12 * r * svdCost l s) := by ring
  exact Nat.le_of_mul_le_mul_left this (by norm_num)

end FastMuon.MuonVsSVD
```

FastMuon/NSMatrix.lean (142 lines)

```
import FastMuon.NSPoly
import FastMuon.MatrixFacts

/-!
# Muon's Newton–Schulz quintic at the matrix level

`NSPoly.lean` proves that the scalar quintic `nsP x = a x + b x^3 + c x^5`,
`(a, b, c) = (3.4445, -4.7750, 2.0315)`, maps `[0, 1.25]` into itself, and appeals to spectral
calculus for the matrix statement.  Here we prove the matrix statement directly, for the spectral
norm `‖·‖` (Mathlib's `L2` operator norm) on real `s × l` matrices:

* `nsStep X = a • X + b • (X Xᵀ X) + c • (X Xᵀ X Xᵀ X)` is one Newton–Schulz step;
* `nsStep_eq`: `nsStep X = X q(XᵀX)` with `q(λ) = a + bλ + cλ²`;
* `nsStep_gram`: `(nsStep X)ᵀ (nsStep X) = g(XᵀX)` with `g(λ) = λ q(λ)²`;
* `nsG_eval_eq`: `g(λ) = nsP(√λ)²` for `λ ≥ 0`;
* `nsStep_norm_le`: `‖X‖ ≤ 1.25 → ‖nsStep X‖ ≤ 1.25`;
* `nsStep_iterate_norm_le`: the same for every iterate;
* `muon_norm_le`: Muon's output `nsStep^[5] (M / ‖M‖_F)` has spectral norm `≤ 1.25` for every real
  `M` (using `‖M‖₂ ≤ ‖M‖_F`).

The matrix step uses only `MatrixFacts.norm_aeval_le_of_posSemidef` (polynomial norm bound for a
PSD matrix, via Mathlib's isometric continuous functional calculus on real matrices), so the scalar
bound `NSPoly.nsP_mem` now transfers to matrices with no unformalized step.
-/

set_option linter.unusedSectionVars false

namespace FastMuon.NSMatrix

open Matrix Polynomial FastMuon.NSPoly FastMuon.MatrixFacts
open scoped Matrix.Norms.L2Operator

variable {s l : Type*} [Fintype s] [Fintype l] [DecidableEq s] [DecidableEq l]

/-- One step of Muon's Newton–Schulz iteration. -/
noncomputable def nsStep (X : Matrix s l ℝ) : Matrix s l ℝ :=
  (3.4445 : ℝ) • X + (-4.7750 : ℝ) • (X * Xᵀ * X) + (2.0315 : ℝ) • (X * Xᵀ * X * Xᵀ * X)

/-- `q(λ) = a + bλ + cλ²`, so that `nsStep X = X q(XᵀX)`. -/
noncomputable def nsQ : ℝ[X] := C 3.4445 + C (-4.7750) * X + C 2.0315 * X ^ 2

/-- `g(λ) = λ q(λ)²`, so that `(nsStep X)ᵀ (nsStep X) = g(XᵀX)`. -/
noncomputable def nsG : ℝ[X] := X * nsQ ^ 2

theorem nsQ_eval (x : ℝ) : nsQ.eval x = 3.4445 + -4.7750 * x + 2.0315 * x ^ 2 := by
  simp [nsQ]

theorem nsG_eval (x : ℝ) : nsG.eval x = x * (3.4445 + -4.7750 * x + 2.0315 * x ^ 2) ^ 2 := by
  simp [nsG, nsQ]

/-- `g(λ) = nsP(√λ)²` for `λ ≥ 0`. -/
theorem nsG_eval_eq {x : ℝ} (hx : 0 ≤ x) : nsG.eval x = nsP (Real.sqrt x) ^ 2 := by
  rw [nsG_eval, nsP]
  have hs : Real.sqrt x ^ 2 = x := Real.sq_sqrt hx
  have h3 : Real.sqrt x ^ 3 = Real.sqrt x * x := by rw [pow_succ, hs, mul_comm]
  have h5 : Real.sqrt x ^ 5 = Real.sqrt x * x ^ 2 := by
    rw [show (5 : ℕ) = 2 * 2 + 1 from rfl, pow_succ, pow_mul, hs, mul_comm]
  rw [h3, h5]
  have : (3.4445 * Real.sqrt x - 4.7750 * (Real.sqrt x * x) + 2.0315 * (Real.sqrt x * x ^ 2)) =
      Real.sqrt x * (3.4445 + -4.7750 * x + 2.0315 * x ^ 2) := by ring
  rw [this, mul_pow, hs]

/-- On `[0, 1.25²]`, `0 ≤ g ≤ 1.25²` (from the scalar bound `nsP_mem`). -/
theorem nsG_eval_mem {x : ℝ} (hx : 0 ≤ x) (hx1 : x ≤ 1.5625) :
    0 ≤ nsG.eval x ∧ nsG.eval x ≤ 1.5625 := by
  rw [nsG_eval_eq hx]
  have hy0 : 0 ≤ Real.sqrt x := Real.sqrt_nonneg x
```

```
  have hy1 : Real.sqrt x ≤ 1.25 := by
    calc Real.sqrt x ≤ Real.sqrt (1.25 ^ 2) := Real.sqrt_le_sqrt (by norm_num; linarith)
      _ = 1.25 := Real.sqrt_sq (by norm_num)
  obtain ⟨h0, h1⟩ := nsP_mem hy0 hy1
  exact ⟨sq_nonneg _, by nlinarith⟩

theorem nsStep_eq (X : Matrix s l ℝ) : nsStep X = X * aeval (Xᵀ * X) nsQ := by
  simp only [nsStep, nsQ, map_add, map_mul, aeval_C, aeval_X,
    Algebra.algebraMap_eq_smul_one, Matrix.mul_add, Matrix.mul_smul, Matrix.smul_mul,
    Matrix.mul_one, Matrix.one_mul, pow_two, Matrix.mul_assoc]

theorem aeval_nsQ_transpose (X : Matrix s l ℝ) :
    (aeval (Xᵀ * X) nsQ)ᵀ = aeval (Xᵀ * X) nsQ := by
  simp [nsQ, Algebra.algebraMap_eq_smul_one, transpose_add, transpose_smul, transpose_pow,
    transpose_mul]

/-- **Gram identity.** `(nsStep X)ᵀ (nsStep X) = g(XᵀX)`. -/
theorem nsStep_gram (X : Matrix s l ℝ) : (nsStep X)ᵀ * nsStep X = aeval (Xᵀ * X) nsG := by
  set A := Xᵀ * X with hA
  set Q := aeval A nsQ with hQ
  have hcomm : Commute A Q := by
    simpa only [aeval_X] using (Commute.all (Polynomial.X : ℝ[X]) nsQ).map (aeval A)
  have hQt : Qᵀ = Q := aeval_nsQ_transpose X
  rw [nsStep_eq, transpose_mul, hQt, nsG, map_mul, map_pow, aeval_X, ← hQ]
  calc Q * Xᵀ * (X * Q) = Q * A * Q := by simp only [hA, Matrix.mul_assoc]
    _ = A * Q * Q := by rw [hcomm.eq]
    _ = A * Q ^ 2 := by rw [pow_two, Matrix.mul_assoc]

/-- **One Newton–Schulz step keeps the spectral norm at most `1.25`.** -/
theorem nsStep_norm_le {X : Matrix s l ℝ} (hX : ‖X‖ ≤ 1.25) : ‖nsStep X‖ ≤ 1.25 := by
  have hpsd : (Xᵀ * X).PosSemidef := by
    simpa only [conjTranspose_eq_transpose_of_trivial] using posSemidef_conjTranspose_mul_self X
  have hAn : ‖Xᵀ * X‖ = ‖X‖ * ‖X‖ := by
    rw [← conjTranspose_eq_transpose_of_trivial, l2_opNorm_conjTranspose_mul_self]
  have hA : ‖Xᵀ * X‖ ≤ 1.5625 := by rw [hAn]; nlinarith [norm_nonneg X]
  have hG : ‖aeval (Xᵀ * X) nsG‖ ≤ 1.5625 :=
    norm_aeval_le_of_posSemidef hpsd nsG (by norm_num) fun x hx0 hx1 => by
      obtain ⟨h0, h1⟩ := nsG_eval_mem hx0 (hx1.trans hA)
      rw [abs_le]; constructor <;> linarith
  have hN : ‖nsStep X‖ * ‖nsStep X‖ ≤ 1.5625 := by
    rw [← l2_opNorm_conjTranspose_mul_self, conjTranspose_eq_transpose_of_trivial, nsStep_gram]
    exact hG
  nlinarith [norm_nonneg (nsStep X)]

/-- **Every Newton–Schulz iterate keeps the spectral norm at most `1.25`.** -/
theorem nsStep_iterate_norm_le (k : ℕ) {X : Matrix s l ℝ} (hX : ‖X‖ ≤ 1.25) :
    ‖nsStep^[k] X‖ ≤ 1.25 := by
  induction k with
  | zero => simpa using hX
  | succ k ih =>
    rw [Function.iterate_succ_apply']
    exact nsStep_norm_le ih

/-- Muon's normalisation `X₀ = M / ‖M‖_F` is a spectral contraction (`‖M‖₂ ≤ ‖M‖_F`). -/
theorem muon_init_norm_le_one (M : Matrix s l ℝ) : ‖(1 / frobNorm M) • M‖ ≤ 1 := by
  have hF := frobNorm_nonneg M
  rw [norm_smul, Real.norm_eq_abs, abs_of_nonneg (one_div_nonneg.mpr hF)]
  rcases hF.eq_or_lt with h | h
  · rw [← h]; simp
  · rw [one_div, inv_mul_le_iff₀ h, mul_one]
    exact l2_opNorm_le_frobNorm M

/-- **Muon's output has spectral norm at most `1.25`.**  For every real `M`, five Newton–Schulz
steps applied to `X₀ = M / ‖M‖_F` give `‖nsStep^[5] X₀‖₂ ≤ 1.25`.  (No `M ≠ 0` hypothesis is needed:
for `M = 0`, `X₀ = 0`.) -/
theorem muon_norm_le (M : Matrix s l ℝ) : ‖nsStep^[5] ((1 / frobNorm M) • M)‖ ≤ 1.25 :=
  nsStep_iterate_norm_le 5 ((muon_init_norm_le_one M).trans (by norm_num))

/-- The same, for any number of steps. -/
theorem muon_iterate_norm_le (k : ℕ) (M : Matrix s l ℝ) :
    ‖nsStep^[k] ((1 / frobNorm M) • M)‖ ≤ 1.25 :=
  nsStep_iterate_norm_le k ((muon_init_norm_le_one M).trans (by norm_num))

end FastMuon.NSMatrix
```

`FastMuon/NSPoly.lean` (101 lines)

```
import Mathlib

/-!
# Error bound for Muon's own Newton–Schulz step (scalar / singular-value level)

Muon applies the odd quintic `p(x) = 3.4445 x - 4.7750 x^3 + 2.0315 x^5` five times to `X / ‖X‖_F`.
For an odd polynomial, `p(X) = U p(Σ) V^T`; the matrix-level norm bound is proved in `NSMatrix.lean` (`muon_norm_le`), so the
singular values of the output are `p^[5]` of the normalised singular values `x ∈ [0, 1]`.
We prove the scalar facts:

* `[0, 1.25]` is invariant (`nsP_mem`): so Muon's output has spectral norm `≤ β ≤ 1.25`;
* `nsP_iter_ge`: after `k` steps, `p^[k] x ≥ min(0.6, 3^k x)`.

So five steps map every normalised singular value `x ≥ 0.6/243 ≈ 0.0025` into `[0.6, 1.25]`:
Muon itself is an `(α, β)`-certified oracle with `β ≤ 1.25`, and `α ≥ 0.6` on the part of the spectrum
above `0.0025 ‖X‖_F` — it is *not* the exact polar factor, which is why `Certificate.lean` is stated
for general `(α, β)`.
-/

namespace FastMuon.NSPoly

/-- Muon's quintic. -/
noncomputable def nsP (x : ℝ) : ℝ := 3.4445 * x - 4.7750 * x ^ 3 + 2.0315 * x ^ 5

theorem nsP_nonneg {x : ℝ} (hx : 0 ≤ x) : 0 ≤ nsP x := by
  unfold nsP
  have : 0 ≤ 3.4445 - 4.7750 * x ^ 2 + 2.0315 * x ^ 4 := by
    nlinarith [sq_nonneg (x ^ 2 - 1.1752)]
  nlinarith [mul_nonneg hx this]

theorem nsP_le {x : ℝ} (hx : 0 ≤ x) (hx1 : x ≤ 1.25) : nsP x ≤ 1.25 := by
  unfold nsP
  nlinarith [sq_nonneg (x - 0.5545), sq_nonneg (x ^ 2 - 0.3075),
    mul_nonneg hx (sq_nonneg (x ^ 2 - 0.3075)), mul_nonneg hx (sq_nonneg (x - 0.5545)),
    mul_nonneg (sub_nonneg.mpr hx1) hx,
    mul_nonneg (mul_nonneg (sub_nonneg.mpr hx1) hx) (sq_nonneg (x - 0.5545)),
    mul_nonneg (mul_nonneg (sub_nonneg.mpr hx1) hx) (sq_nonneg (x - 1.05))]

theorem nsP_mem {x : ℝ} (hx : 0 ≤ x) (hx1 : x ≤ 1.25) : 0 ≤ nsP x ∧ nsP x ≤ 1.25 :=
  ⟨nsP_nonneg hx, nsP_le hx hx1⟩

/-- Small singular values at least triple. -/
theorem nsP_ge_three {x : ℝ} (hx : 0 ≤ x) (hx1 : x ≤ 0.3) : 3 * x ≤ nsP x := by
  unfold nsP
  have hx2 : x ^ 2 ≤ 0.09 := by nlinarith
  have : 0 ≤ 0.4445 - 4.7750 * x ^ 2 + 2.0315 * x ^ 4 := by nlinarith [sq_nonneg (x ^ 2)]
  nlinarith [mul_nonneg hx this]

/-- Mid and large singular values land in `[0.6, 1.25]`. -/
theorem nsP_ge_band {x : ℝ} (hx : 0.3 ≤ x) (hx1 : x ≤ 1.25) : 0.6 ≤ nsP x := by
  unfold nsP
  nlinarith [sq_nonneg (x - 1.05), sq_nonneg (x ^ 2 - 1.1028),
    mul_nonneg (sub_nonneg.mpr hx) (sub_nonneg.mpr hx1),
    mul_nonneg (mul_nonneg (sub_nonneg.mpr hx) (sub_nonneg.mpr hx1)) (sq_nonneg (x - 1.05)),
    mul_nonneg (sub_nonneg.mpr hx) (sq_nonneg (x - 1.05)),
    mul_nonneg (sub_nonneg.mpr hx1) (sq_nonneg (x - 1.05)),
    mul_nonneg (mul_nonneg (sub_nonneg.mpr hx) (sub_nonneg.mpr hx)) (sub_nonneg.mpr hx1)]

theorem nsP_iter_mem (k : ℕ) {x : ℝ} (hx : 0 ≤ x) (hx1 : x ≤ 1.25) :
    0 ≤ nsP^[k] x ∧ nsP^[k] x ≤ 1.25 := by
  induction k with
  | zero => exact ⟨hx, hx1⟩
  | succ k ih =>
    rw [Function.iterate_succ_apply']
    exact nsP_mem ih.1 ih.2

/-- **NS error bound.** After `k` steps every normalised singular value `x ∈ [0, 1.25]` satisfies
`min(0.6, 3^k x) ≤ p^[k] x ≤ 1.25`. -/
theorem nsP_iter_ge (k : ℕ) {x : ℝ} (hx : 0 ≤ x) (hx1 : x ≤ 1.25) :
    min 0.6 (3 ^ k * x) ≤ nsP^[k] x := by
  induction k with
  | zero => simp
  | succ k ih =>
    rw [Function.iterate_succ_apply']
    obtain ⟨h0, h1⟩ := nsP_iter_mem k hx hx1
    by_cases hk : 0.3 ≤ nsP^[k] x
    · exact (min_le_left _ _).trans (nsP_ge_band hk h1)
    · rw [not_le] at hk
      have h3 := nsP_ge_three h0 hk.le
      have hmin : min 0.6 (3 ^ k * x) = 3 ^ k * x := by
```

```
        rcases min_cases (0.6 : ℝ) (3 ^ k * x) with ⟨h, _⟩ | ⟨h, _⟩
        · rw [h] at ih; linarith
        · exact h
      rw [hmin] at ih
      calc min 0.6 (3 ^ (k + 1) * x) ≤ 3 ^ (k + 1) * x := min_le_right _ _
        _ = 3 * (3 ^ k * x) := by ring
        _ ≤ 3 * nsP^[k] x := by linarith
        _ ≤ nsP (nsP^[k] x) := h3

/-- Muon's five steps: every normalised singular value `x ≥ 0.6/243` ends in `[0.6, 1.25]`. -/
theorem muon_five_steps {x : ℝ} (hx : 0.6 / 243 ≤ x) (hx1 : x ≤ 1.25) :
    0.6 ≤ nsP^[5] x ∧ nsP^[5] x ≤ 1.25 := by
  have hx0 : 0 ≤ x := le_trans (by norm_num) hx
  refine ⟨?_, (nsP_iter_mem 5 hx0 hx1).2⟩
  have h := nsP_iter_ge 5 hx0 hx1
  have : (0.6 : ℝ) ≤ 3 ^ 5 * x := by
    have : (3 : ℝ) ^ 5 = 243 := by norm_num
    rw [this]; linarith [show (0.6:ℝ) / 243 * 243 = 0.6 by norm_num, mul_le_mul_of_nonneg_left hx (by norm_num : (0:ℝ) ≤ 243)]
  rwa [min_eq_left this] at h

end FastMuon.NSPoly
```

FastMuon/OptimalTail.lean (44 lines)

```
import Mathlib

/-!
# The certificate-optimal tail scale for head–tail

For the head–tail direction `O = U P + c • T`, `HeadTail.lean` proves
`⟪X, O⟫ = a + c b` with `a = ⟪Uᵀ X, P⟫`, `b = ‖T‖_F²`, and `‖O‖₂ ≤ √(p² + c² t²)` with `p = ‖P‖₂`,
`t = ‖T‖₂`.  So the *certified* quality `⟪X, O⟫ / β_bound` of a tail scale `c` is
`(a + c b) / √(p² + c² t²)`.

* `certified_ratio_le`: for every `c ≥ 0`, `(a + c b) ≤ √((a/p)² + (b/t)²) · √(p² + c² t²)`
  (Cauchy–Schwarz);
* `certified_ratio_opt`: equality at `c* = b p² / (a t²)`.

So the best tail scale for the guarantee is `c* = ‖T‖_F² ‖P‖² / (⟪H, P⟫ ‖T‖₂²)`, computable in `O(s l)`
from quantities the oracle already has; `py/ortho.py` exposes it as `tau_mode="opt"`.  Note that `c*` can
exceed `1/‖T‖₂`: the guarantee sometimes prefers to over-weight the tail.
-/

namespace FastMuon.OptimalTail

theorem certified_ratio_le {a b t p c : ℝ} (ht : 0 < t) (hp : 0 < p) :
    a + c * b ≤ Real.sqrt ((a / p) ^ 2 + (b / t) ^ 2) * Real.sqrt (p ^ 2 + c ^ 2 * t ^ 2) := by
  set u := a / p with hu
  set v := b / t with hv
  have ha : a = u * p := by rw [hu]; field_simp
  have hb : b = v * t := by rw [hv]; field_simp
  rw [← Real.sqrt_mul (by positivity)]
  apply Real.le_sqrt_of_sq_le
  rw [ha, hb]
  nlinarith [sq_nonneg (u * (c * t) - v * p)]

theorem certified_ratio_opt {a b t p : ℝ} (ha : 0 < a) (hb : 0 ≤ b) (ht : 0 < t) (hp : 0 < p) :
    let c := b * p ^ 2 / (a * t ^ 2)
    a + c * b = Real.sqrt ((a / p) ^ 2 + (b / t) ^ 2) * Real.sqrt (p ^ 2 + c ^ 2 * t ^ 2) := by
  intro c
  have hc : 0 ≤ c := by positivity
  rw [← Real.sqrt_mul (by positivity)]
  have hlhs : 0 ≤ a + c * b := by positivity
  rw [eq_comm, Real.sqrt_eq_iff_mul_self_eq (by positivity) hlhs]
  simp only [c]
  field_simp

end FastMuon.OptimalTail
```

`FastMuon/Polish.lean` (122 lines)

```
import FastMuon.NSMatrix
import FastMuon.HeadTail

/-!
# One cubic polishing step after head-tail: `β ≤ 1`, certified

Head-tail gives a direction with `‖O‖₂ ≤ √2` (`HeadTail.htDir_norm_le_sqrt_two`).  One cubic
Newton-Schulz step `O' = (3/2) O - (1/2) O Oᵀ O` costs two products (`4 s² l` flops, a fifth of Muon's
five quintic steps) and, measured on real momentum, lifts the certificate quality above Muon's
(`κ`: 0.75 for Muon, 0.73 with one step after rank-16 head-tail, 0.81 with two, 0.91 with one after rank 64).

Here we prove the norm half of that certificate at the matrix level:

* `cubic_gram`: `O'ᵀ O' = g(OᵀO)` with `g(λ) = λ (3/2 - λ/2)²`;
* `one_sub_g`: `1 - g(λ) = (λ - 1)² (4 - λ) / 4`, so `0 ≤ g ≤ 1` on `[0, 4]`;
* `cubicStep_norm_le_one`: `‖O‖₂ ≤ 2 → ‖O'‖₂ ≤ 1`;
* `polished_ht_norm_le_one`: the polished head-tail direction has `‖O'‖₂ ≤ 1`, i.e. `β = 1`.
-/

set_option linter.unusedSectionVars false

namespace FastMuon.Polish

open Matrix Polynomial FastMuon.MatrixFacts FastMuon.HeadTail
open scoped Matrix.Norms.L2Operator

variable {s l r : Type*} [Fintype s] [Fintype l] [Fintype r]
  [DecidableEq s] [DecidableEq l] [DecidableEq r]

/-- One cubic Newton-Schulz step. -/
noncomputable def cubicStep (X : Matrix s l ℝ) : Matrix s l ℝ :=
  (1.5 : ℝ) • X + (-0.5 : ℝ) • (X * Xᵀ * X)

noncomputable def cQ : ℝ[X] := C 1.5 + C (-0.5) * X
noncomputable def cG : ℝ[X] := X * cQ ^ 2

theorem cG_eval (x : ℝ) : cG.eval x = x * (1.5 + -0.5 * x) ^ 2 := by
  simp [cG, cQ]

theorem one_sub_g (x : ℝ) : 1 - cG.eval x = (x - 1) ^ 2 * (4 - x) / 4 := by
  rw [cG_eval]; ring

theorem cG_eval_mem {x : ℝ} (hx : 0 ≤ x) (hx1 : x ≤ 4) : 0 ≤ cG.eval x ∧ cG.eval x ≤ 1 := by
  constructor
  · rw [cG_eval]; positivity
  · have h := one_sub_g x
    have : 0 ≤ (x - 1) ^ 2 * (4 - x) / 4 := by
      apply div_nonneg _ (by norm_num); exact mul_nonneg (sq_nonneg _) (by linarith)
    linarith

theorem cubicStep_eq (X : Matrix s l ℝ) : cubicStep X = X * aeval (Xᵀ * X) cQ := by
  simp only [cubicStep, cQ, map_add, map_mul, aeval_C, aeval_X,
    Algebra.algebraMap_eq_smul_one, Matrix.mul_add, Matrix.mul_smul, Matrix.smul_mul,
    Matrix.mul_one, Matrix.one_mul, Matrix.mul_assoc]

theorem aeval_cQ_transpose (X : Matrix s l ℝ) : (aeval (Xᵀ * X) cQ)ᵀ = aeval (Xᵀ * X) cQ := by
  simp [cQ, Algebra.algebraMap_eq_smul_one, transpose_add, transpose_smul, transpose_mul]

/-- **Gram identity.** `(cubicStep X)ᵀ (cubicStep X) = g(XᵀX)`. -/
theorem cubic_gram (X : Matrix s l ℝ) : (cubicStep X)ᵀ * cubicStep X = aeval (Xᵀ * X) cG := by
  set A := Xᵀ * X with hA
  set Q := aeval A cQ with hQ
  have hcomm : Commute A Q := by
    simpa only [aeval_X] using (Commute.all (Polynomial.X : ℝ[X]) cQ).map (aeval A)
  have hQt : Qᵀ = Q := aeval_cQ_transpose X
  rw [cubicStep_eq, transpose_mul, hQt, cG, map_mul, map_pow, aeval_X, ← hQ]
  calc Q * Xᵀ * (X * Q) = Q * A * Q := by simp only [hA, Matrix.mul_assoc]
    _ = A * Q * Q := by rw [hcomm.eq]
    _ = A * Q ^ 2 := by rw [pow_two, Matrix.mul_assoc]

/-- **One cubic step maps the spectral ball of radius 2 into the unit ball.** -/
theorem cubicStep_norm_le_one {X : Matrix s l ℝ} (hX : ‖X‖ ≤ 2) : ‖cubicStep X‖ ≤ 1 := by
  have hpsd : (Xᵀ * X).PosSemidef := by
    simpa only [conjTranspose_eq_transpose_of_trivial] using posSemidef_conjTranspose_mul_self X
  have hAn : ‖Xᵀ * X‖ = ‖X‖ * ‖X‖ := by
    rw [← conjTranspose_eq_transpose_of_trivial, l2_opNorm_conjTranspose_mul_self]
  have hA : ‖Xᵀ * X‖ ≤ 4 := by rw [hAn]; nlinarith [norm_nonneg X]
```

```
  have hG : ‖aeval (X^T * X) cG‖ ≤ 1 :=
    norm_aeval_le_of_posSemidef hpsd cG (by norm_num) fun x hx0 hx1 => by
      obtain ⟨h0, h1⟩ := cG_eval_mem hx0 (hx1.trans hA)
      rw [abs_le]; constructor <;> linarith
  have hN : ‖cubicStep X‖ * ‖cubicStep X‖ ≤ 1 := by
    rw [← l2_opNorm_conjTranspose_mul_self, conjTranspose_eq_transpose_of_trivial, cubic_gram]
    exact hG
  nlinarith [norm_nonneg (cubicStep X)]

/-- **Polished head-tail is a contraction (`β = 1`).** Under the hypotheses of
`htDir_norm_le_sqrt_two` (orthonormal frame, contractive head factor, `τ ≥ ‖T‖₂`), one cubic step on
the head-tail direction gives spectral norm at most `1`. -/
theorem polished_ht_norm_le_one {U : Matrix s r ℝ} (hU : U^T * U = 1) {P : Matrix r l ℝ} {τ : ℝ}
    {X : Matrix s l ℝ} (hP : ‖P‖ ≤ 1) (hτ : 0 < τ) (hT : ‖tail U X‖ ≤ τ) :
    ‖cubicStep (htDir U P τ X)‖ ≤ 1 := by
  apply cubicStep_norm_le_one
  have h := htDir_norm_le_sqrt_two hU hP hτ hT
  have h2 : Real.sqrt 2 ≤ 2 := by
    have h4 : Real.sqrt 4 = 2 := by
      rw [show (4:ℝ) = 2 ^ 2 by norm_num, Real.sqrt_sq (by norm_num)]
    calc Real.sqrt 2 ≤ Real.sqrt 4 := Real.sqrt_le_sqrt (by norm_num)
      _ = 2 := h4
  linarith

/-- **Robust version.** Polishing tolerates a tail-scale estimate that is off by up to `√3`: if
`‖T‖₂ ≤ √3 τ` (instead of `‖T‖₂ ≤ τ`), the head-tail direction still has `‖O‖₂ ≤ 2`, so one cubic step
still gives `‖O'‖₂ ≤ 1`.  A power-iteration estimate of `‖T‖₂` only has to be within a factor `1.73`. -/
theorem polished_ht_norm_le_one_robust {U : Matrix s r ℝ} (hU : U^T * U = 1) {P : Matrix r l ℝ}
    {τ : ℝ} {X : Matrix s l ℝ} (hP : ‖P‖ ≤ 1) (hτ : 0 < τ)
    (hT : ‖tail U X‖ ≤ Real.sqrt 3 * τ) :
    ‖cubicStep (htDir U P τ X)‖ ≤ 1 := by
  apply cubicStep_norm_le_one
  have h := htDir_norm_sq_le hU P τ X
  have hP2 : ‖P‖ * ‖P‖ ≤ 1 := by nlinarith [norm_nonneg P]
  have hq : τ⁻¹ * ‖tail U X‖ ≤ Real.sqrt 3 := by
    rw [inv_mul_le_iff₀ hτ]; linarith
  have hq0 : 0 ≤ τ⁻¹ * ‖tail U X‖ := mul_nonneg (inv_nonneg.mpr hτ.le) (norm_nonneg _)
  have h3 : Real.sqrt 3 * Real.sqrt 3 = 3 := Real.mul_self_sqrt (by norm_num)
  have hT2 : (τ⁻¹ * τ⁻¹) * (‖tail U X‖ * ‖tail U X‖) ≤ 3 := by
    have : (τ⁻¹ * ‖tail U X‖) * (τ⁻¹ * ‖tail U X‖) ≤ Real.sqrt 3 * Real.sqrt 3 :=
      mul_le_mul hq hq hq0 (Real.sqrt_nonneg 3)
    nlinarith
  nlinarith [norm_nonneg (htDir U P τ X)]

end FastMuon.Polish
```

FastMuon/Separation.lean (99 lines)

```
import FastMuon.MuonVsSVD

/-!
# Separation: neither Muon nor the cheap head-tail oracle dominates

The certificate quality of a direction `O` for a matrix `M` is `κ = ⟨M, O⟩ / (‖M‖_* ‖O‖₂)`
(the rate in `Certificate.lean` degrades by `1/κ`; the exact polar factor has `κ = 1`).
For a spectral map `σ ↦ φ(σ)` (any odd matrix polynomial, the exact polar factor, or head-tail with an
exact subspace), `⟨M, O⟩ = Σ σᵢ φ(σᵢ)`, `‖M‖_* = Σ σᵢ`, `‖O‖₂ = maxᵢ φ(σᵢ)` — standard spectral calculus,
which we take as the bridge and do not re-prove here.

**Spiked spectrum** (one singular value `1`, `4095` equal to `ε = 4·10⁻⁴`, i.e. a 4096 × 4096 matrix):

* Muon's five Frobenius-normalised Newton–Schulz steps give `κ ≤ 0.57` (`ns5_spiked_kappa`; numerically 0.551);
* rank-1 head-tail with tail scale `τ = ε` gives `κ = 1` (`ht_spiked_kappa`), at `O(mn)` cost
  instead of `O(mn²)`.

So "Newton–Schulz is the gold standard and cheaper methods can only lose" is false; on the power-law
spectra we measured in training the comparison goes the other way at small rank, so neither dominates.
-/

namespace FastMuon.Separation

open FastMuon.NSPoly FastMuon.MuonVsSVD

/-- On `[0, 1.25]` the quintic is below its linear part: `p x ≤ 3.4445 x`. -/
theorem nsP_le_lin {x : ℝ} (hx : 0 ≤ x) (hx1 : x ≤ 1.25) : nsP x ≤ 3.4445 * x := by
  unfold nsP
  have hx2 : x ^ 2 ≤ 1.5625 := by nlinarith
  nlinarith [mul_nonneg (pow_nonneg hx 3) (by nlinarith : (0:ℝ) ≤ 4.7750 - 2.0315 * x ^ 2)]

```

```
/-- Small singular values grow at most geometrically: `p^[k] x ≤ 3.4445^k x`. -/
theorem iter_le_lin (k : ℕ) {x : ℝ} (hx : 0 ≤ x) (hx1 : x ≤ 1.25) :
    nsP^[k] x ≤ 3.4445 ^ k * x := by
  induction k with
  | zero => simp
  | succ k ih =>
    rw [Function.iterate_succ_apply']
    obtain ⟨h0, h1⟩ := nsP_iter_mem k hx hx1
    calc nsP (nsP^[k] x) ≤ 3.4445 * nsP^[k] x := nsP_le_lin h0 h1
      _ ≤ 3.4445 * (3.4445 ^ k * x) := by linarith
      _ = 3.4445 ^ (k + 1) * x := by ring

lemma step0 {x : ℝ} (h0 : 0.99 ≤ x) (h1 : x ≤ 1) : 0.69 ≤ nsP x ∧ nsP x ≤ 0.73 := by
  unfold nsP
  have a := sub_nonneg.mpr h0; have b := sub_nonneg.mpr h1
  constructor <;> nlinarith [mul_nonneg a b, mul_nonneg (mul_nonneg a b) a, mul_nonneg (mul_nonneg a b) b,
    mul_nonneg (mul_nonneg a a) a, mul_nonneg (mul_nonneg b b) b, sq_nonneg (x - 1)]

/-- The top singular value, normalised by a Frobenius norm within 1% of it, ends in `[0.65, 0.90]`. -/
theorem five_steps_near_one {x : ℝ} (h0 : 0.99 ≤ x) (h1 : x ≤ 1) :
    0.65 ≤ nsP^[5] x ∧ nsP^[5] x ≤ 0.90 := by
  have h := step0 h0 h1
  have hA := stepA h.1 h.2
  have hB := stepB hA.1 hA.2
  have hC := stepC hB.1 hB.2
  have hD := stepD hC.1 hC.2
  simpa [Function.iterate_succ_apply'] using hD

/-- **Muon on a spiked spectrum.** `F` is the Frobenius norm `√(1 + 4095 ε^2)`, `ε = 4·10^-4`
(any `1 ≤ F ≤ 1/0.99` works). With `y = p^[5](1/F)`, `z = p^[5](ε/F)` the singular values Muon outputs,
the certificate quality is at most `0.57`. -/
theorem ns5_spiked_kappa (F : ℝ) (hF1 : 1 ≤ F) (hF2 : F ≤ 1 / 0.99) :
    (nsP^[5] (1 / F) + 4095 * 4e-4 * nsP^[5] (4e-4 / F)) /
        ((1 + 4095 * 4e-4) * max (nsP^[5] (1 / F)) (nsP^[5] (4e-4 / F))) ≤ 0.57 := by
  have hFpos : 0 < F := by linarith
  have hx0 : (0.99 : ℝ) ≤ 1 / F := by
    rw [le_div_iff₀ hFpos]
    have h := mul_le_mul_of_nonneg_left hF2 (by norm_num : (0:ℝ) ≤ 0.99)
    have e : (0.99 : ℝ) * (1 / 0.99) = 1 := by norm_num
    linarith
  have hx1 : 1 / F ≤ 1 := by rw [div_le_one hFpos]; exact hF1
  obtain ⟨hy0, hy1⟩ := five_steps_near_one hx0 hx1
  have he0 : (0 : ℝ) ≤ 4e-4 / F := by positivity
  have he1 : 4e-4 / F ≤ 4e-4 := by
    rw [div_le_iff₀ hFpos]; nlinarith
  have hz0 := (nsP_iter_mem 5 he0 (by linarith)).1
  have hz1 : nsP^[5] (4e-4 / F) ≤ 3.4445 ^ 5 * 4e-4 :=
    (iter_le_lin 5 he0 (by linarith)).trans (mul_le_mul_of_nonneg_left he1 (by positivity))
  set y := nsP^[5] (1 / F)
  set z := nsP^[5] (4e-4 / F)
  have hmax : y ≤ max y z := le_max_left _ _
  have hden : 0 < (1 + 4095 * 4e-4) * max y z := by
    have : (0:ℝ) < max y z := lt_of_lt_of_le (by linarith) hmax
    positivity
  rw [div_le_iff₀ hden]
  have hzsmall : z ≤ 0.19396 := by
    have : (3.4445 : ℝ) ^ 5 * 4e-4 ≤ 0.19396 := by norm_num
    linarith
  nlinarith [mul_le_mul_of_nonneg_left hmax (by norm_num : (0:ℝ) ≤ 0.57 * (1 + 4095 * 4e-4))]

/-- **Rank-1 head–tail on the same spectrum.** With the exact top singular direction and `τ = ε`, the
spectral map is `φ(1) = 1`, `φ(ε) = ε/τ = 1`, so `κ = 1` — at `O(mn)` cost. -/
theorem ht_spiked_kappa :
    (1 * (1 : ℝ) + 4095 * 4e-4 * ((4e-4 : ℝ) / 4e-4)) /
      ((1 + 4095 * 4e-4) * max (1 : ℝ) ((4e-4 : ℝ) / 4e-4)) = 1 := by
  norm_num

end FastMuon.Separation
```

`FastMuon/Spectral.lean` (158 lines)

```
import FastMuon.Certificate
import FastMuon.HeadTail

/-!
# Muon's geometry on real matrices, and the end-to-end head-tail theorem

We instantiate `Geometry` on `Matrix s l ℝ` with

* the Frobenius pairing `⟪A, B⟫ = tr (Aᵀ B)`,
* `N` = Mathlib's spectral norm (`L2` operator norm),
* `D` = **the dual norm of the spectral norm**, `nuc G = sup_{‖O‖₂ ≤ 1} ⟪G, O⟫`
  (mathematically this is the nuclear norm; we never need its singular-value formula).

Hölder's inequality then holds by construction, so `Geometry.muon_family_rate` applies to matrices in
exactly the geometry where Muon is steepest descent.

On top of that we prove:

* `nuc_frame_le`: an orthonormal frame cannot increase the nuclear norm: `nuc (U H) ≤ nuc H`;
* `ht_alignment_lower` (**alignment error bound**): for the head-tail direction `O`,
  `⟪X, O⟫ ≥ α_h nuc X - (α_h nuc T - ‖T‖_F²/τ)`, where `α_h` is the head factor's own certificate
  (`α_h = 1` for an exact polar factor).  The only loss is the tail's nuclear mass not recovered by
  the scaled tail;
* `ht_muon_rate`: **HT-Muon with momentum converges at Muon's rate**, with `β = √2` and
  `α = α_h - (tail loss fraction)`.
-/

set_option linter.unusedSectionVars false

namespace FastMuon.Spectral

open Matrix FastMuon.HeadTail Finset
open scoped Matrix.Norms.L2Operator

variable {s l r : Type*} [Fintype s] [Fintype l] [Fintype r]
  [DecidableEq s] [DecidableEq l] [DecidableEq r]

/-- Frobenius pairing as a bilinear map. -/
def frobBilin : Matrix s l ℝ →ₗ[ℝ] Matrix s l ℝ →ₗ[ℝ] ℝ :=
  LinearMap.mk₂ ℝ (fun A B => frob A B)
    (fun A₁ A₂ B => by simp [frob, transpose_add, Matrix.add_mul, trace_add])
    (fun c A B => by simp [frob, transpose_smul, Matrix.smul_mul, trace_smul])
    (fun A B₁ B₂ => by simp [frob, Matrix.mul_add, trace_add])
    (fun c A B => by simp [frob, Matrix.mul_smul, trace_smul])

@[simp] theorem frobBilin_apply (A B : Matrix s l ℝ) : frobBilin A B = frob A B := rfl

/-- `G ↦ ⟪G, ·⟫` as a continuous functional on `(Matrix s l ℝ, ‖·‖₂)`. -/
noncomputable def frobCLM : Matrix s l ℝ →ₗ[ℝ] (Matrix s l ℝ →L[ℝ] ℝ) :=
  (LinearMap.toContinuousLinearMap : (Matrix s l ℝ →ₗ[ℝ] ℝ) ≃ₗ[ℝ] _).toLinearMap ∘ₗ frobBilin

@[simp] theorem frobCLM_apply (A B : Matrix s l ℝ) : frobCLM A B = frob A B := rfl

/-- The nuclear norm as the dual of the spectral norm. -/
noncomputable def nuc : Seminorm ℝ (Matrix s l ℝ) :=
  (normSeminorm ℝ (Matrix s l ℝ →L[ℝ] ℝ)).comp frobCLM

theorem nuc_apply (A : Matrix s l ℝ) : nuc A = ‖frobCLM A‖ := rfl

/-- Hölder / von Neumann: `|⟪A, B⟫| ≤ ‖A‖_* ‖B‖₂`. -/
theorem holder (A B : Matrix s l ℝ) : |frob A B| ≤ nuc A * ‖B‖ := by
  have := (frobCLM A).le_opNorm B
  simpa [nuc_apply, Real.norm_eq_abs] using this

/-- Muon's geometry on `s × l` real matrices. -/
noncomputable def geometry : Geometry (Matrix s l ℝ) where
  ip := frobBilin
  N := normSeminorm ℝ _
  D := nuc
  holder A B := by simpa [coe_normSeminorm] using holder A B

theorem norm_one_le : ‖(1 : Matrix r r ℝ)‖ ≤ 1 := by
  have h := l2_opNorm_conjTranspose_mul_self (1 : Matrix r r ℝ)
  rw [conjTranspose_one, Matrix.one_mul] at h
  nlinarith [norm_nonneg (1 : Matrix r r ℝ)]

/-- An orthonormal frame is a contraction. -/
theorem frame_norm_le {U : Matrix s r ℝ} (hU : Uᵀ * U = 1) : ‖U‖ ≤ 1 := by
  have h := l2_opNorm_conjTranspose_mul_self U
```

```
  rw [conjTranspose_eq_transpose_of_trivial, hU] at h
  nlinarith [norm_nonneg U, norm_one_le (r := r)]

theorem frob_frame (U : Matrix s r ℝ) (H : Matrix r l ℝ) (O : Matrix s l ℝ) :
    frob (U * H) O = frob H (Uᵀ * O) := by
  simp [frob, transpose_mul, Matrix.mul_assoc]

/-- An orthonormal frame cannot increase the nuclear norm: `‖U H‖_* ≤ ‖H‖_*`. -/
theorem nuc_frame_le {U : Matrix s r ℝ} (hU : Uᵀ * U = 1) (H : Matrix r l ℝ) :
    nuc (U * H) ≤ nuc H := by
  rw [nuc_apply]
  refine ContinuousLinearMap.opNorm_le_bound _ (apply_nonneg _ _) fun O => ?_
  rw [frobCLM_apply, frob_frame, Real.norm_eq_abs]
  calc |frob H (Uᵀ * O)| ≤ nuc H * ‖Uᵀ * O‖ := holder _ _
    _ ≤ nuc H * (‖Uᵀ‖ * ‖O‖) :=
        mul_le_mul_of_nonneg_left (l2_opNorm_mul _ _) (apply_nonneg _ _)
    _ ≤ nuc H * (1 * ‖O‖) := by
        apply mul_le_mul_of_nonneg_left _ (apply_nonneg _ _)
        apply mul_le_mul_of_nonneg_right _ (norm_nonneg _)
        rw [← conjTranspose_eq_transpose_of_trivial, l2_opNorm_conjTranspose]
        exact frame_norm_le hU
    _ = nuc H * ‖O‖ := by ring

/-- **Alignment error bound for the head–tail direction.** If the head factor `P` is
`α_h`-certified for the head `H = Uᵀ X`, then
`⟪X, O⟫ ≥ α_h ‖X‖_* - (α_h ‖T‖_* - ‖T‖_F² / τ)`. -/
theorem ht_alignment_lower {U : Matrix s r ℝ} (hU : Uᵀ * U = 1) (P : Matrix r l ℝ) (τ : ℝ)
    (X : Matrix s l ℝ) {αh : ℝ} (hαh : 0 ≤ αh) (hP : αh * nuc (Uᵀ * X) ≤ frob (Uᵀ * X) P) :
    αh * nuc X - (αh * nuc (tail U X) - τ⁻¹ * frob (tail U X) (tail U X)) ≤
      frob X (htDir U P τ X) := by
  rw [align hU P τ X]
  have hsplit : X = U * (Uᵀ * X) + tail U X := by simp [tail]
  have htri : nuc X ≤ nuc (U * (Uᵀ * X)) + nuc (tail U X) := by
    conv_lhs => rw [hsplit]
    exact map_add_le_add _ _ _
  have hfr := nuc_frame_le hU (Uᵀ * X)
  have h1 : αh * nuc X ≤ αh * nuc (Uᵀ * X) + αh * nuc (tail U X) := by
    have := mul_le_mul_of_nonneg_left (htri.trans (add_le_add_left hfr _)) hαh
    linarith
  linarith

/-- **HT-Muon converges at Muon's rate.**  Run `x (t+1) = x t - η • O t` with the head–tail direction
`O t = htDir (U t) (P t) (τ t) (m t)` computed from the EMA momentum `m t` of noisy gradients.
Assume, at every step: the frame is orthonormal, the head factor is a contraction with certificate
`α_h`, the tail scale dominates the tail (`‖T‖₂ ≤ τ`), and the tail loss is small:
`α_h ‖T‖_* - ‖T‖_F²/τ ≤ (α_h - α) ‖m‖_*`.  Then, with `β = √2`,

`avg_t ‖∇f(x t)‖_* ≤ (f x₀ - f_low)/(α η T) + L η·2/(2α) + (α+√2)/α · (μ L η √2/(1-μ) + σ)`. -/
theorem ht_muon_rate {f : Matrix s l ℝ → ℝ} {grad : Matrix s l ℝ → Matrix s l ℝ} {L : ℝ}
    (hf : (geometry (s := s) (l := l)).Smooth f grad L)
    {x m ghat : ℕ → Matrix s l ℝ} {U : ℕ → Matrix s r ℝ} {P : ℕ → Matrix r l ℝ} {τ : ℕ → ℝ}
    {η α αh μ σ flow : ℝ} (hη : 0 < η) (hα : 0 < α) (hαh : α ≤ αh) (hμ0 : 0 ≤ μ) (hμ1 : μ < 1)
    (hU : ∀ t, (U t)ᵀ * U t = 1) (hP : ∀ t, ‖P t‖ ≤ 1)
    (hPc : ∀ t, αh * nuc ((U t)ᵀ * m t) ≤ frob ((U t)ᵀ * m t) (P t))
    (hτ : ∀ t, 0 < τ t) (hT : ∀ t, ‖tail (U t) (m t)‖ ≤ τ t)
    (hloss : ∀ t, αh * nuc (tail (U t) (m t)) - (τ t)⁻¹ * frob (tail (U t) (m t)) (tail (U t) (m t))
      ≤ (αh - α) * nuc (m t))
    (hstep : ∀ t, x (t + 1) = x t - η • htDir (U t) (P t) (τ t) (m t))
    (hm0 : m 0 = ghat 0) (hm : ∀ t, m (t + 1) = μ • m t + (1 - μ) • ghat (t + 1))
    (hnoise : ∀ t, nuc (ghat t - grad (x t)) ≤ σ)
    (hlow : ∀ y, flow ≤ f y) {T : ℕ} (hT0 : 0 < T) :
    (∑ t ∈ range T, nuc (grad (x t))) / T ≤
      (f (x 0) - flow) / (α * η * T) + L * η * Real.sqrt 2 ^ 2 / (2 * α) +
        (α + Real.sqrt 2) / α * (μ * (L * η * Real.sqrt 2) / (1 - μ) + σ) := by
  have hαh0 : 0 ≤ αh := le_trans hα.le hαh
  have horacle : ∀ t, α * (geometry (s := s) (l := l)).D (m t) ≤
      (geometry (s := s) (l := l)).ip (m t) (htDir (U t) (P t) (τ t) (m t)) := by
    intro t
    have h := ht_alignment_lower (hU t) (P t) (τ t) (m t) hαh0 (hPc t)
    show α * nuc (m t) ≤ frob (m t) (htDir (U t) (P t) (τ t) (m t))
    nlinarith [hloss t, h]
  have hN : ∀ t, (geometry (s := s) (l := l)).N (htDir (U t) (P t) (τ t) (m t)) ≤ Real.sqrt 2 := by
    intro t
    show ‖htDir (U t) (P t) (τ t) (m t)‖ ≤ Real.sqrt 2
    exact htDir_norm_le_sqrt_two (hU t) (hP t) (hτ t) (hT t)
  exact (geometry (s := s) (l := l)).muon_family_rate hf hη hα (Real.sqrt_nonneg 2) hμ0 hμ1
    hstep horacle hN hm0 hm hnoise hlow hT0

end FastMuon.Spectral
```

Pilot

`Pilot/O4_Certificate.lean` (482 lines)

```lean
import Mathlib

/-!
# O4: the certificate score does not determine one-step progress

Setting: square real matrices `Matrix n n ℝ`, Frobenius inner product `⟪A, B⟫ = ∑ᵢⱼ Aᵢⱼ Bᵢⱼ`,
quadratic loss `f(W) = ½ ⟪W, H W⟫` with a Frobenius-self-adjoint linear operator `H`.
Its gradient is `G = H W`, and the step `W ↦ W - η d` decreases `f` by
`η ⟪G, d⟫ - η²/2 ⟪d, H d⟫`.

* `spec A` is Mathlib's $\ell^2 \to \ell^2$ operator norm (the spectral norm, `Matrix.Norms.L2Operator`).
* `nuc A` is the nuclear norm, defined as the sum of the singular values
  `∑ᵢ √λᵢ(Aᵀ A)` using Mathlib's eigenvalues of the Hermitian matrix `Aᴴ A`.
* `kappa G d = ⟪G, d⟫ / (nuc G * spec d)` is the certificate.
-/

open Matrix
open scoped Matrix.Norms.L2Operator

set_option linter.unusedSectionVars false

namespace O4

variable {n : Type*} [Fintype n] [DecidableEq n]

/-- Frobenius inner product `⟪A, B⟫ = ∑ᵢⱼ Aᵢⱼ Bᵢⱼ = tr(Aᵀ B)`. -/
def frob (A B : Matrix n n ℝ) : ℝ := ∑ i, ∑ j, A i j * B i j

/-- Spectral norm: Mathlib's `ℓ² → ℓ²` operator norm of the matrix. -/
noncomputable def spec (A : Matrix n n ℝ) : ℝ := ‖A‖

/-- Nuclear norm: the sum of the singular values, `∑ᵢ √(eigenvalueᵢ (Aᴴ A))`. -/
noncomputable def nuc (A : Matrix n n ℝ) : ℝ :=
  ∑ i, Real.sqrt ((isHermitian_conjTranspose_mul_self A).eigenvalues i)

/-- The certificate `κ(G, d) = ⟪G, d⟫ / (‖G‖_* ‖d‖_op)`. -/
noncomputable def kappa (G d : Matrix n n ℝ) : ℝ := frob G d / (nuc G * spec d)

/-- Stable rank `‖d‖_F² / ‖d‖_op²`. -/
noncomputable def srank (d : Matrix n n ℝ) : ℝ := frob d d / spec d ^ 2

/-- Quadratic loss `f(W) = ½ ⟪W, H W⟫`. -/
noncomputable def loss (H : Matrix n n ℝ →ₗ[ℝ] Matrix n n ℝ) (W : Matrix n n ℝ) : ℝ :=
  frob W (H W) / 2

/-- `H` is self-adjoint for the Frobenius inner product. -/
def SelfAdj (H : Matrix n n ℝ →ₗ[ℝ] Matrix n n ℝ) : Prop := ∀ A B, frob A (H B) = frob (H A) B

/-- The set of one-step decreases `f(W) - f(W - η d)`, `η ∈ ℝ`. Its greatest element is the
"best one-step decrease". -/
def decreases (H : Matrix n n ℝ →ₗ[ℝ] Matrix n n ℝ) (W d : Matrix n n ℝ) : Set ℝ :=
  Set.range fun η : ℝ => loss H W - loss H (W - η • d)

/-! ### Bilinearity of `frob` -/

lemma frob_comm (A B : Matrix n n ℝ) : frob A B = frob B A := by
  simp [frob, mul_comm]

lemma frob_sub_left (A B C : Matrix n n ℝ) : frob (A - B) C = frob A C - frob B C := by
  simp [frob, sub_mul, Finset.sum_sub_distrib]

lemma frob_sub_right (A B C : Matrix n n ℝ) : frob A (B - C) = frob A B - frob A C := by
  simp [frob, mul_sub, Finset.sum_sub_distrib]

lemma frob_add_left (A B C : Matrix n n ℝ) : frob (A + B) C = frob A C + frob B C := by
  simp [frob, add_mul, Finset.sum_add_distrib]

lemma frob_add_right (A B C : Matrix n n ℝ) : frob A (B + C) = frob A B + frob A C := by
  simp [frob, mul_add, Finset.sum_add_distrib]

lemma frob_smul_left (c : ℝ) (A B : Matrix n n ℝ) : frob (c • A) B = c * frob A B := by
  simp [frob, Finset.mul_sum, mul_assoc]

lemma frob_smul_right (c : ℝ) (A B : Matrix n n ℝ) : frob A (c • B) = c * frob A B := by
  simp [frob, Finset.mul_sum, mul_left_comm]

```

```
lemma frob_self_nonneg (A : Matrix n n ℝ) : 0 ≤ frob A A :=
  Finset.sum_nonneg fun i _ => Finset.sum_nonneg fun j _ => mul_self_nonneg (A i j)

/-! ### General quadratic: exact one-step decrease -/

/-- Exact expansion: `f(W - η d) = f(W) - η ⟪H W, d⟫ + η^2/2 ⟪d, H d⟫`, so `G = H W` is the
gradient and `⟪d, H d⟫` is the curvature along `d`. -/
theorem loss_step (H : Matrix n n ℝ →ₗ[ℝ] Matrix n n ℝ) (hH : SelfAdj H) (W d : Matrix n n ℝ)
    (η : ℝ) :
    loss H (W - η • d) = loss H W - η * frob (H W) d + η ^ 2 / 2 * frob d (H d) := by
  have h1 : frob W (H d) = frob (H W) d := hH W d
  have h2 : frob d (H W) = frob (H W) d := frob_comm _ _
  simp only [loss, map_sub, map_smul, frob_sub_left, frob_sub_right, frob_smul_left,
    frob_smul_right, h1, h2]
  ring

/-- `G = H W` is the gradient of `f` at `W`: the directional derivative of `f` at `W`
along any `d` is `⟪H W, d⟫`. -/
theorem loss_hasDerivAt (H : Matrix n n ℝ →ₗ[ℝ] Matrix n n ℝ) (hH : SelfAdj H)
    (W d : Matrix n n ℝ) :
    HasDerivAt (fun t : ℝ => loss H (W + t • d)) (frob (H W) d) 0 := by
  have e : (fun t : ℝ => loss H (W + t • d)) =
      fun t => loss H W + t * frob (H W) d + t ^ 2 / 2 * frob d (H d) := by
    funext t
    have := loss_step H hH W d (-t)
    rw [show W - (-t) • d = W + t • d by rw [neg_smul, sub_neg_eq_add]] at this
    rw [this]; ring
  rw [e]
  have h1 : HasDerivAt (fun t : ℝ => loss H W + t * frob (H W) d) (frob (H W) d) 0 := by
    convert ((hasDerivAt_id' (0 : ℝ)).mul_const (frob (H W) d)).const_add (loss H W) using 1
    simp
  have h2 : HasDerivAt (fun t : ℝ => t ^ 2 / 2 * frob d (H d)) 0 0 := by
    convert ((hasDerivAt_pow 2 (0 : ℝ)).div_const 2).mul_const (frob d (H d)) using 1
    simp
  convert h1.add h2 using 1
  simp

/-- **Best one-step decrease on a quadratic.** If the curvature along `d` is positive, the
largest decrease `f(W) - f(W - η d)` over all step sizes `η` is exactly
`⟪G, d⟫^2 / (2 ⟪d, H d⟫)` with `G = H W`, attained at `η = ⟪G, d⟫ / ⟪d, H d⟫`. -/
theorem best_step (H : Matrix n n ℝ →ₗ[ℝ] Matrix n n ℝ) (hH : SelfAdj H) (W d : Matrix n n ℝ)
    (hq : 0 < frob d (H d)) :
    IsGreatest (decreases H W d) (frob (H W) d ^ 2 / (2 * frob d (H d))) := by
  refine ⟨⟨frob (H W) d / frob d (H d), ?_⟩, ?_⟩
  · simp only [loss_step H hH]
    field_simp
    ring
  · rintro _ ⟨η, rfl⟩
    simp only [loss_step H hH]
    rw [le_div_iff₀ (by positivity)]
    nlinarith [sq_nonneg (frob d (H d) * η - frob (H W) d)]

/-- Same, restricted to nonnegative step sizes (when `⟪G, d⟫ ≥ 0`, i.e. `d` is a descent
direction): the optimum is unchanged. -/
theorem best_step_nonneg (H : Matrix n n ℝ →ₗ[ℝ] Matrix n n ℝ) (hH : SelfAdj H)
    (W d : Matrix n n ℝ) (hq : 0 < frob d (H d)) (hg : 0 ≤ frob (H W) d) :
    IsGreatest ((fun η : ℝ => loss H W - loss H (W - η • d)) '' Set.Ici 0)
      (frob (H W) d ^ 2 / (2 * frob d (H d))) := by
  obtain ⟨⟨η, hη⟩, hub⟩ := best_step H hH W d hq
  refine ⟨⟨frob (H W) d / frob d (H d), Set.mem_Ici.2 (div_nonneg hg hq.le), ?_⟩, ?_⟩
  · simp only [loss_step H hH]
    field_simp
    ring
  · rintro _ ⟨η', -, rfl⟩
    exact hub ⟨η', rfl⟩

/-! ### What governs the decrease: the (H-weighted) stable rank -/

/-- **Exact characterization, general `H`.** The best decrease equals
`κ^2 ‖G‖_*^2 / (2 ρ_H(d))` where `ρ_H(d) = ⟪d, H d⟫ / ‖d‖_op^2` is the `H`-weighted stable rank.
Pure algebra: holds for any positive normalisation `nuc G`. -/
theorem decrease_eq_kappa_general (H : Matrix n n ℝ →ₗ[ℝ] Matrix n n ℝ) (G d : Matrix n n ℝ)
    (hN : nuc G ≠ 0) (hs : spec d ≠ 0) (hq : frob d (H d) ≠ 0) :
    frob G d ^ 2 / (2 * frob d (H d)) =
      kappa G d ^ 2 * nuc G ^ 2 / (2 * (frob d (H d) / spec d ^ 2)) := by
  unfold kappa
  field_simp

/-- **Exact characterization, `H = id` (f = ½‖W‖_F^2).** The best decrease is
`κ^2 ‖G‖_*^2 / (2 · srank d)`: besides the certificate, it is governed by the stable rank
```

```
`‖d‖_F² / ‖d‖_op²` of the direction. -/
theorem decrease_eq_kappa_srank (G d : Matrix n n ℝ) (hN : nuc G ≠ 0) (hs : spec d ≠ 0)
    (hF : frob d d ≠ 0) :
    frob G d ^ 2 / (2 * frob d d) = kappa G d ^ 2 * nuc G ^ 2 / (2 * srank d) := by
  unfold kappa srank
  field_simp

/-- **Sandwich for general `H` with `m I ≤ H ≤ L I` along `d`.**
`κ²‖G‖_*² / (2 L srank d) ≤ best decrease ≤ κ²‖G‖_*² / (2 m srank d)`. -/
theorem decrease_sandwich (H : Matrix n n ℝ →ₗ[ℝ] Matrix n n ℝ) (G d : Matrix n n ℝ)
    (m L : ℝ) (hm : 0 < m) (hN : nuc G ≠ 0) (hs : spec d ≠ 0) (hF : 0 < frob d d)
    (hlo : m * frob d d ≤ frob d (H d)) (hhi : frob d (H d) ≤ L * frob d d) :
    kappa G d ^ 2 * nuc G ^ 2 / (2 * L * srank d) ≤ frob G d ^ 2 / (2 * frob d (H d)) ∧
      frob G d ^ 2 / (2 * frob d (H d)) ≤ kappa G d ^ 2 * nuc G ^ 2 / (2 * m * srank d) := by
  have hq : 0 < frob d (H d) := lt_of_lt_of_le (mul_pos hm hF) hlo
  have hL : 0 < L := by nlinarith
  have hk : kappa G d ^ 2 * nuc G ^ 2 = frob G d ^ 2 / spec d ^ 2 := by
    unfold kappa; field_simp
  have hsr : srank d = frob d d / spec d ^ 2 := rfl
  have hs2 : 0 < spec d ^ 2 := by positivity
  rw [hk, hsr]
  constructor
  · rw [div_le_div_iff₀ (by positivity) (by positivity)]
    have : frob G d ^ 2 / spec d ^ 2 * (2 * frob d (H d)) ≤
        frob G d ^ 2 / spec d ^ 2 * (2 * (L * frob d d)) := by gcongr
    calc frob G d ^ 2 / spec d ^ 2 * (2 * frob d (H d))
        ≤ frob G d ^ 2 / spec d ^ 2 * (2 * (L * frob d d)) := this
      _ = frob G d ^ 2 * (2 * L * (frob d d / spec d ^ 2)) := by field_simp
  · rw [div_le_div_iff₀ (by positivity) (by positivity)]
    have : frob G d ^ 2 * (2 * (m * frob d d)) ≤ frob G d ^ 2 * (2 * frob d (H d)) := by
      gcongr
    calc frob G d ^ 2 * (2 * m * (frob d d / spec d ^ 2))
        = frob G d ^ 2 / spec d ^ 2 * (2 * (m * frob d d)) := by field_simp
      _ ≤ frob G d ^ 2 / spec d ^ 2 * (2 * frob d (H d)) := by
        gcongr

/-- **κ and the stable rank together determine the decrease (`H = id`), and with κ fixed the
decrease pins down the stable rank.** For two directions with the same nonzero certificate
`κ(G,d₁) = κ(G,d₂)`, their best one-step decreases are equal iff their stable ranks are. -/
theorem decrease_eq_iff_srank_eq (G d₁ d₂ : Matrix n n ℝ) (hN : nuc G ≠ 0)
    (hs₁ : spec d₁ ≠ 0) (hs₂ : spec d₂ ≠ 0) (hF₁ : frob d₁ d₁ ≠ 0) (hF₂ : frob d₂ d₂ ≠ 0)
    (hk : kappa G d₁ = kappa G d₂) (hk0 : kappa G d₁ ≠ 0) :
    frob G d₁ ^ 2 / (2 * frob d₁ d₁) = frob G d₂ ^ 2 / (2 * frob d₂ d₂) ↔
      srank d₁ = srank d₂ := by
  rw [decrease_eq_kappa_srank G d₁ hN hs₁ hF₁, decrease_eq_kappa_srank G d₂ hN hs₂ hF₂, ← hk]
  have hsr₁ : srank d₁ ≠ 0 := div_ne_zero hF₁ (pow_ne_zero 2 hs₁)
  have hsr₂ : srank d₂ ≠ 0 := div_ne_zero hF₂ (pow_ne_zero 2 hs₂)
  have hc : kappa G d₁ ^ 2 * nuc G ^ 2 ≠ 0 := by positivity
  constructor
  · intro h
    field_simp at h
    exact h.symm
  · intro h; rw [h]

/-- **Strict separation for any well-conditioned quadratic.** If `m ‖d‖_F² ≤ ⟪d, H d⟫ ≤ L ‖d‖_F²`
along both directions, the certificates are equal and nonzero, and the stable ranks differ by
more than the condition number (`L · srank d₁ < m · srank d₂`), then `d₁` gives a strictly
larger best one-step decrease than `d₂`. -/
theorem strict_gap (H : Matrix n n ℝ →ₗ[ℝ] Matrix n n ℝ) (G d₁ d₂ : Matrix n n ℝ) (m L : ℝ)
    (hm : 0 < m) (hN : nuc G ≠ 0) (hs₁ : spec d₁ ≠ 0) (hs₂ : spec d₂ ≠ 0)
    (hF₁ : 0 < frob d₁ d₁) (hF₂ : 0 < frob d₂ d₂)
    (hlo₁ : m * frob d₁ d₁ ≤ frob d₁ (H d₁)) (hhi₁ : frob d₁ (H d₁) ≤ L * frob d₁ d₁)
    (hlo₂ : m * frob d₂ d₂ ≤ frob d₂ (H d₂)) (hhi₂ : frob d₂ (H d₂) ≤ L * frob d₂ d₂)
    (hk : kappa G d₁ = kappa G d₂) (hk0 : kappa G d₁ ≠ 0)
    (hsr : L * srank d₁ < m * srank d₂) :
    frob G d₂ ^ 2 / (2 * frob d₂ (H d₂)) < frob G d₁ ^ 2 / (2 * frob d₁ (H d₁)) := by
  obtain ⟨lo₁, -⟩ := decrease_sandwich H G d₁ m L hm hN hs₁ hF₁ hlo₁ hhi₁
  obtain ⟨-, hi₂⟩ := decrease_sandwich H G d₂ m L hm hN hs₂ hF₂ hlo₂ hhi₂
  have hsr₁ : 0 < srank d₁ := div_pos hF₁ (by positivity)
  have hL : 0 < L := by nlinarith
  have hc : 0 < kappa G d₁ ^ 2 * nuc G ^ 2 := by positivity
  rw [← hk] at hi₂
  calc frob G d₂ ^ 2 / (2 * frob d₂ (H d₂))
      ≤ kappa G d₁ ^ 2 * nuc G ^ 2 / (2 * m * srank d₂) := hi₂
    _ < kappa G d₁ ^ 2 * nuc G ^ 2 / (2 * L * srank d₁) := by
      apply div_lt_div_of_pos_left hc (by positivity)
      nlinarith
    _ ≤ frob G d₁ ^ 2 / (2 * frob d₁ (H d₁)) := lo₁

/-! ### Norm computations -/

```

```
/-- For an orthogonal projection (`Pᴴ P = P`) all singular values are `0` or `1`, so the
nuclear norm is the trace. -/
lemma nuc_of_proj (P : Matrix n n ℝ) (hP : Pᴴ * P = P) : nuc P = P.trace := by
  have hA : (Pᴴ * P).IsHermitian := isHermitian_conjTranspose_mul_self P
  have h1 : Pᴴ * P = Pᴴ := by
    have := congrArg conjTranspose hP
    rwa [conjTranspose_mul, conjTranspose_conjTranspose] at this
  have hsym : Pᴴ = P := h1.symm.trans hP
  have hidem : (Pᴴ * P) * (Pᴴ * P) = Pᴴ * P := by
    rw [hP]; nth_rewrite 1 [← hsym]; exact hP
  have hev : ∀ i, hA.eigenvalues i * hA.eigenvalues i = hA.eigenvalues i := by
    intro i
    set v : n → ℝ := ⇑(hA.eigenvectorBasis i)
    have hv : (Pᴴ * P) *ᵥ v = hA.eigenvalues i • v := hA.mulVec_eigenvectorBasis i
    have hv0 : v ≠ 0 :=
      (WithLp.ofLp_eq_zero 2).ne.2 <| hA.eigenvectorBasis.orthonormal.ne_zero i
    have h2 : (Pᴴ * P) *ᵥ ((Pᴴ * P) *ᵥ v) = (Pᴴ * P) *ᵥ v := by
      rw [mulVec_mulVec, hidem]
    rw [hv, mulVec_smul, hv, smul_smul] at h2
    have h3 : (hA.eigenvalues i * hA.eigenvalues i - hA.eigenvalues i) • v = 0 := by
      rw [sub_smul, h2, sub_self]
    rcases smul_eq_zero.1 h3 with h | h
    · linarith
    · exact absurd h hv0
  have hsqrt : ∀ i, Real.sqrt (hA.eigenvalues i) = hA.eigenvalues i := by
    intro i
    have h := hev i
    have : hA.eigenvalues i = 0 ∨ hA.eigenvalues i = 1 := by
      have : hA.eigenvalues i * (hA.eigenvalues i - 1) = 0 := by linarith
      rcases mul_eq_zero.1 this with h | h
      · exact Or.inl h
      · exact Or.inr (by linarith)
    rcases this with h | h <;> rw [h] <;> simp
  calc nuc P = ∑ i, Real.sqrt (hA.eigenvalues i) := rfl
    _ = ∑ i, hA.eigenvalues i := Finset.sum_congr rfl fun i _ => hsqrt i
    _ = (Pᴴ * P).trace := by rw [hA.trace_eq_sum_eigenvalues]; simp
    _ = P.trace := by rw [hP]

/-- Every entry is bounded by the spectral norm: `|D_{ij}| = |e_iᵀ D e_j| ≤ ‖D‖_op`. -/
lemma entry_le_spec (D : Matrix n n ℝ) (i j : n) : |D i j| ≤ spec D := by
  have h := l2_opNorm_mulVec D (EuclideanSpace.single j (1 : ℝ))
  have h2 := PiLp.norm_apply_le
    ((EuclideanSpace.equiv n ℝ).symm (D *ᵥ (EuclideanSpace.single j (1 : ℝ)))) i
  simp at h h2
  exact h2.trans h

lemma norm_fin2 (a b : ℝ) : ‖(![a, b] : Fin 2 → ℝ)‖ = max |a| |b| := by
  apply le_antisymm
  · refine (pi_norm_le_iff_of_nonneg (le_max_of_le_left (abs_nonneg a))).2 ?_
    intro i; fin_cases i <;> simp
  · refine max_le ?_ ?_
    · simpa using norm_le_pi_norm (![a, b] : Fin 2 → ℝ) 0
    · simpa using norm_le_pi_norm (![a, b] : Fin 2 → ℝ) 1

lemma frob_single_left (i₀ j₀ : n) (c : ℝ) (D : Matrix n n ℝ) :
    frob (single i₀ j₀ c) D = c * D i₀ j₀ := by
  simp [frob, single_apply, ite_mul, Finset.sum_ite_eq, ite_and]

lemma frob_one_one : frob (1 : Matrix n n ℝ) 1 = Fintype.card n := by
  simp [frob, one_apply]

lemma selfAdj_id : SelfAdj (LinearMap.id : Matrix n n ℝ →ₗ[ℝ] Matrix n n ℝ) := fun _ _ => rfl

/-- Best decrease for `f(W) = ½‖W‖_F²` (`H = id`, gradient `G = W`). -/
lemma best_step_id (W d : Matrix n n ℝ) (hF : 0 < frob d d) :
    IsGreatest (decreases LinearMap.id W d) (frob W d ^ 2 / (2 * frob d d)) := by
  simpa using best_step LinearMap.id selfAdj_id W d (by simpa using hF)

/-! ### Counterexample 1 (2×2, maximal certificate κ = 1)

`f(W) = ½‖W‖_F²` at `W = G₀ = diag(1,0)` (gradient `G₀`). Both `d₁ = diag(1,0)` and `d₂ = I`
have spectral norm 1 and maximise `⟪G₀, D⟫` over the spectral unit ball (both are exact
spectral-norm steepest-descent / LMO directions), so `κ = 1` for both; yet the best one-step
decreases are `1/2` and `1/4`. -/

/-- `G₀ = diag(1, 0)`: the weight matrix, and also the gradient of `½‖W‖_F²` there. -/
def G₀ : Matrix (Fin 2) (Fin 2) ℝ := !![1, 0; 0, 0]
/-- `d₁ = diag(1, 0)`, stable rank 1. -/
def d₁ : Matrix (Fin 2) (Fin 2) ℝ := !![1, 0; 0, 0]
```

```
/-- `d₂ = I`, stable rank 2. -/
def d₂ : Matrix (Fin 2) (Fin 2) ℝ := 1

lemma G₀_eq_diag : G₀ = diagonal ![1, 0] := by
  ext i j; fin_cases i <;> fin_cases j <;> simp [G₀]

lemma nuc_G₀ : nuc G₀ = 1 := by
  rw [nuc_of_proj]
  · simp [G₀, trace_fin_two]
  · ext i j; fin_cases i <;> fin_cases j <;> simp [G₀, Matrix.mul_apply, Fin.sum_univ_two]

lemma spec_d₁ : spec d₁ = 1 := by
  have : d₁ = G₀ := rfl
  rw [this, spec, G₀_eq_diag, l2_opNorm_diagonal, norm_fin2]; norm_num

lemma spec_d₂ : spec d₂ = 1 := norm_one

theorem counterexample_2x2 :
    kappa G₀ d₁ = 1 ∧ kappa G₀ d₂ = 1 ∧
    spec d₁ = 1 ∧ spec d₂ = 1 ∧
    (∀ D : Matrix (Fin 2) (Fin 2) ℝ, spec D ≤ 1 → frob G₀ D ≤ frob G₀ d₁) ∧
    frob G₀ d₁ = frob G₀ d₂ ∧
    (LinearMap.id : Matrix (Fin 2) (Fin 2) ℝ →ₗ[ℝ] Matrix (Fin 2) (Fin 2) ℝ) G₀ = G₀ ∧
    IsGreatest (decreases LinearMap.id G₀ d₁) (1 / 2) ∧
    IsGreatest (decreases LinearMap.id G₀ d₂) (1 / 4) := by
  have f1 : frob G₀ d₁ = 1 := by simp [frob, G₀, d₁, Fin.sum_univ_two]
  have f2 : frob G₀ d₂ = 1 := by simp [frob, G₀, d₂, Fin.sum_univ_two]
  have f11 : frob d₁ d₁ = 1 := by simp [frob, d₁, Fin.sum_univ_two]
  have f22 : frob d₂ d₂ = 2 := by norm_num [frob, d₂, one_apply]
  refine ⟨?_, ?_, spec_d₁, spec_d₂, ?_, by rw [f1, f2], rfl, ?_, ?_⟩
  · simp [kappa, f1, nuc_G₀, spec_d₁]
  · simp [kappa, f2, nuc_G₀, spec_d₂]
  · intro D hD
    have : frob G₀ D = D 0 0 := by simp [frob, G₀, Fin.sum_univ_two]
    rw [this, f1]
    exact (le_abs_self _).trans ((entry_le_spec D 0 0).trans hD)
  · have := best_step_id G₀ d₁ (by rw [f11]; norm_num)
    rw [f1, f11] at this
    convert this using 1; norm_num
  · have := best_step_id G₀ d₂ (by rw [f22]; norm_num)
    rw [f2, f22] at this
    convert this using 1; norm_num

/-- **The claim, formalised, is false.** "For every Frobenius-self-adjoint `H`, every `W`, and
directions `d₁, d₂` with positive curvature, equal certificates `κ(H W, d₁) = κ(H W, d₂)` force
equal best one-step decreases." -/
theorem kappa_does_not_determine_decrease :
    ¬ ∀ (H : Matrix (Fin 2) (Fin 2) ℝ →ₗ[ℝ] Matrix (Fin 2) (Fin 2) ℝ), SelfAdj H →
      ∀ (W d d' : Matrix (Fin 2) (Fin 2) ℝ) (D D' : ℝ),
        0 < frob d (H d) → 0 < frob d' (H d') →
        kappa (H W) d = kappa (H W) d' →
        IsGreatest (decreases H W d) D → IsGreatest (decreases H W d') D' → D = D' := by
  intro h
  obtain ⟨k1, k2, -, -, -, -, -, g1, g2⟩ := counterexample_2x2
  have := h LinearMap.id selfAdj_id G₀ d₁ d₂ _ _
    (by simp [frob, d₁, Fin.sum_univ_two]) (by simp [frob, d₂, one_apply])
    (by simpa using k1.trans k2.symm) g1 g2
  norm_num at this

/-! ### Counterexample 2 (n×n family: the gap is a factor `n`) -/

`G = e_{i₀} e_{i₀}ᵀ`, `d₁ = G` (stable rank 1), `d₂ = I_n` (stable rank n): both κ = 1, both are
LMO directions, best decreases `1/2` and `1/(2n)`. So κ does not even determine the decrease
up to a dimension-free constant. -/

theorem counterexample_family [Nonempty n] (i₀ : n) :
    kappa (single i₀ i₀ (1 : ℝ)) (single i₀ i₀ 1) = 1 ∧
    kappa (single i₀ i₀ (1 : ℝ)) 1 = 1 ∧
    (∀ D : Matrix n n ℝ, spec D ≤ 1 → frob (single i₀ i₀ (1 : ℝ)) D ≤ 1) ∧
    IsGreatest (decreases LinearMap.id (single i₀ i₀ (1 : ℝ)) (single i₀ i₀ 1)) (1 / 2) ∧
    IsGreatest (decreases LinearMap.id (single i₀ i₀ (1 : ℝ)) 1)
      (1 / (2 * Fintype.card n)) := by
  have hnuc : nuc (single i₀ i₀ (1 : ℝ)) = 1 := by
    rw [nuc_of_proj, trace_single_eq_same]
    rw [conjTranspose_single, single_mul_single_same]; simp
  have hspec : spec (single i₀ i₀ (1 : ℝ)) = 1 := by
    have : (single i₀ i₀ (1 : ℝ)) = diagonal (Pi.single i₀ 1) := by
      ext i j
      rw [single_apply, diagonal_apply]
      by_cases h : i = j
      · subst h; by_cases h' : i = i₀
```

```
        · subst h'; simp
        · simp [h', Ne.symm h']
      · simp [h]; rintro rfl h2; exact h h2
    rw [spec, this, l2_opNorm_diagonal, Pi.norm_single]; simp
  have hspec1 : spec (1 : Matrix n n ℝ) = 1 := norm_one
  have f1 : frob (single i₀ i₀ (1 : ℝ)) (single i₀ i₀ 1) = 1 := by
    rw [frob_single_left]; simp
  have f2 : frob (single i₀ i₀ (1 : ℝ)) 1 = 1 := by
    rw [frob_single_left]; simp
  have hcard : (0 : ℝ) < Fintype.card n := by exact_mod_cast Fintype.card_pos
  refine ⟨?_, ?_, ?_, ?_, ?_⟩
  · simp [kappa, f1, hnuc, hspec]
  · simp [kappa, f2, hnuc, hspec1]
  · intro D hD
    rw [frob_single_left, one_mul]
    exact (le_abs_self _).trans ((entry_le_spec D i₀ i₀).trans hD)
  · have := best_step_id (single i₀ i₀ (1 : ℝ)) (single i₀ i₀ 1) (by rw [f1]; norm_num)
    rw [f1] at this
    convert this using 1; norm_num
  · have := best_step_id (single i₀ i₀ (1 : ℝ)) 1 (by rw [frob_one_one]; exact hcard)
    rw [f2, frob_one_one] at this
    convert this using 1; norm_num

/-! ### Counterexample 3 (full-rank gradient `G = I₂`, κ = 3/5)

Rules out "it is only a rank-deficient-gradient artefact": at `W = I` (gradient `I`, nuclear norm
2), `e₁ = diag(1, 1/5)` and `e₂ = [[3,4],[-4,3]]` (= 5 × rotation) both have κ = 3/5, with best
decreases `9/13` and `9/25`. -/

noncomputable def e₁ : Matrix (Fin 2) (Fin 2) ℝ := !![1, 0; 0, 1 / 5]
def e₂ : Matrix (Fin 2) (Fin 2) ℝ := !![3, 4; -4, 3]

lemma nuc_one2 : nuc (1 : Matrix (Fin 2) (Fin 2) ℝ) = 2 := by
  rw [nuc_of_proj]
  · simp
  · simp

lemma spec_e₁ : spec e₁ = 1 := by
  have : e₁ = diagonal ![1, 1 / 5] := by
    ext i j; fin_cases i <;> fin_cases j <;> simp [e₁]
  rw [spec, this, l2_opNorm_diagonal, norm_fin2]; norm_num

lemma spec_e₂ : spec e₂ = 5 := by
  have h := l2_opNorm_conjTranspose_mul_self e₂
  have : e₂ᴴ * e₂ = diagonal ![25, 25] := by
    ext i j; fin_cases i <;> fin_cases j <;>
      simp [e₂, Matrix.mul_apply, Fin.sum_univ_two] <;> norm_num
  rw [this, l2_opNorm_diagonal, norm_fin2] at h
  have h0 := norm_nonneg e₂
  norm_num at h
  rw [spec]; nlinarith

theorem counterexample_fullrank :
    kappa 1 e₁ = 3 / 5 ∧ kappa 1 e₂ = 3 / 5 ∧
    IsGreatest (decreases LinearMap.id 1 e₁) (9 / 13) ∧
    IsGreatest (decreases LinearMap.id 1 e₂) (9 / 25) := by
  have f1 : frob 1 e₁ = 6 / 5 := by
    simp [frob, e₁, Fin.sum_univ_two, one_apply]; norm_num
  have f2 : frob 1 e₂ = 6 := by
    simp [frob, e₂, Fin.sum_univ_two, one_apply]; norm_num
  have f11 : frob e₁ e₁ = 26 / 25 := by simp [frob, e₁, Fin.sum_univ_two]; norm_num
  have f22 : frob e₂ e₂ = 50 := by simp [frob, e₂, Fin.sum_univ_two]; norm_num
  refine ⟨?_, ?_, ?_, ?_⟩
  · rw [kappa, f1, nuc_one2, spec_e₁]; norm_num
  · rw [kappa, f2, nuc_one2, spec_e₂]; norm_num
  · have := best_step_id 1 e₁ (by rw [f11]; norm_num)
    rw [f1, f11] at this
    convert this using 1; norm_num
  · have := best_step_id 1 e₂ (by rw [f22]; norm_num)
    rw [f2, f22] at this
    convert this using 1; norm_num

end O4

#print axioms O4.best_step
#print axioms O4.loss_hasDerivAt
#print axioms O4.decrease_eq_kappa_srank
#print axioms O4.decrease_eq_kappa_general
#print axioms O4.decrease_sandwich
#print axioms O4.decrease_eq_iff_srank_eq
#print axioms O4.strict_gap
```

```
#print axioms O4.counterexample_2x2
#print axioms O4.kappa_does_not_determine_decrease
#print axioms O4.counterexample_family
#print axioms O4.counterexample_fullrank
```

Axioms

`Axioms.lean` (37 lines)

```
import FastMuon
open FastMuon
#print axioms FastMuon.Geometry.muon_family_rate
#print axioms FastMuon.Geometry.avg_bound
#print axioms FastMuon.Geometry.momentum_error
#print axioms FastMuon.Geometry.certificate_transfer
#print axioms FastMuon.HeadTail.gram_split
#print axioms FastMuon.HeadTail.htDir_norm_le_sqrt_two
#print axioms FastMuon.HeadTail.align
#print axioms FastMuon.HeadTail.align_nonneg
#print axioms FastMuon.HeadTail.head_isometry
#print axioms FastMuon.Cost.speedup
#print axioms FastMuon.Cost.speedup_unbounded
#print axioms FastMuon.Cost.speedup_4096_16
#print axioms FastMuon.NSPoly.nsP_iter_ge
#print axioms FastMuon.NSPoly.muon_five_steps
#print axioms FastMuon.Spectral.geometry
#print axioms FastMuon.Spectral.nuc_frame_le
#print axioms FastMuon.Spectral.ht_alignment_lower
#print axioms FastMuon.Spectral.ht_muon_rate
#print axioms FastMuon.MuonVsSVD.claimed_contraction_false
#print axioms FastMuon.MuonVsSVD.claimed_five_step_false
#print axioms FastMuon.MuonVsSVD.five_steps_from_one_not_close
#print axioms FastMuon.MuonVsSVD.claimed_efficiency_false
#print axioms FastMuon.MuonVsSVD.ns_vs_svd
#print axioms FastMuon.MuonVsSVD.ht_vs_svd
#print axioms FastMuon.Separation.ns5_spiked_kappa
#print axioms FastMuon.Separation.ht_spiked_kappa
#print axioms FastMuon.OptimalTail.certified_ratio_le
#print axioms FastMuon.OptimalTail.certified_ratio_opt
#print axioms FastMuon.NSMatrix.nsStep_norm_le
#print axioms FastMuon.NSMatrix.muon_norm_le
#print axioms FastMuon.MatrixFacts.norm_aeval_le
#print axioms FastMuon.MatrixFacts.l2_opNorm_le_frobNorm
#print axioms FastMuon.Polish.cubicStep_norm_le_one
#print axioms FastMuon.Polish.polished_ht_norm_le_one
#print axioms FastMuon.Polish.polished_ht_norm_le_one_robust
```

Comparator challenges

`ComparatorChallenges/Certificate.lean` (52 lines)

```
import Mathlib

/-!
# Comparator challenge: `FastMuon.Geometry.muon_family_rate` (`FastMuon/Certificate.lean`)

Imports only Mathlib.  The structures `Geometry` and `Geometry.Smooth` are copied verbatim from
`FastMuon/Certificate.lean`; `lake comparator` checks that they are identical constants in the
solution and that the solution proves exactly the statement below.
-/

namespace FastMuon

open Finset

/-- The geometry: a bilinear pairing and two seminorms related by Hölder's inequality. -/
structure Geometry (E : Type*) [AddCommGroup E] [Module ℝ E] where
  ip : E →ₗ[ℝ] E →ₗ[ℝ] ℝ
  N : Seminorm ℝ E
  D : Seminorm ℝ E
  holder : ∀ x y, |ip x y| ≤ D x * N y

variable {E : Type*} [AddCommGroup E] [Module ℝ E]

namespace Geometry

variable (G : Geometry E)

```

```
/-- An `L`-smooth objective in the geometry `(N, D)`: the descent-lemma upper bound and a
`D`-Lipschitz gradient. -/
structure Smooth (f : E → ℝ) (grad : E → E) (L : ℝ) : Prop where
  L_nonneg : 0 ≤ L
  upper : ∀ x y, f y ≤ f x + G.ip (grad x) (y - x) + L / 2 * G.N (y - x) ^ 2
  lip : ∀ x y, G.D (grad y - grad x) ≤ L * G.N (y - x)

/-- **Main theorem (Muon family with momentum).**
`avg_t D(∇f(x t)) ≤ (f x₀ - f_low)/(α η T) + L η β²/(2α) + (α+β)/α · (μ L η β/(1-μ) + σ)`. -/
theorem muon_family_rate {f : E → ℝ} {grad : E → E} {L : ℝ} (hf : G.Smooth f grad L)
    {x d m ghat : ℕ → E} {η α β μ σ flow : ℝ} (hη : 0 < η) (hα : 0 < α) (hβ : 0 ≤ β)
    (hμ0 : 0 ≤ μ) (hμ1 : μ < 1)
    (hstep : ∀ t, x (t + 1) = x t - η • d t)
    (horacle : ∀ t, α * G.D (m t) ≤ G.ip (m t) (d t)) (hN : ∀ t, G.N (d t) ≤ β)
    (hm0 : m 0 = ghat 0) (hm : ∀ t, m (t + 1) = μ • m t + (1 - μ) • ghat (t + 1))
    (hnoise : ∀ t, G.D (ghat t - grad (x t)) ≤ σ)
    (hlow : ∀ y, flow ≤ f y) {T : ℕ} (hT : 0 < T) :
    (∑ t ∈ range T, G.D (grad (x t))) / T ≤
      (f (x 0) - flow) / (α * η * T) + L * η * β ^ 2 / (2 * α) +
        (α + β) / α * (μ * (L * η * β) / (1 - μ) + σ) := by
  sorry

end Geometry

end FastMuon
```

`ComparatorChallenges/Cost.lean` (27 lines)

```
import Mathlib

/-!
# Comparator challenge: `FastMuon.Cost.speedup`, `FastMuon.Cost.speedup_4096_16` (`FastMuon/Cost.lean`)

Imports only Mathlib.  The flop-count functions `nsCost` and `htCost` are copied verbatim from
`FastMuon/Cost.lean`.
-/

namespace FastMuon.Cost

/-- Flops of `k` Newton–Schulz quintic steps on an `s × l` matrix (`s ≤ l`). -/
def nsCost (s l k : ℕ) : ℕ := k * (4 * s ^ 2 * l + 2 * s ^ 3)

/-- Flops of one head–tail call with rank `r`. -/
def htCost (s l r : ℕ) : ℕ := 6 * s * l * r + 6 * (s + l) * r ^ 2 + 20 * r ^ 3 + 10 * s * l

/-- **Speed-up guarantee.** For `1 ≤ r ≤ s ≤ l`: `5 s · htCost ≤ 12 r · nsCost₅`. -/
theorem speedup (s l r : ℕ) (hr : 1 ≤ r) (hrs : r ≤ s) (hsl : s ≤ l) :
    5 * s * htCost s l r ≤ 12 * r * nsCost s l 5 := by
  sorry

/-- Concrete instance: a 4096 × 4096 layer, rank 16 — more than 1000× fewer flops. -/
theorem speedup_4096_16 : 1000 * htCost 4096 4096 16 ≤ nsCost 4096 4096 5 := by
  sorry

end FastMuon.Cost
```

`ComparatorChallenges/HeadTail.lean` (34 lines)

```
import Mathlib

/-!
# Comparator challenge: `FastMuon.HeadTail.htDir_norm_le_sqrt_two` (`FastMuon/HeadTail.lean`)

Imports only Mathlib.  `tail` and `htDir` are copied verbatim from `FastMuon/HeadTail.lean`.
The norm is Mathlib's spectral norm (`L2` operator norm, `open scoped Matrix.Norms.L2Operator`).
-/

set_option linter.unusedSectionVars false

namespace FastMuon.HeadTail

open Matrix
open scoped Matrix.Norms.L2Operator

variable {s l r : Type*} [Fintype s] [Fintype l] [Fintype r]
  [DecidableEq s] [DecidableEq l] [DecidableEq r]

/-- The tail of `X` outside the column span of `U`. -/
def tail (U : Matrix s r ℝ) (X : Matrix s l ℝ) : Matrix s l ℝ := X - U * (Uᵀ * X)

```

```
/-- The head-tail direction `U P + τ⁻¹ • tail U X`. -/
noncomputable def htDir (U : Matrix s r ℝ) (P : Matrix r l ℝ) (τ : ℝ) (X : Matrix s l ℝ) :
    Matrix s l ℝ :=
  U * P + τ⁻¹ • tail U X

/-- **`β ≤ √2`.** If `Uᵀ U = 1`, `‖P‖ ≤ 1` and `‖T‖ ≤ τ`, then `‖O‖ ≤ √2`. -/
theorem htDir_norm_le_sqrt_two {U : Matrix s r ℝ} (hU : Uᵀ * U = 1) {P : Matrix r l ℝ} {τ : ℝ}
    {X : Matrix s l ℝ} (hP : ‖P‖ ≤ 1) (hτ : 0 < τ) (hT : ‖tail U X‖ ≤ τ) :
    ‖htDir U P τ X‖ ≤ Real.sqrt 2 := by
  sorry

end FastMuon.HeadTail
```

ComparatorChallenges/MuonVsSVD.lean (42 lines)

```
import Mathlib

/-!
# Comparator challenge: `FastMuon.MuonVsSVD.claimed_contraction_false`, `FastMuon.MuonVsSVD.ns_vs_svd`
(`FastMuon/MuonVsSVD.lean`)

Imports only Mathlib.  `NSPoly.nsP`, `Cost.nsCost` and `MuonVsSVD.svdCost` are copied verbatim from
`FastMuon/NSPoly.lean`, `FastMuon/Cost.lean` and `FastMuon/MuonVsSVD.lean`.
-/

namespace FastMuon.NSPoly

/-- Muon's quintic. -/
noncomputable def nsP (x : ℝ) : ℝ := 3.4445 * x - 4.7750 * x ^ 3 + 2.0315 * x ^ 5

end FastMuon.NSPoly

namespace FastMuon.Cost

/-- Flops of `k` Newton-Schulz quintic steps on an `s × l` matrix (`s ≤ l`). -/
def nsCost (s l k : ℕ) : ℕ := k * (4 * s ^ 2 * l + 2 * s ^ 3)

end FastMuon.Cost

namespace FastMuon.MuonVsSVD

open FastMuon.NSPoly FastMuon.Cost

/-- One Newton-Schulz step does not contract the error on `[0.5, 1.5]`. -/
theorem claimed_contraction_false :
    ¬ ∀ σ : ℝ, 0.5 ≤ σ → σ ≤ 1.5 → |nsP σ - 1| < |σ - 1| := by
  sorry

/-- Golub-Reinsch SVD (`Σ` and `V`) of an `m × n` matrix, `m ≥ n`, as in the write-up. -/
def svdCost (m n : ℕ) : ℕ := 4 * m * n ^ 2 + 8 * n ^ 3

/-- **NS vs SVD in flops.** For `s ≤ l`, five NS steps need between 2.5× and 5× the flops of SVD. -/
theorem ns_vs_svd (s l : ℕ) (hsl : s ≤ l) :
    5 * svdCost l s ≤ 2 * nsCost s l 5 ∧ nsCost s l 5 ≤ 5 * svdCost l s := by
  sorry

end FastMuon.MuonVsSVD
```

ComparatorChallenges/NSMatrix.lean (46 lines)

```
import Mathlib

/-!
# Comparator challenge: `FastMuon.NSMatrix.muon_norm_le` (`FastMuon/NSMatrix.lean`)

Imports only Mathlib.  `MatrixFacts.frobNorm` and `NSMatrix.nsStep` are copied verbatim from
`FastMuon/MatrixFacts.lean` and `FastMuon/NSMatrix.lean`.  `‖·‖` is Mathlib's spectral norm
(`L2` operator norm).
-/

set_option linter.unusedSectionVars false

namespace FastMuon.MatrixFacts

open Matrix Polynomial

section Frobenius

open scoped Matrix.Norms.Frobenius
```

```

variable {s l : Type*} [Fintype s] [Fintype l]

/-- The Frobenius norm `‖M‖_F = √(∑ᵢⱼ Mᵢⱼ²)`, written out explicitly. -/
noncomputable def frobNorm (M : Matrix s l ℝ) : ℝ := Real.sqrt (∑ i, ∑ j, M i j ^ 2)

end Frobenius

end FastMuon.MatrixFacts

namespace FastMuon.NSMatrix

open Matrix Polynomial FastMuon.MatrixFacts
open scoped Matrix.Norms.L2Operator

variable {s l : Type*} [Fintype s] [Fintype l] [DecidableEq s] [DecidableEq l]

/-- One step of Muon's Newton–Schulz iteration. -/
noncomputable def nsStep (X : Matrix s l ℝ) : Matrix s l ℝ :=
  (3.4445 : ℝ) • X + (-4.7750 : ℝ) • (X * Xᵀ * X) + (2.0315 : ℝ) • (X * Xᵀ * X * Xᵀ * X)

/-- **Muon's output has spectral norm at most `1.25`.**  For every real `M`, five Newton–Schulz
steps applied to `X₀ = M / ‖M‖_F` give `‖nsStep^[5] X₀‖₂ ≤ 1.25`. -/
theorem muon_norm_le (M : Matrix s l ℝ) : ‖nsStep^[5] ((1 / frobNorm M) • M)‖ ≤ 1.25 := by
  sorry

end FastMuon.NSMatrix
```

ComparatorChallenges/Separation.lean (27 lines)

```
import Mathlib

/-!
# Comparator challenge: `FastMuon.Separation.ns5_spiked_kappa` (`FastMuon/Separation.lean`)

Imports only Mathlib.  `NSPoly.nsP` is copied verbatim from `FastMuon/NSPoly.lean`.
-/

namespace FastMuon.NSPoly

/-- Muon's quintic. -/
noncomputable def nsP (x : ℝ) : ℝ := 3.4445 * x - 4.7750 * x ^ 3 + 2.0315 * x ^ 5

end FastMuon.NSPoly

namespace FastMuon.Separation

open FastMuon.NSPoly

/-- **Muon on a spiked spectrum.** With `y = p^[5](1/F)`, `z = p^[5](ε/F)`, `ε = 4·10⁻⁴`, the
certificate quality `κ` is at most `0.57`. -/
theorem ns5_spiked_kappa (F : ℝ) (hF1 : 1 ≤ F) (hF2 : F ≤ 1 / 0.99) :
    (nsP^[5] (1 / F) + 4095 * 4e-4 * nsP^[5] (4e-4 / F)) /
        ((1 + 4095 * 4e-4) * max (nsP^[5] (1 / F)) (nsP^[5] (4e-4 / F))) ≤ 0.57 := by
  sorry

end FastMuon.Separation
```

ComparatorChallenges/Spectral.lean (127 lines)

```
import Mathlib

/-!
# Comparator challenge: `FastMuon.Spectral.ht_muon_rate` (`FastMuon/Spectral.lean`)

Imports only Mathlib.  Copied verbatim from the source (and checked constant-for-constant by
`lake comparator`, proofs inside definitions included):

* `Geometry`, `Geometry.Smooth` from `FastMuon/Certificate.lean`;
* `tail`, `htDir`, `frob` from `FastMuon/HeadTail.lean`;
* `frobBilin`, `frobCLM`, `nuc` (the dual of the spectral norm), `holder` (Hölder's inequality, used
  inside `geometry`), `geometry` (Muon's geometry on `Matrix s l ℝ`) and their `@[simp]` /
  `rfl` lemmas from `FastMuon/Spectral.lean`.
-/

namespace FastMuon

/-- The geometry: a bilinear pairing and two seminorms related by Hölder's inequality. -/
structure Geometry (E : Type*) [AddCommGroup E] [Module ℝ E] where
```

```
  ip : E →ₗ[ℝ] E →ₗ[ℝ] ℝ
  N : Seminorm ℝ E
  D : Seminorm ℝ E
  holder : ∀ x y, |ip x y| ≤ D x * N y

variable {E : Type*} [AddCommGroup E] [Module ℝ E]

namespace Geometry

variable (G : Geometry E)

/-- An `L`-smooth objective in the geometry `(N, D)`: the descent-lemma upper bound and a
`D`-Lipschitz gradient. -/
structure Smooth (f : E → ℝ) (grad : E → E) (L : ℝ) : Prop where
  L_nonneg : 0 ≤ L
  upper : ∀ x y, f y ≤ f x + G.ip (grad x) (y - x) + L / 2 * G.N (y - x) ^ 2
  lip : ∀ x y, G.D (grad y - grad x) ≤ L * G.N (y - x)

end Geometry

end FastMuon

set_option linter.unusedSectionVars false

namespace FastMuon.HeadTail

open Matrix
open scoped Matrix.Norms.L2Operator

variable {s l r : Type*} [Fintype s] [Fintype l] [Fintype r]
  [DecidableEq s] [DecidableEq l] [DecidableEq r]

/-- The tail of `X` outside the column span of `U`. -/
def tail (U : Matrix s r ℝ) (X : Matrix s l ℝ) : Matrix s l ℝ := X - U * (Uᵀ * X)

/-- The head–tail direction `U P + τ⁻¹ • tail U X`. -/
noncomputable def htDir (U : Matrix s r ℝ) (P : Matrix r l ℝ) (τ : ℝ) (X : Matrix s l ℝ) :
    Matrix s l ℝ :=
  U * P + τ⁻¹ • tail U X

/-- Frobenius pairing `⟪A, B⟫ = tr (Aᵀ B)`. -/
def frob (A B : Matrix s l ℝ) : ℝ := (Aᵀ * B).trace

end FastMuon.HeadTail

namespace FastMuon.Spectral

open Matrix FastMuon.HeadTail Finset
open scoped Matrix.Norms.L2Operator

variable {s l r : Type*} [Fintype s] [Fintype l] [Fintype r]
  [DecidableEq s] [DecidableEq l] [DecidableEq r]

/-- Frobenius pairing as a bilinear map. -/
def frobBilin : Matrix s l ℝ →ₗ[ℝ] Matrix s l ℝ →ₗ[ℝ] ℝ :=
  LinearMap.mk₂ ℝ (fun A B => frob A B)
    (fun A₁ A₂ B => by simp [frob, transpose_add, Matrix.add_mul, trace_add])
    (fun c A B => by simp [frob, transpose_smul, Matrix.smul_mul, trace_smul])
    (fun A B₁ B₂ => by simp [frob, Matrix.mul_add, trace_add])
    (fun c A B => by simp [frob, Matrix.mul_smul, trace_smul])

@[simp] theorem frobBilin_apply (A B : Matrix s l ℝ) : frobBilin A B = frob A B := rfl

/-- `G ↦ ⟪G, ·⟫` as a continuous functional on `(Matrix s l ℝ, ‖·‖₂)`. -/
noncomputable def frobCLM : Matrix s l ℝ →ₗ[ℝ] Matrix s l ℝ →L[ℝ] ℝ :=
  (LinearMap.toContinuousLinearMap : (Matrix s l ℝ →ₗ[ℝ] ℝ) ≃ₗ[ℝ] _).toLinearMap ∘ₗ frobBilin

@[simp] theorem frobCLM_apply (A B : Matrix s l ℝ) : frobCLM A B = frob A B := rfl

/-- The nuclear norm as the dual of the spectral norm. -/
noncomputable def nuc : Seminorm ℝ (Matrix s l ℝ) :=
  (normSeminorm ℝ (Matrix s l ℝ →L[ℝ] ℝ)).comp frobCLM

theorem nuc_apply (A : Matrix s l ℝ) : nuc A = ‖frobCLM A‖ := rfl

/-- Hölder / von Neumann: `|⟪A, B⟫| ≤ ‖A‖_* ‖B‖₂`. -/
theorem holder (A B : Matrix s l ℝ) : |frob A B| ≤ nuc A * ‖B‖ := by
  have := (frobCLM A).le_opNorm B
  simpa [nuc_apply, Real.norm_eq_abs] using this

/-- Muon's geometry on `s × l` real matrices. -/
```

```
noncomputable def geometry : Geometry (Matrix s l ℝ) where
  ip := frobBilin
  N := normSeminorm ℝ _
  D := nuc
  holder A B := by simpa [coe_normSeminorm] using holder A B

/-- **HT-Muon converges at Muon's rate.**  With `β = √2`,
`avg_t ‖∇f(x t)‖_* ≤ (f x₀ - f_low)/(α η T) + L η·2/(2α) + (α+√2)/α · (μ L η √2/(1-μ) + σ)`. -/
theorem ht_muon_rate {f : Matrix s l ℝ → ℝ} {grad : Matrix s l ℝ → Matrix s l ℝ} {L : ℝ}
    (hf : (geometry (s := s) (l := l)).Smooth f grad L)
    {x m ghat : ℕ → Matrix s l ℝ} {U : ℕ → Matrix s r ℝ} {P : ℕ → Matrix r l ℝ} {τ : ℕ → ℝ}
    {η α αh μ σ flow : ℝ} (hη : 0 < η) (hα : 0 < α) (hαh : α ≤ αh) (hμ0 : 0 ≤ μ) (hμ1 : μ < 1)
    (hU : ∀ t, (U t)ᵀ * U t = 1) (hP : ∀ t, ‖P t‖ ≤ 1)
    (hPc : ∀ t, αh * nuc ((U t)ᵀ * m t) ≤ frob ((U t)ᵀ * m t) (P t))
    (hτ : ∀ t, 0 < τ t) (hT : ∀ t, ‖tail (U t) (m t)‖ ≤ τ t)
    (hloss : ∀ t, αh * nuc (tail (U t) (m t)) - (τ t)⁻¹ * frob (tail (U t) (m t)) (tail (U t) (m t))
      ≤ (αh - α) * nuc (m t))
    (hstep : ∀ t, x (t + 1) = x t - η • htDir (U t) (P t) (τ t) (m t))
    (hm0 : m 0 = ghat 0) (hm : ∀ t, m (t + 1) = μ • m t + (1 - μ) • ghat (t + 1))
    (hnoise : ∀ t, nuc (ghat t - grad (x t)) ≤ σ)
    (hlow : ∀ y, flow ≤ f y) {T : ℕ} (hT0 : 0 < T) :
    (∑ t ∈ range T, nuc (grad (x t))) / T ≤
      (f (x 0) - flow) / (α * η * T) + L * η * Real.sqrt 2 ^ 2 / (2 * α) +
        (α + Real.sqrt 2) / α * (μ * (L * η * Real.sqrt 2) / (1 - μ) + σ) := by
  sorry

end FastMuon.Spectral
```