ZPOLES: FAST FRACTIONAL MATRIX POWERS FOR OPTIMIZATION VIA RATIONAL APPROXIMATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Fractional matrix powers underpin gradient preconditioning in Shampoo and momentum transforms in Muon, but their repeated computation can be expensive. Rational approximations use a small number of shifted matrix inverses, whose cost has limited their appeal for GPU training. We propose ZPoles, a method for computing fractional powers with a new mixed-precision linear solver. The solver keeps diagonal operations in higher precision and batches half-precision matrix multiplications across the shifted linear systems. On an RTX 5090 GPU, the solver is 3.7–4.1× faster than `cuSOLVER`, trading numerical accuracy for runtime. For matrices of size 1024×1024 to 12288×12288, ZPoles computes inverse fourth roots 2.6–8.1× faster than coupled Newton iteration at a fixed 10-iteration budget, with lower error when both use half-precision matrix-multiplication operands. In GPT-2 pretraining, ZPoles reduces observed Shampoo step time relative to coupled Newton, with lower or similar validation loss. Muon and fractional-power Muon provide further evidence of applicability to momentum transforms. These results support rational approximation as a practical computational primitive for GPU-based optimization.

1 INTRODUCTION

Fractional matrix powers let optimizers act on correlations between parameters. They precondition gradients using accumulated statistics or reshape the singular values of momentum. Full-matrix AdaGrad illustrates the first use by rescaling the gradient with an inverse square root (Duchi et al., 2011):

$$\mathbf{H}_t = \sum_{i=1}^{t} \mathbf{g}_i \mathbf{g}_i^\top + \epsilon \mathbf{I}, \qquad \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \mathbf{H}_t^{-1/2} \mathbf{g}_t. \tag{1}$$

Here $\boldsymbol{\theta}_t$ is the parameter vector, $\mathbf{g}_t$ its gradient, η_t the learning rate, $\epsilon > 0$ a damping parameter, and $\mathbf{I}$ the identity matrix. Capturing correlations across all parameters, however, requires a matrix too large for modern neural networks.

Shampoo makes this approach more manageable by accumulating smaller statistics along each tensor dimension (Gupta et al., 2018). For a matrix gradient $\mathbf{G}_t$, its two-sided preconditioning takes the form

$$\mathbf{L}_t = \sum_{i \le t} \mathbf{G}_i \mathbf{G}_i^\top + \epsilon \mathbf{I}, \quad \mathbf{R}_t = \sum_{i \le t} \mathbf{G}_i^\top \mathbf{G}_i + \epsilon \mathbf{I}, \quad \mathbf{W}_{t+1} = \mathbf{W}_t - \eta_t \mathbf{L}_t^{-1/4} \mathbf{G}_t \mathbf{R}_t^{-1/4}. \tag{2}$$

Muon uses matrix structure in a different way: it approximately orthogonalizes momentum using Newton–Schulz (NS) iterations (Jordan et al., 2024). The polar update of a full-row-rank momentum matrix $\mathbf{M} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\top$ is $\mathbf{U}\mathbf{V}^\top = (\mathbf{M}\mathbf{M}^\top)^{-1/2}\mathbf{M}$ (Bernstein & Newhouse, 2024). Fractional-power variants retain singular-value information through $\mathbf{U}\boldsymbol{\Sigma}^p\mathbf{V}^\top = (\mathbf{M}\mathbf{M}^\top)^{-(1-p)/2}\mathbf{M}$, where $0 < p < 1$ (Dong & Sawin, 2026; Shumaylov et al., 2026; Pang et al., 2026; Qi et al., 2026; Massena et al., 2026; Wu et al., 2026). DynMuon dynamically schedules p during training, improving pretraining performance over Muon (Wu et al., 2026). Such schedules can also use negative p, amplifying small singular directions rather than suppressing them. For $-1 < p < 1$, both regimes use a fractional power with $q = (p-1)/2$, which need not correspond to an integer root.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

The same primitive appears beyond optimizer updates. Neural network whitening uses an inverse covariance square root to normalize and decorrelate centered activations, which can improve optimization conditioning and accelerate convergence (Desjardins et al., 2015; Huang et al., 2019).

Computing $\mathbf{A}^q$ by eigendecomposition for $-1 < q < 0$ is expensive when $\mathbf{A} \succ 0$ changes every training step. Reusing preconditioners reduces this cost, but stale roots can slow convergence (Shi et al., 2023). Polynomial iterations such as coupled Newton use GPU-friendly matrix multiplications (Guo & Higham, 2006; Iannazzo, 2006), but resolving small eigenvalues can require many iterations. Rational approximation is a classical approach to computing matrix functions (Higham, 2008). It can capture rapid spectral variation near zero with few terms, providing accurate approximations over wide spectral intervals (van den Eshof et al., 2002; Hale et al., 2008). However, rational approximation requires computing matrix inverses or solving linear systems, operations considered too slow on GPUs for practical training (Amsel et al., 2026; Massena et al., 2026).

In this paper, **we challenge the view that rational approximation is impractical for training on GPUs.** We argue that its practical limitations stem largely from the lack of efficient GPU solvers for the resulting linear systems, rather than from rational approximation itself. General-purpose solvers such as `cuSOLVER` prioritize high numerical accuracy. Optimization in machine learning often tolerates lower numerical accuracy, allowing the main computation to use `fp16` or `bf16` operands with `fp32` accumulation (Micikevicius et al., 2018). Separate solver calls miss opportunities to batch operations across shifts and assemble only the required weighted inverse sum. Each shift still requires its own factorization. We introduce ZPoles to exploit these numerical and structural opportunities for fast fractional matrix powers.

Our contributions are threefold: (1) a mixed-precision solver that batches `fp16` matrix multiplications while retaining `fp32` diagonal operations and accumulation; (2) an existence bound for positive-shift approximants and bounds on solve and covariance-drift errors; and (3) matrix benchmarks and billion-token GPT-2 training experiments. Our solver is 3.7–$4.1\times$ faster than `cuSOLVER` (Table 1). Shampoo training has lower observed step cost with lower or comparable validation loss; Muon and fractional-power Muon test momentum transforms.

2 Computing Fractional Matrix Powers

For $\mathbf{A} \succ 0$, we seek $\mathbf{A}^q$ with $-1 < q < 0$. Inverse square and fourth roots correspond to $q = -1/2$ and $q = -1/4$; dynamic momentum transforms also require other exponents. Eigendecomposition, Newton iterations, and rational approximation provide three routes to these powers.

Eigendecomposition. If $\mathbf{A} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^\top$, then $\mathbf{A}^q = \mathbf{U}\mathbf{\Lambda}^q\mathbf{U}^\top$. Applying the power to each eigenvalue supports arbitrary exponents. Dense decomposition costs $O(n^3)$ and must be repeated when the input changes. We use higher-precision eigendecomposition as an accuracy reference.

Newton iterations. These methods replace decomposition with successive polynomial updates computed by matrix multiplications. Coupled Newton (CN) provides a construction for inverse integer roots (Guo & Higham, 2006; Iannazzo, 2006). For $q = -1/r$, with integer $r > 1$, initialize $\mathbf{X}_0 = z^{1/r}\mathbf{I}$ and $\mathbf{M}_0 = z\mathbf{A}$ using a positive scale z, then iterate

$$\mathbf{T}_t = ((r+1)\mathbf{I} - \mathbf{M}_t)/r, \qquad \mathbf{X}_{t+1} = \mathbf{X}_t\mathbf{T}_t, \qquad \mathbf{M}_{t+1} = \mathbf{T}_t^r\mathbf{M}_t. \tag{3}$$

The exact-arithmetic invariant $\mathbf{M}_t = \mathbf{A}\mathbf{X}_t^r$ connects the residual to the root: $\mathbf{M}_t \to \mathbf{I}$ implies $\mathbf{X}_t \to \mathbf{A}^{-1/r}$ under positive-spectrum initialization. Scaling the initial spectrum into $(0, 1]$ is sufficient for convergence. For $r = 4$, two successive squarings form $\mathbf{T}_t^4$, giving 4 matrix multiplications per iteration. Convergence is locally quadratic, but small initial eigenvalues can require many iterations. Anil et al. (2020) adapt this approach to training with damping, warm starts, and higher precision for ill-conditioned statistics.

Newton–Schulz (NS) is used for both inverse roots and polar factors, but the updates differ. Setting $r = 2$ in Equation (3) gives classical inverse-square-root NS with $\mathbf{T}_t = (3\mathbf{I} - \mathbf{M}_t)/2$. In contrast, Zhang et al. (2026) start from a Frobenius-normalized rectangular matrix $\mathbf{G}$, set $\mathbf{R}_0 = \mathbf{G}\mathbf{G}^\top$ and $\mathbf{Q}_0 = \mathbf{I}$, and use $\mathbf{Z}_t = a_t\mathbf{I} + b_t\mathbf{R}_{t-1} + c_t\mathbf{R}_{t-1}^2$, $\mathbf{Q}_t = \mathbf{Q}_{t-1}\mathbf{Z}_t$, and $\mathbf{R}_t = \mathbf{Z}_t\mathbf{R}_{t-1}\mathbf{Z}_t$. After K iterations, $\mathbf{Q}_K\mathbf{G}$ approximates the polar factor. The accumulator $\mathbf{Q}_K$ plays the role of an

inverse-square-root multiplier, but a finite quintic schedule with Polar Express coefficients (Amsel et al., 2026) is not the classical Newton correction. Only $(a_t, b_t, c_t) = (3/2, -1/2, 0)$ recovers that correction. Practical Gram-NS adds stabilization beyond this naive recurrence (Zhang et al., 2026).

Another coupled form, Newton–Denman–Beavers (NDB), computes a square root and its inverse together (Higham, 2008). For scaled SPD $\mathbf{A}$ with $\|\mathbf{I} - \mathbf{A}\|_2 < 1$, initialize $\mathbf{Y}_0 = \mathbf{A}$, $\mathbf{Z}_0 = \mathbf{I}$, and iterate

$$\mathbf{E}_t = (3\mathbf{I} - \mathbf{Z}_t\mathbf{Y}_t)/2, \qquad \mathbf{Y}_{t+1} = \mathbf{Y}_t\mathbf{E}_t, \qquad \mathbf{Z}_{t+1} = \mathbf{E}_t\mathbf{Z}_t. \tag{4}$$

The limits are $\mathbf{A}^{1/2}$ and $\mathbf{A}^{-1/2}$, respectively. Two calls give $\mathbf{A}^{-1/4}$: first compute the square root, then its inverse square root. Each iteration uses three matrix multiplications; identity initialization eliminates two in the first iteration. DASH combines NDB with batched preconditioner blocks and multi-power-iteration scaling (Modoranu et al., 2026).

Rational approximation. van den Eshof et al. (2002) compute the action of the matrix sign function on a vector. For a nonsingular Hermitian matrix $\mathbf{H}$, their partial-fraction approach uses

$$\operatorname{sign}(\mathbf{H})\mathbf{b} \approx \sum_{i=1}^{k} \omega_i \mathbf{H}(\mathbf{H}^2 + \tau_i\mathbf{I})^{-1}\mathbf{b}. \tag{5}$$

They use the classical optimal rational approximation of Zolotarev (1877) and solve the shifted linear systems. The identity $\operatorname{sign}(\mathbf{H}) = \mathbf{H}(\mathbf{H}^2)^{-1/2}$ connects this approximation to an inverse square root.

Hale et al. (2008) instead compute fractional powers and other matrix functions through contour quadrature. Specializing their contour formula to a power gives

$$\mathbf{A}^q = \frac{\mathbf{A}}{2\pi\mathrm{i}} \int_\Gamma z^{q-1}(z\mathbf{I} - \mathbf{A})^{-1}\, dz \approx \mathbf{A} \sum_{i=1}^{k} \eta_i z_i^{q-1}(z_i\mathbf{I} - \mathbf{A})^{-1}. \tag{6}$$

Here Γ encloses the positive spectrum counterclockwise without crossing the branch cut; z_i and η_i are quadrature nodes and weights, including the factor $1/(2\pi\mathrm{i})$ in the weights. Conformal maps and trapezoidal quadrature give geometric convergence. General contour nodes can be complex, although specialized constructions can avoid complex arithmetic.

Computing a dense rational approximation with k inverse terms requires k shifted matrix inverses, or equivalent linear solves. Direct computation costs $O(kn^3)$ for $n \times n$ matrices; each dense matrix multiplication in a polynomial method also costs $O(n^3)$. The difficulty is GPU throughput: factorizations and triangular solves have dependencies that dense matrix multiplications avoid, making rational methods appear impractical for training (Amsel et al., 2026; Massena et al., 2026). The approximation count alone therefore does not determine runtime. A useful implementation must turn the independent shifted inverses into GPU-efficient work while retaining enough accuracy for optimization. This is the solver problem addressed next.

3 ZPoles: Rational Approximation with a Mixed-Precision Solver

We propose ZPoles, a rational approximation method for fractional matrix powers that reduces the cost of the shifted inverses introduced above. "Z" denotes the approximate power of $\mathbf{A}$; "Poles" refers to its shifted inverse terms. Our key technical contribution is a mixed-precision solver that batches these independent shifted linear systems, retaining higher precision in small diagonal blocks and using matrix multiplications for most of the work (Section 3.2).

For $\mathbf{A} \succ 0$ and $-1 < q < 0$, we write

$$\mathbf{A}^q \approx \mathbf{Z} = \alpha\mathbf{I} + \sum_{i=1}^{k} \omega_i(\mathbf{A} + \tau_i\mathbf{I})^{-1}, \tag{7}$$

where $\alpha \geq 0$, $\omega_i > 0$, and $\tau_i > 0$ are coefficients computed in Section 3.1.

This form follows from a real-axis integral: for $t > 0$ and $-1 < q < 0$,

$$t^q = \frac{\sin(-\pi q)}{\pi} \int_0^\infty \frac{s^q}{t+s}\, ds \approx \sum_{i=1}^{k} \frac{\omega_i}{t+\tau_i}. \tag{8}$$

Positive quadrature gives the shifted terms; the fitted form in Equation (7) also allows a nonnegative constant α. Hale's contour formula (Equation (6)) can produce complex nodes and weights. For $-1 < q < 0$, the real-axis integral instead gives positive shifts and weights: each $\mathbf{A} + \tau_i \mathbf{I} \succ 0$ uses the same Cholesky solver, without cancellation between scalar terms. Hale et al. also give specialized real-arithmetic constructions. Small shifts resolve variation near zero; larger shifts capture the smoother response elsewhere.

Let $\mathbf{A} = \mathbf{X}\mathbf{X}^\top + \epsilon\mathbf{I}$, with damping $\epsilon > 0$. Set $u = \|\mathbf{A}\|_F$ and normalize in place, $\mathbf{A} \leftarrow \mathbf{A}/u$. Its eigenvalues lie in $[\epsilon/u, 1]$: damping supplies the lower bound and the Frobenius norm the upper bound. This interval covers the spectrum in exact arithmetic without estimating its smallest eigenvalue. Damping defines the target matrix; normalization changes only its scale. We apply Equation (7) to this normalized matrix and restore the original scale with $u^q\mathbf{Z}$. For $\mathbf{A} + \tau_i\mathbf{I} = \mathbf{L}_i\mathbf{L}_i^\top$ and $\mathbf{T}_i = \mathbf{L}_i^{-1}$, each inverse is $\mathbf{T}_i^\top\mathbf{T}_i$. Algorithm 1 summarizes the computation.

Algorithm 1 ZPoles fractional power

Require: Positive-definite $\mathbf{A} \in \mathbb{R}^{n\times n}$ in `fp32`; exponent $q \in (-1, 0)$; pole count k
Require: Coefficients for $\mathbf{A}/\|\mathbf{A}\|_F$: α, $\{(\tau_i, \omega_i)\}_{i=1}^k$ (computed as described in Section 3.1)
1: $u \leftarrow \|\mathbf{A}\|_F$; $\mathbf{A} \leftarrow \mathbf{A}/u$
2: **for** $i = 1, \ldots, k$ in parallel **do**
3: $\mathbf{L}_i, \mathbf{D}_i \leftarrow$ BlockedCholesky$(\mathbf{A}, \tau_i)$ ▷ Algorithm 2; `fp16` outputs
4: $\mathbf{T}_i \leftarrow$ RecursiveInverse$(\mathbf{L}_i, \mathbf{D}_i)$ ▷ Algorithm 3
5: **end for**
6: $\mathbf{Z} \leftarrow \alpha\mathbf{I} + \sum_{i=1}^k \omega_i \mathbf{T}_i^\top \mathbf{T}_i$
7: **return** $u^q\mathbf{Z}$ ▷ power of the original input

3.1 Coefficient Construction

Coefficients depend on q, k, and the normalized spectral interval. Round $\kappa = u/\epsilon$ upward to the next quarter power of two (minimum 2), giving $\bar{\kappa}$, and set $\ell = \bar{\kappa}^{-1/2}$. The bank supplies coefficients on $[\ell^2, 1]$, which contains the normalized spectral interval $[\epsilon/u, 1]$.

For $q \neq -1/2$, initialize from k-point Gauss–Jacobi quadrature. Let z_i, w_i be its nodes and weights for the weight $(1-z)^{-q-1}(1+z)^q$ on $[-1, 1]$. Substituting $s = \ell(1+z)/(1-z)$ in Equation (8) gives

$$\tau_i^{(0)} = \ell\frac{1+z_i}{1-z_i}, \qquad \omega_i^{(0)} = \frac{2\ell^{1+q}\sin(-\pi q)}{\pi}\frac{w_i}{1-z_i}, \qquad \alpha^{(0)} = \max\{10^{-5}, 0.1(1+q)\}. \quad (9)$$

We then optimize the logarithms of all $2k+1$ coefficients, minimizing

$$\sum_{r=1}^{1024}\left[\log\left(\alpha + \sum_{i=1}^k \frac{\omega_i}{t_r + \tau_i}\right) - q\log t_r\right]^2, \qquad t_r = \ell^{\,2(1024-r)/1023}. \quad (10)$$

Logarithmic samples resolve both ends of the interval; log-relative error balances their response magnitudes, since $\log(1+e) \approx e$ for small relative error e. Bounded nonlinear least squares in `fp64` preserves positivity by exponentiating the fitted variables. Pure quadrature uses the initial shifts and weights with $\alpha = 0$.

For $q = -1/2$, we construct a Zolotarev sign approximant $s(x)$ on $[-1, -\ell] \cup [\ell, 1]$. The multiplier $s(\sqrt{t})/\sqrt{t}$ approximates $t^{-1/2}$ on $[\ell^2, 1]$; partial fractions give the positive shifts and weights (Zolotarev, 1877; Nakatsukasa & Freund, 2016). Appendix A gives the construction, fitting settings, and numerical coefficient sets (Table 3). Dynamic exponents use log-coefficient interpolation (Appendix E).

3.2 Batched Mixed-Precision Linear Solves

The shifted linear systems in Equation (7) share the same matrix $\mathbf{A}$, but each shift has its own Cholesky factors; no factorization is shared across poles. We organize their computation around (1) **mixed precision:** `fp16` matrix multiplications with `fp32` accumulation and diagonal operations; (2) **shared structure:** batched panel updates and inverse merges across poles; and (3) **reuse:** diagonal inverses seed the recursive inversion, whose weighted outputs form one matrix-power operator.

Blocked Cholesky (Algorithm 2). Blocked Cholesky processes one block column at a time, expressing most work as matrix multiplications suitable for batched mixed precision (Dong et al., 2014; Abdelfattah et al., 2020). For $\mathbf{A} + \tau\mathbf{I} = \mathbf{L}\mathbf{L}^\top$, the updates at block column j are

$$\mathbf{S}_j = \mathbf{A}_{jj} + \tau\mathbf{I} - \sum_{r<j} \mathbf{L}_{jr}\mathbf{L}_{jr}^\top, \qquad \mathbf{P}_j = \mathbf{A}_{>j,j} - \sum_{r<j} \mathbf{L}_{>j,r}\mathbf{L}_{jr}^\top. \tag{11}$$

Here $> j$ denotes block rows below j. Factor $\mathbf{S}_j = \mathbf{L}_{jj}\mathbf{L}_{jj}^\top$, form $\mathbf{D}_j = \mathbf{L}_{jj}^{-1}$, and compute $\mathbf{L}_{>j,j} = \mathbf{P}_j\mathbf{D}_j^\top$. Thus only diagonal blocks require factorization and inversion; all larger updates are matrix multiplications.

Our implementation uses 128×128 blocks and advances each block column across all shifted linear systems in a batch (Figure 1). Schur outputs $\mathbf{S}_j, \mathbf{P}_j$ and diagonal operations use `fp32`. To reduce rounding error in the `fp16` updates, we employ compensated matrix multiplication (CMul), using high–residual operand splitting (Markidis et al., 2018) and the scaled three-product correction of Ootomo & Yokota (2022). For a factor block or panel operand $\mathbf{B} \in \{\mathbf{L}_{ab}, \mathbf{P}_j, \mathbf{D}_j\}$, define $\mathbf{B}_h = \texttt{fp16}(\mathbf{B})$ and $\mathbf{B}_e = \texttt{fp16}(4096[\mathbf{B} - \texttt{fp32}(\mathbf{B}_h)])$. We approximate $\mathbf{U}\mathbf{V}^\top$ by

$$\mathrm{CMul}(\mathbf{U}, \mathbf{V}) = \mathbf{U}_h\mathbf{V}_h^\top + (\mathbf{U}_h\mathbf{V}_e^\top + \mathbf{U}_e\mathbf{V}_h^\top)/4096. \tag{12}$$

These three `fp16` GEMMs accumulate in `fp32`. Completed factor blocks retain both parts; panel operands are split internally. Omitting the residual–residual product saves 25% of the four-product expansion; the larger gain over `fp32` GEMMs comes from `fp16` Tensor Cores. Algorithm 3 receives only $\mathbf{L}_h$ and rounded diagonal inverses; residuals are used only during Cholesky (Appendix C.4).

Recursive triangular inversion (Algorithm 3). Classical block-recursive inversion (Demmel et al., 2007; Higham, 1995) merges the diagonal inverses from Algorithm 2 using

$$\begin{bmatrix} \mathbf{L}_{11} & \mathbf{0} \\ \mathbf{L}_{21} & \mathbf{L}_{22} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{T}_{11} & \mathbf{0} \\ -\mathbf{T}_{22}\mathbf{L}_{21}\mathbf{T}_{11} & \mathbf{T}_{22} \end{bmatrix}, \qquad \mathbf{T}_{ii} = \mathbf{L}_{ii}^{-1}. \tag{13}$$

Multiplying these two block matrices gives the identity: the diagonal blocks multiply to identities and the lower-left terms cancel. Doubling the group size at each level builds the full inverse. Each merge uses two matrix multiplications; independent pairs across poles run together (Figure 2).

Merge depth is logarithmic, but work grows with matrix size and pole count. Initialization performs the first merge; later levels use two launches each. Merge intermediates use `fp16` storage and `fp32` accumulation. Weighted assembly in Equation (7) accumulates in `fp32` but stores the nonconstant operator in `fp16`, before conversion to `fp32`, addition of $\alpha\mathbf{I}$, and rescaling. An `fp32` output therefore does not imply `fp32` accuracy throughout.

3.3 Numerical Stability and Implementation Constraints

A symmetric perturbation $\mathbf{\Delta}$ preserves positive definiteness if $\|\mathbf{\Delta}\|_2 < \lambda_{\min}(\mathbf{A}) + \tau_i$. Small shifts increase sensitivity; more poles need not reduce total error. Panel compensation does not correct rounding in triangular inversion or operator storage. Training checks factorization status and finite outputs, but neither check certifies accuracy. The backend requires contiguous `fp32` inputs, dimensions divisible by 128, and one stream; padding and concurrent calls are untested. Inverse buffers occupy $2kn^2$ bytes beyond factor and accumulation storage. The damped-statistics wrapper requires nonzero positive-semidefinite statistics and positive damping. The dynamic momentum experiment instead selects its interval heuristically (Appendix E); the covering-interval guarantee for damped statistics does not apply to that heuristic.

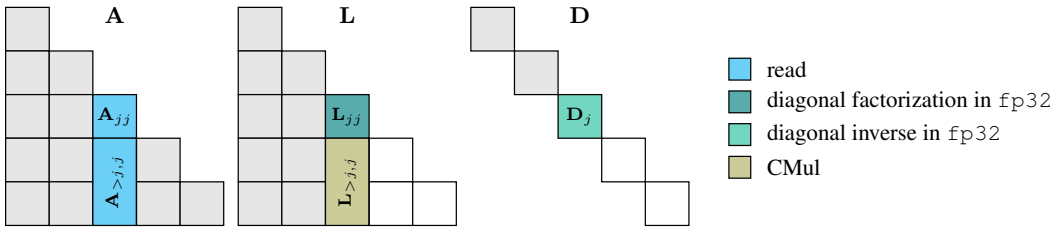


Figure 1: Converting a panel solve into a matrix multiplication. The illustrated column is $j = 2$ (zero-based). Cyan marks the input diagonal block and panel. After subtracting contributions from earlier columns, the diagonal block is factored and inverted in `fp32`. The inverse $\mathbf{D}_j$ converts the panel solve into $\mathbf{P}_j\mathbf{D}_j^\top$, computed with CMul. Algorithm 2 repeats this block-column update across all shifted matrices in a batch; only their diagonal shifts differ.

Algorithm 2 BlockedCholesky: mixed-precision factorization

Require: `fp32` input matrix $\mathbf{A}$ and shift $\tau > 0$; block size $\beta = 128$
Store each completed $\mathbf{L}$ block as $(\mathbf{L}_h, \mathbf{L}_e)$ using the split above.
Temporary blocks and diagonal inverses $\mathbf{D}_j$ use `fp32`.

for block columns $j = 0, \ldots, n/\beta - 1$ **do**
  $\mathbf{S}_j \leftarrow \mathbf{A}_{jj} + \tau\mathbf{I} - \sum_{r<j} \mathrm{CMul}(\mathbf{L}_{jr}, \mathbf{L}_{jr})$
  $\mathbf{L}_{jj} \leftarrow \mathrm{chol}(\mathbf{S}_j)$ ▷ `fp32` Cholesky factorization
  $\mathbf{D}_j \leftarrow \mathbf{L}_{jj}^{-1}$ ▷ `fp32` triangular inversion
  $\mathbf{P}_j \leftarrow \mathbf{A}_{>j,j} - \sum_{r<j} \mathrm{CMul}(\mathbf{L}_{>j,r}, \mathbf{L}_{jr})$
  $\mathbf{L}_{>j,j} \leftarrow \mathrm{CMul}(\mathbf{P}_j, \mathbf{D}_j)$
end for
return `fp16`$(\mathbf{L})$, `fp16`$(\mathbf{D})$

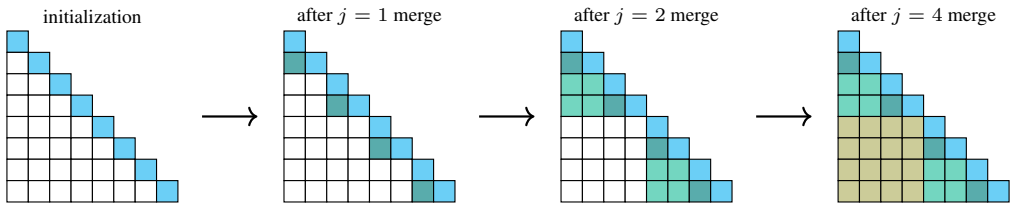


Figure 2: Building the triangular inverse through independent merges (Algorithm 3). Panels show $\mathbf{T}$, initialized from $\mathbf{D}_0, \ldots, \mathbf{D}_7$. A merge joins two j-block groups into one $2j$-block group. Subscripts $11, 21, 22$ denote local positions within that pair, not global block indices. Colors mark the newly computed $\mathbf{T}_{21}$. Each merge preserves the two child inverses and forms the lower-left block as $-\mathbf{T}_{22}\mathbf{L}_{21}\mathbf{T}_{11}$ using two matrix multiplications. Disjoint pairs, including those from different shifted matrices, are independent within a level. Doubling the group size recovers the full inverse in logarithmically many dependent levels; an unpaired group is carried forward unchanged.

Algorithm 3 RecursiveInverse: merge diagonal inverse blocks

Require: `fp16` factor $\mathbf{L}$ and diagonal inverses $\mathbf{D}$ from Algorithm 2
$\mathbf{T} \leftarrow \mathrm{blockdiag}(\mathbf{D}_0, \mathbf{D}_1, \ldots)$
for $j = 1, 2, 4, \ldots$ while $j < \lceil n/\beta \rceil$ **do**
  for all adjacent group pairs independently; use local positions $11, 21, 22$ **do**
    $\mathbf{T}_{21} \leftarrow -\mathbf{T}_{22}\mathbf{L}_{21}\mathbf{T}_{11}$ ▷ `fp16` GEMMs
  end for
  Carry any unpaired group to the next level
end for
return $\mathbf{T}$

4 THEORETICAL ANALYSIS

The analysis characterizes the approximation family and separates error sources; it does not guarantee the accuracy of fitted coefficients or training performance. Error arises from rational approximation, computed inverses, and changes in cached preconditioner inputs. Throughout, matrices are SPD and coefficients satisfy $\alpha \geq 0$, $\omega_i > 0$, and $\tau_i > 0$. All proofs appear in Appendix B.

For $-1 < q < 0$, Equation (8) places fractional powers in the Stieltjes class: positive mixtures of shifted inverses. This gives an existence guarantee within the same positive-shift family as ZPoles.

Proposition 1 (Constructive convergence). *Fix $0 < a < b$ and $-1 < q < 0$. Let $\chi = (\sqrt{b} - \sqrt{a})/(\sqrt{b} + \sqrt{a})$. For every $k \geq 1$, there exist positive τ_i, ω_i such that, with $\alpha = 0$, $r_k(t) = \alpha + \sum_{i=1}^{k} \omega_i/(t + \tau_i)$ satisfies*

$$\max_{t \in [a,b]} |t^{-q} r_k(t) - 1| \leq \frac{2(1+\chi)}{1-\chi} \chi^{2k}. \tag{14}$$

For an SPD matrix with spectrum in $[a, b]$, the same bound holds for $\|r_k(\mathbf{A}) - \mathbf{A}^q\|_F / \|\mathbf{A}^q\|_F$.

The bound establishes existence, not the accuracy of our fitted coefficients. At $b/a = 100$, it gives 4.02, 0.807, and 0.0325 for $k = 4, 8, 16$: it is vacuous as a small-error guarantee at 4 poles. Allowing zero weights makes the approximation families nested; the best attainable error cannot increase with k. Finite-precision computations need not improve monotonically.

Proposition 2 (Residual-to-power error). *Fix $-1 < q < 0$ and let $\lambda_{\min}(\mathbf{A}) \geq a > 0$. Suppose a rational approximant with $\alpha \geq 0$, $\omega_i > 0$, and $\tau_i > 0$ has scalar relative error at most ε on its spectrum. For computed inverse terms $\widehat{\mathbf{S}}_i$, set $\mathbf{F}_i = (\mathbf{A} + \tau_i \mathbf{I})\widehat{\mathbf{S}}_i - \mathbf{I}$. Define the assembly error by $\widehat{\mathbf{Z}} = \alpha \mathbf{I} + \sum_i \omega_i \widehat{\mathbf{S}}_i + \mathbf{E}$. Then*

$$\frac{\|\widehat{\mathbf{Z}} - \mathbf{A}^q\|_F}{\|\mathbf{A}^q\|_F} \leq \varepsilon + \frac{\|\mathbf{E}\|_F + \sum_i \omega_i \|\mathbf{F}_i\|_F/(a + \tau_i)}{\|\mathbf{A}^q\|_F}. \tag{15}$$

This bound fixes the stored input and coefficients. It separates scalar approximation from solve and assembly errors; input-formation errors are not included.

Proposition 3 (Effect of covariance drift). *For $-1 < q < 0$, if $\mathbf{A}, \mathbf{D} \succeq a\mathbf{I}$ with $a > 0$, then*

$$\|\mathbf{A}^q - \mathbf{D}^q\|_2 \leq (-q) a^{q-1} \|\mathbf{A} - \mathbf{D}\|_2. \tag{16}$$

A cached approximation $\widehat{\mathbf{D}^q}$ incurs its own error plus this drift term relative to the current power $\mathbf{A}^q$. The bound explains the sensitivity to stale statistics; it does not predict training loss.

5 EXPERIMENTS

We compare ZPoles with coupled Newton (CN), Newton–Denman–Beavers (NDB), Newton–Schulz (NS), Gram-form Newton–Schulz (Gram-NS), and Newton–Schulz with a second-order Taylor correction (NS–Taylor). The comparisons cover linear solves, complete matrix-power computation, and training. Within each optimizer, backends share the training configuration.

5.1 SETUP

Primary experiments use RTX 5090 GPUs; ZPoles uses 4 poles unless stated otherwise. Our solver uses CMul (12) by default throughout, except in the explicit GEMM ablations, for its speed–accuracy tradeoff (Table 4). All NS baselines use Polar Express coefficients (Amsel et al., 2026). Figure 3 uses `fp16` matrix-multiplication operands and an `fp64` reference; the solver comparison uses the library's `fp32` path. Matrix benchmarks test scaling at orders 1024–12288. The main matrix benchmarks process one matrix per call. The solver comparison averages 10 random inputs per size; the root comparison uses one fixed input per size. Each input is timed 20 times after warmup. Methods share the same input within each comparison, and errors use relative Frobenius norms. Appendix C gives protocols and precision controls; Section 5.4 specifies the training setup.

5.2 LINEAR SOLVES: cuSOLVER AND OURS

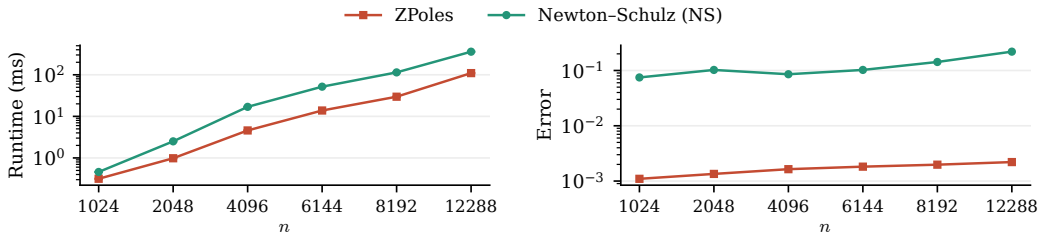
We compute the inverse $\mathbf{A}^{-1}$ for $\mathbf{A} = \mathbf{X}\mathbf{X}^\top/n + 0.01\mathbf{I}$, with independent standard Gaussian X_{ij}. A linear solver obtains $\mathbf{Y} = \mathbf{A}^{-1}$ by solving $\mathbf{A}\mathbf{Y} = \mathbf{I}$ with the n columns of $\mathbf{I}$ as right-hand sides. The baseline uses PyTorch's `cuSOLVER` backend: `cholesky_ex` factors $\mathbf{A}$, then `cholesky_solve` solves against $\mathbf{I}$, both in `fp32`. Timing excludes Gram construction. Ours uses one unshifted inverse term. Table 1 reports error $100\|\widehat{\mathbf{Y}} - \mathbf{Y}\|_F/\|\mathbf{Y}\|_F$, where $\widehat{\mathbf{Y}}$ is the computed solution and $\mathbf{Y}$ the `fp64` reference. Ours is 3.7–4.1× faster with 57–99× higher error. Appendix C.1 gives ablations. For complete ZPoles at fixed coefficients, ours is 4.9–10.2× faster with slightly higher error (Appendix Table 6).

Table 1: Computing $\mathbf{A}^{-1}$ on RTX 5090. Results averaged over 10 random matrices; Ours denotes the isolated linear solver. Bold marks the lowest value per size and metric.

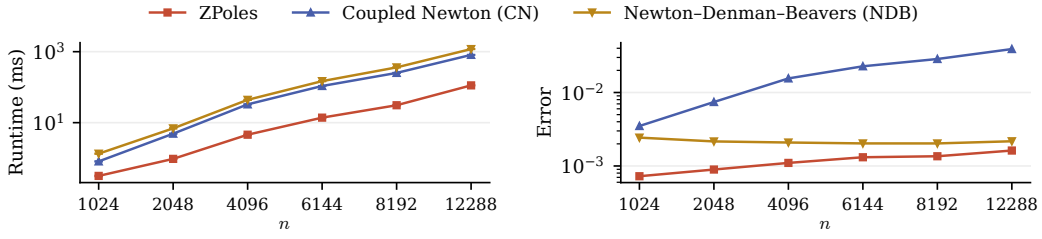
Metric	Method	1024	2048	4096	6144	8192	12288
Runtime (ms)	cuSOLVER	0.909	2.268	7.067	18.292	34.344	98.602
	Ours	**0.244**	**0.592**	**1.889**	**4.481**	**8.601**	**25.215**
Error (%)	cuSOLVER	**0.00076**	**0.00105**	**0.00148**	**0.00109**	**0.00110**	**0.00113**
	Ours	0.07554	0.08129	0.08444	0.08529	0.08583	0.08655

5.3 INVERSE-ROOT RUNTIME AND APPROXIMATION ERROR

For standard Gaussian $\mathbf{X} \in \mathbb{R}^{n\times n}$, we form $\mathbf{A} = \mathbf{X}\mathbf{X}^\top + \epsilon\mathbf{I}$, with $\epsilon = 0.01\,\mathrm{tr}(\mathbf{X}\mathbf{X}^\top)/n$. Figure 3 compares complete runtime and relative Frobenius error at the six tested sizes. For inverse fourth roots, ZPoles is 2.6–8.1× faster than CN and 4.2–11.7× faster than NDB, with lower error under the iteration counts in the caption and the precision settings in Section 5.1. Inverse square roots are compared with Gram-form NS. CN's fixed budget becomes less adequate as Frobenius scaling shrinks the initial spectrum. NS targets the polar factor; its multiplier error emphasizes small eigenvalues unresolved after 5 iterations. Accuracy-matched controls appear in Appendix C.



(a) Inverse square root $\mathbf{A}^{-1/2}$.



(b) Inverse fourth root $\mathbf{A}^{-1/4}$.

Figure 3: Inverse square roots (a) and fourth roots (b) of damped Gaussian Gram matrices. Runtime includes scaling and assembly; multiplier error is $\|\widehat{\mathbf{Z}} - \mathbf{A}^q\|_F/\|\mathbf{A}^q\|_F$ against an `fp64` reference. Each size uses one fixed matrix. Runtime is the mean and sample standard deviation of 20 timed repetitions after warmup; error is measured on that matrix. Budgets: ZPoles 4 poles; CN 10 iterations; NDB two 10-iteration stages; Gram-form NS 5 iterations.

5.4 TRAINING

We compare matrix routines in GPT-2-Small/Medium (124M/355M) pretraining on FineWeb (Penedo et al., 2024) (Table 2): approximately 1B tokens, 32 sequences of length 1024 per update. We use maximum learning rate $\eta_{\max} = 0.002$ for the main comparison and additionally test 0.001 and 0.005 with the same training budget; Appendix D.3 reports the completed comparisons. The compared optimizers update attention and MLP weight matrices; AdamW updates token and positional embeddings, the tied output head, biases, and LayerNorm parameters, using the same learning rate and schedule. CN, Gram-NS, and ZPoles use `fp16` GEMM operands; NDB uses DASH's `fp32` routine (Modoranu et al., 2026) because its `fp16` training runs failed. Muon and DynMuon retain released `bf16` NS operands. Full protocols appear in Appendices D and E.

Shampoo. The grafted Shampoo optimizer uses $q = -1/4$, relative ridge 0.01, $\eta_{\max} = 0.002$, and per-update root refreshes on blocks of order at most 768. ZPoles uses 4 poles, CN 10 iterations, and NDB two 10-iteration stages with multi-power-iteration scaling. In these paired runs, ZPoles lowers Shampoo loss by 0.0009 on GPT-2-Small and 0.0021 on GPT-2-Medium, with observed step-time reductions of 24.2% and 23.8% relative to CN. Relative to NDB, the loss reductions are 0.0198 and 0.0411. Appendix D.1 plots loss against time.

Muon. The main Muon comparison uses $\eta_{\max} = 0.002$, selected from the same Small-model grid. ZPoles reaches slightly lower loss than NS and Gram-NS (Zhang et al., 2026) on both models, but the differences from NS are only 0.0003 and 0.0016. Loss curves appear in Appendix D.1.

Fractional-power Muon. We use DynMuon's released exponent schedule (Wu et al., 2026) and $\eta_{\max} = 0.002$. ZPoles replaces the polar and negative-power transforms, leaving the raw-momentum branch unchanged. In the negative-power branch, it directly approximates $\mathbf{A}^{(p-1)/2}$ instead of applying a Taylor correction to a polar factor. Relative to NS–Taylor, loss decreases by 0.0113 on GPT-2-Small and 0.0028 on GPT-2-Medium; observed step time decreases by 2.5% and 4.9%, respectively.

Table 2: Final GPT-2 validation loss and mean runtime/step (ms) from separate runs, excluding validation and the first 100 updates. Optimizer settings are shared within each group. Bold marks the lowest loss and runtime per group and model.

		GPT-2-Small		GPT-2-Medium	
Optimizer	Backend	Val loss	Runtime/step	Val loss	Runtime/step
Shampoo	CN	3.7099	298.3	3.5593	797.0
	NDB	3.7288	419.1	3.5983	1166.1
	ZPoles	**3.7090**	**225.9**	**3.5572**	**607.4**
Muon	NS	3.5821	194.2	3.4033	536.4
	Gram-NS	3.5828	191.7	3.4059	526.0
	ZPoles	**3.5818**	**189.1**	**3.4017**	**516.6**
DynMuon	NS–Taylor	3.5542	196.9	3.3860	550.2
	ZPoles	**3.5429**	**192.1**	**3.3832**	**523.2**

6 CONCLUSION

ZPoles combines rational approximation with batched mixed-precision solves for fractional matrix powers. Under the tested accuracy budgets, it reduces matrix-power runtime and observed GPT-2 step cost with lower or similar validation loss. Future work will explore how rational approximations can inform the design of new optimization methods and broaden the role of matrix functions in machine learning.

AI USE STATEMENT

GPT-6, Kimi-K3, and GLM-5.3 assisted with method and hypothesis development, formulating mathematical claims and drafting and checking proofs, experimental design, method implementation and debugging, and interpreting results. They also assisted with literature searches, manuscript editing, and preparing figures and tables. The authors are responsible for the final text, mathematical claims, code, and experimental results.

REFERENCES

Ahmad Abdelfattah, Stan Tomov, and Jack Dongarra. Investigating the benefit of FP16-enabled mixed-precision solvers for symmetric positive definite matrices using GPUs. In *Computational Science – ICCS*, 2020.

Noah Amsel, David Persson, Christopher Musco, and Robert M. Gower. The Polar Express: Optimal matrix sign methods and their application to the Muon algorithm. In *International Conference on Learning Representations*, 2026.

Rohan Anil, Vineet Gupta, Tomer Koren, Kevin Regan, and Yoram Singer. Scalable second order optimization for deep learning. *arXiv preprint arXiv:2002.09018*, 2020.

Jeremy Bernstein and Laker Newhouse. Old optimizer, new norm: An anthology. *arXiv preprint arXiv:2409.20325*, 2024.

James Demmel, Ioana Dumitriu, and Olga Holtz. Fast linear algebra is stable. *Numerische Mathematik*, 2007.

Guillaume Desjardins, Karen Simonyan, Razvan Pascanu, and Koray Kavukcuoglu. Natural neural networks. In *Advances in Neural Information Processing Systems*, 2015.

Tingxing Dong, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. A fast batched Cholesky factorization on a GPU. In *43rd International Conference on Parallel Processing*, 2014.

Yihe Dong and Will Sawin. Muon^p: Muon with fractional spectral powers. *arXiv preprint arXiv:2606.13867*, 2026.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 2011.

Chun-Hua Guo and Nicholas J. Higham. A Schur–Newton method for the matrix pth root and its inverse. *SIAM Journal on Matrix Analysis and Applications*, 2006.

Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. In *Proceedings of the 35th International Conference on Machine Learning*, Proceedings of Machine Learning Research, 2018.

Nicholas Hale, Nicholas J. Higham, and Lloyd N. Trefethen. Computing A^α, $\log(A)$, and related matrix functions by contour integrals. *SIAM Journal on Numerical Analysis*, 2008.

Nicholas J. Higham. Stability of parallel triangular system solvers. *SIAM Journal on Scientific Computing*, 1995.

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

Lei Huang, Yi Zhou, Fan Zhu, Li Liu, and Ling Shao. Iterative normalization: Beyond standardization towards efficient whitening. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019.

Bruno Iannazzo. On the Newton method for the matrix pth root. *SIAM Journal on Matrix Analysis and Applications*, 2006.

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

Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. NVIDIA Tensor Core programmability, performance & precision. In *IEEE International Parallel and Distributed Processing Symposium Workshops*, 2018.

Thomas Massena, Corentin Friedrich, and Mathieu Serrurier. From SGD to Muon: Adaptive optimization via Schatten-p norms. *arXiv preprint arXiv:2605.19781*, 2026.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *International Conference on Learning Representations*, 2018.

Ionut-Vlad Modoranu, Philip Zmushko, Erik Schultheis, Mher Safaryan, and Dan Alistarh. DASH: Faster Shampoo via batched block preconditioning and efficient inverse-root solvers. *arXiv preprint arXiv:2602.02016*, 2026.

Yuji Nakatsukasa and Roland W. Freund. Computing fundamental matrix decompositions accurately via the matrix sign function in two iterations: The power of Zolotarev's functions. *SIAM Review*, 2016.

Hiroyuki Ootomo and Rio Yokota. Recovering single precision accuracy from Tensor Cores while surpassing the FP32 theoretical peak performance. *The International Journal of High Performance Computing Applications*, 2022.

Tianyu Pang, Yujie Fang, Zihang Liu, Shenyang Deng, Lei Hsiung, Shuhua Yu, and Yaoqing Yang. HTMuon: Improving Muon via heavy-tailed spectral correction. *arXiv preprint arXiv:2603.10067*, 2026.

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. In *Advances in Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2406.17557.

Xianbiao Qi, Marco Chen, Jiaquan Ye, Yelin He, and Rong Xiao. Delving into Muon and beyond: Deep analysis and extensions. *arXiv preprint arXiv:2602.04669*, 2026.

Hao-Jun Michael Shi, Tsung-Hsien Lee, Shintaro Iwasaki, Jose Gallego-Posada, Zhijing Li, Kaushik Rangadurai, Dheevatsa Mudigere, and Michael Rabbat. A distributed data-parallel PyTorch implementation of the distributed Shampoo optimizer for training neural networks at-scale. *arXiv preprint arXiv:2309.06497*, 2023.

Zakhar Shumaylov, Nathaël Da Costa, Peter Zaika, Bálint Mucsányi, Alex Massucco, Yoav Gelberg, Carola-Bibiane Schönlieb, Yarin Gal, and Philipp Hennig. Muon is not that special: Random or inverted spectra work just as well. *arXiv preprint arXiv:2605.11181*, 2026.

Jasper van den Eshof, Andreas Frommer, Thomas Lippert, Klaus Schilling, and Henk A. van der Vorst. Numerical methods for the QCD overlap operator: I. sign-function and error bounds. *Computer Physics Communications*, 2002.

Fangzhou Wu, Rikhav Shah, Sandeep Silwal, and Qiuyi Zhang. DynMuon: A dynamic spectral shaping view of Muon. *arXiv preprint arXiv:2605.17109*, 2026.

Jack Zhang, Noah Amsel, Berlin Chen, and Tri Dao. Gram Newton–Schulz: A fast, hardware-aware Newton–Schulz algorithm for Muon. https://dao-lab.ai/blog/2026/gram-newton-schulz/, 2026.

Egor Ivanovich Zolotarev. Application of elliptic functions to questions of functions deviating least and most from zero. *Zapiski Imperatorskoi Akademii Nauk, St. Petersburg*, 1877.

A Coefficient construction and numerical values

A.1 Quadrature and fitted coefficients

Section 3.1 gives the quadrature formulas and fitting objective. Here we specify the remaining construction details and numerical values.

Pure quadrature needs no nonlinear fit: use the shifts and weights in Section 3.1 with $\alpha = 0$. The positive constant $\alpha^{(0)}$ instead initializes the fitted family. A positive lower endpoint is essential: t^q diverges as $t \to 0$, whereas a finite sum with positive shifts remains finite. Consequently, these coefficients target $[\ell^2, 1]$, not a uniformly accurate inverse power on all of $(0, 1]$.

A.2 Explicit inverse-square-root construction

For the interval $[\ell^2, 1]$, set $m = 1 - \ell^2$ and $v_r = rK(m)/(2k+1)$, $r = 1, \ldots, 2k$, where K is the complete elliptic integral of the first kind and m is its parameter. Using Jacobi elliptic functions, compute

$$c_r = \ell^2 \left(\frac{\mathrm{sn}(v_r \mid m)}{\mathrm{cn}(v_r \mid m)} \right)^2, \qquad \tau_i = c_{2i-1}, \quad e_i = c_{2i}.$$

The partial-fraction coefficients before centering are

$$\alpha_0 = \prod_{i=1}^{k} \frac{1+\tau_i}{1+e_i}, \qquad \omega_{i,0} = \alpha_0 \frac{\prod_{r=1}^{k}(e_r - \tau_i)}{\prod_{r \neq i}(\tau_r - \tau_i)}.$$

Set $b = \ell[\alpha_0 + \sum_i \omega_{i,0}/(\ell^2 + \tau_i)]$ and $g = 2/(1+b)$. The returned coefficients are $\alpha = g\alpha_0$, $\omega_i = g\omega_{i,0}$, and the unchanged τ_i. This is the elliptic-function construction used in the code; no nonlinear fit is performed for $q = -1/2$. For other exponents, fitting bounds log-coefficients to $[-60, 30]$, allows 400 function calls, and uses tolerances 10^{-11}. The least-squares routine retains a finite final iterate even if it reaches its call limit. Reaching that limit does not certify convergence or a uniform approximation-error bound.

A.3 Numerical coefficient sets

Table 3 gives representative 4-pole sets produced by the same coefficient generator used in the experiments. The intervals $[1/128, 1]$ and $[1/1024, 1]$ are entries in the covering bank. These are normalized-matrix coefficients; Algorithm 1 restores the scale with u^q. The $q = -1/2$ rows use Zolotarev coefficients; the other rows use the fitted construction in Section 3.1. Values are printed with four digits after the decimal in scientific notation; computations retain unrounded coefficients.

B Proofs of the theoretical results

Proof of Proposition 1. Write $d\mu(s) = \sin(-\pi q) s^q ds/\pi$ and $\gamma = \sqrt{ab}$. Define the finite positive measure $d\mu(s)/(\gamma + s)$ and push it forward to $(-1, 1)$ under $v = (s-\gamma)/(s+\gamma)$; denote the resulting measure by $d\nu(v)$. Its total mass is $M = \gamma^q$. For $z = (t-\gamma)/(t+\gamma)$,

$$t^q = \frac{2\gamma}{t+\gamma} \int_{-1}^{1} \frac{d\nu(v)}{1 - zv}.$$

A k-node positive Gaussian rule with nodes $v_i \in (-1, 1)$ and weights $a_i > 0$ is exact for polynomials through degree $2k-1$. It produces shifted inverses with $\tau_i = \gamma(1+v_i)/(1-v_i) > 0$ and $\omega_i = 2\gamma a_i/(1-v_i) > 0$. The remainder after truncating the geometric series of $(1-zv)^{-1}$ at degree $2k-1$ is at most $|z|^{2k}/(1-|z|)$. Both integration and quadrature have total mass M, hence their absolute difference is at most $2M|z|^{2k}/(1-|z|)$. The exact integral is at least $M/(1+|z|)$. The map $t \mapsto (t-\gamma)/(t+\gamma)$ is increasing and takes the values $-\chi$ and χ at a and b, respectively; hence $|z| \leq \chi$. Dividing proves Equation (14). For the matrix bound, let λ_i be the eigenvalues of $\mathbf{A}$, $e_i = \lambda_i^{-q} r_k(\lambda_i) - 1$, and $\delta = 2(1+\chi)\chi^{2k}/(1-\chi)$. In an orthonormal eigenbasis,

$$\|r_k(\mathbf{A}) - \mathbf{A}^q\|_F^2 = \sum_i e_i^2 \lambda_i^{2q} \leq \delta^2 \|\mathbf{A}^q\|_F^2.$$

Table 3: Coefficients for $\alpha + \sum_{i=1}^{4} \omega_i/(t+\tau_i)$ on $[1/\bar{\kappa}, 1]$.

$\bar{\kappa}$	q	α	i	τ_i	ω_i
128	$-1/4$	3.7253×10^{-1}	1	3.2899×10^{-3}	1.0595×10^{-2}
			2	3.9146×10^{-2}	4.0735×10^{-2}
			3	2.9504×10^{-1}	1.8842×10^{-1}
			4	3.7383×10^{0}	2.0473×10^{0}
128	$-1/2$	2.7006×10^{-1}	1	1.4930×10^{-3}	5.2141×10^{-2}
			2	2.1326×10^{-2}	9.3174×10^{-2}
			3	1.4043×10^{-1}	2.2207×10^{-1}
			4	1.0960×10^{0}	8.2156×10^{-1}
128	$-5/8$	1.7899×10^{-1}	1	1.0872×10^{-3}	1.1516×10^{-1}
			2	1.9028×10^{-2}	1.3510×10^{-1}
			3	1.2460×10^{-1}	2.4744×10^{-1}
			4	9.3047×10^{-1}	6.8235×10^{-1}
1024	$-1/4$	6.2162×10^{-1}	1	5.7007×10^{-4}	2.9322×10^{-3}
			2	7.9371×10^{-3}	1.3441×10^{-2}
			3	7.0690×10^{-2}	6.7693×10^{-2}
			4	7.7928×10^{-1}	5.3301×10^{-1}
1024	$-1/2$	3.4328×10^{-1}	1	3.1259×10^{-4}	2.4654×10^{-2}
			2	5.7403×10^{-3}	5.6429×10^{-2}
			3	5.4541×10^{-2}	1.6612×10^{-1}
			4	5.9135×10^{-1}	6.6656×10^{-1}
1024	$-5/8$	2.4515×10^{-1}	1	2.3053×10^{-4}	6.5914×10^{-2}
			2	5.0812×10^{-3}	9.5100×10^{-2}
			3	4.7128×10^{-2}	2.0535×10^{-1}
			4	4.8544×10^{-1}	5.9295×10^{-1}

This proof concerns a quadrature construction; it does not certify the coefficient procedures used in the experiments. □

Proof of Proposition 2. Subtracting the exact inverse gives $\widehat{\mathbf{S}}_i - (\mathbf{A} + \tau_i \mathbf{I})^{-1} = (\mathbf{A} + \tau_i \mathbf{I})^{-1} \mathbf{F}_i$. The triangle inequality proves the claim. □

Proof of Proposition 3. The matrix inverse identity gives

$$(\mathbf{A} + s\mathbf{I})^{-1} - (\mathbf{D} + s\mathbf{I})^{-1} = (\mathbf{A} + s\mathbf{I})^{-1}(\mathbf{D} - \mathbf{A})(\mathbf{D} + s\mathbf{I})^{-1}.$$

Applying the scalar integral representation in each matrix's eigenbasis gives

$$\mathbf{A}^q - \mathbf{D}^q = \frac{\sin(-\pi q)}{\pi} \int_0^\infty s^q \left[(\mathbf{A} + s\mathbf{I})^{-1} - (\mathbf{D} + s\mathbf{I})^{-1} \right] ds.$$

The individual integrands are bounded in norm by $s^q/(a+s)$, which is integrable at both endpoints because $-1 < q < 0$. Taking spectral norms and using the inverse identity reduces the constant to $\sin(-\pi q) \int_0^\infty s^q (a+s)^{-2} ds/\pi = (-q) a^{q-1}$, the negative derivative of a^q. A uniform lower bound is necessary; a nearly singular undamped covariance can be extremely sensitive to drift. □

C Matrix experiments and implementation records

C.1 Linear Solver Ablation

All four solver variants receive the same rounded `fp32` input and solve against an identity right-hand side. The reference is an `fp64` Cholesky solve of that input. Relative solution error is $\|\widehat{\mathbf{S}} - \mathbf{S}_\star\|_F / \|\mathbf{S}_\star\|_F$. Gram construction and the reference are outside timing; factorization, inversion or triangular solves, assembly, and output conversion are included. The solver test fixes an

absolute ridge after scaling the Gram matrix by $1/n$, whereas the root tests use a trace-relative ridge. These are separate workloads; methods within each comparison receive identical inputs. The custom solver uses one pole with zero shift and unit weight to isolate inverse construction; it retains compensated factor storage. Its `fp32`-GEMM ablation reconstructs stored factor operands in `fp32` and replaces the compensated products with IEEE `fp32` multiplication. Other kernels and storage are unchanged; this is not a separately tuned all-`fp32` solver. The library path selects the `cuSOLVER` backend and calls `torch.linalg.cholesky_ex` followed by `torch.cholesky_solve`.

For each matrix size, we generate 10 independent Gaussian inputs. After warmup, we time each method 20 times on each input, giving 200 runtime measurements per method and size. We report their mean. Timing uses CUDA graphs on an otherwise idle RTX 5090, alternating forward and reverse method order between repetitions. Errors are averaged over trials. All factorizations succeeded and all outputs were finite. The normalized residual $\|\mathbf{A}\widehat{\mathbf{S}} - \mathbf{I}\|_F/\|\mathbf{I}\|_F$ is approximately 0.011 for the CMul and `fp32`-GEMM variants, versus 8.4×10^{-6}–1.63×10^{-5} for `cuSOLVER`. The nearly identical errors of the custom variants indicate that replacing these GEMMs alone does not remove the error from their shared storage, recursive-inverse, and assembly stages.

Table 4 isolates the benefit of CMul: it is 1.8–3.3× faster than strict `fp32` GEMMs with nearly identical inverse-solution error. This speed–accuracy tradeoff motivates CMul as the default in all experiments using our solver, except the explicit GEMM ablations. Direct `fp16` GEMMs reduce runtime by a further 12–29%, but increase solution error by 9.7–10.6×. This variant uses only high-part operands in Schur updates and casts panel operands to `fp16` before multiplication; accumulation and diagonal operations remain `fp32`. Factor storage, triangular inversion, and assembly are unchanged. CMul and direct `fp16` GEMMs were measured together; the `fp32` and library rows use a separate run on the same 10 inputs and follow the same timing protocol. All added trials returned finite outputs with successful factorizations.

Table 4: GEMM precision ablation and `cuSOLVER` comparison on RTX 5090. Results averaged over 10 random matrices; error is relative solution error in %, using the same inputs and protocol as Table 1. Ours denotes the isolated linear solver. Bold marks the lowest value per size and metric.

n	Method	Runtime (ms)	Error (%)
1024	`cuSOLVER`	0.909	**0.00076**
1024	Ours (`fp16` GEMMs)	**0.214**	0.80212
1024	Ours (`fp32` GEMMs)	0.448	0.07556
1024	Ours (CMul)	0.244	0.07554
2048	`cuSOLVER`	2.268	**0.00105**
2048	Ours (`fp16` GEMMs)	**0.495**	0.81862
2048	Ours (`fp32` GEMMs)	1.398	0.08127
2048	Ours (CMul)	0.592	0.08129
4096	`cuSOLVER`	7.067	**0.00148**
4096	Ours (`fp16` GEMMs)	**1.496**	0.83036
4096	Ours (`fp32` GEMMs)	5.455	0.08442
4096	Ours (CMul)	1.889	0.08444
6144	`cuSOLVER`	18.292	**0.00109**
6144	Ours (`fp16` GEMMs)	**3.498**	0.82804
6144	Ours (`fp32` GEMMs)	13.841	0.08529
6144	Ours (CMul)	4.481	0.08529
8192	`cuSOLVER`	34.344	**0.00110**
8192	Ours (`fp16` GEMMs)	**6.363**	0.83386
8192	Ours (`fp32` GEMMs)	27.979	0.08581
8192	Ours (CMul)	8.601	0.08583
12288	`cuSOLVER`	98.602	**0.00113**
12288	Ours (`fp16` GEMMs)	**18.027**	0.83675
12288	Ours (`fp32` GEMMs)	80.502	0.08654
12288	Ours (CMul)	25.215	0.08655

Table 5: RTX 4090 reproduction of Table 4, with the same inputs and protocol. The `fp16`-GEMM and CMul rows were measured together in an additional run; the other rows were remeasured on the same 10 inputs. Entries are mean ± sample standard deviation across 10 random matrices. Each matrix is timed 20 times before computing its mean runtime. Bold marks the lowest mean per size and metric.

n	Method	Runtime (ms)	Error (%)
1024	`cuSOLVER`	0.768 ± 0.042	**0.00075 ± 0.00001**
1024	Ours (`fp16` GEMMs)	**0.248 ± 0.001**	0.79564 ± 0.01212
1024	Ours (`fp32` GEMMs)	0.480 ± 0.027	0.07548 ± 0.00070
1024	Ours (CMul)	0.274 ± 0.000	0.07543 ± 0.00081
2048	`cuSOLVER`	1.989 ± 0.002	**0.00105 ± 0.00001**
2048	Ours (`fp16` GEMMs)	**0.542 ± 0.026**	0.82080 ± 0.00901
2048	Ours (`fp32` GEMMs)	1.463 ± 0.001	0.08103 ± 0.00054
2048	Ours (CMul)	0.627 ± 0.031	0.08106 ± 0.00053
4096	`cuSOLVER`	8.355 ± 0.008	**0.00149 ± 0.00002**
4096	Ours (`fp16` GEMMs)	**1.683 ± 0.001**	0.82763 ± 0.00493
4096	Ours (`fp32` GEMMs)	6.150 ± 0.005	0.08420 ± 0.00031
4096	Ours (CMul)	2.075 ± 0.002	0.08419 ± 0.00032
6144	`cuSOLVER`	22.574 ± 0.052	**0.00164 ± 0.00001**
6144	Ours (`fp16` GEMMs)	**4.075 ± 0.002**	0.83183 ± 0.00386
6144	Ours (`fp32` GEMMs)	16.607 ± 0.027	0.08535 ± 0.00050
6144	Ours (CMul)	5.167 ± 0.002	0.08538 ± 0.00052
8192	`cuSOLVER`	44.390 ± 0.128	**0.00157 ± 0.00001**
8192	Ours (`fp16` GEMMs)	**7.765 ± 0.003**	0.83389 ± 0.00539
8192	Ours (`fp32` GEMMs)	36.650 ± 0.214	0.08587 ± 0.00021
8192	Ours (CMul)	10.256 ± 0.003	0.08589 ± 0.00024
12288	`cuSOLVER`	132.338 ± 0.365	**0.00112 ± 0.00000**
12288	Ours (`fp16` GEMMs)	**23.835 ± 0.017**	0.83676 ± 0.00269
12288	Ours (`fp32` GEMMs)	107.867 ± 0.414	0.08658 ± 0.00024
12288	Ours (CMul)	40.997 ± 0.012	0.08660 ± 0.00024

C.2 Inverse-root benchmarks and solver comparison

At $n = 2048$, a 5-pole ZPoles approximation reaches relative error 3.22×10^{-4} in 1.04 ms, below the 4-pole error at this size. We scan 4–14 CN iterations in both `fp16` and `fp32` on the same fixed matrix. The fastest tested CN configuration meeting that error threshold uses 11 `fp32` iterations: 12.44 ms with 2.78×10^{-4} error. Each runtime averages 20 timed repetitions after warmup; this accuracy-matched result concerns one input and the tested scan.

C.3 ZPoles with different linear solvers

We use the damped Gaussian construction of Section 5.3 in a separate single-matrix run, replacing only the shifted inverse backend of ZPoles. Both implementations use 4 poles, identical coefficient banks, Frobenius scaling, damping, and Gaussian inputs. The library implementation batches all 4 poles for one input matrix in `fp32`, using `torch.linalg.cholesky_ex` and `torch.cholesky_solve` with the `cuSOLVER` backend. Our implementation uses the mixed-precision solver of Section 3.2. Timing includes normalization, bank lookup, shifted inverses, weighted assembly, and scale restoration; offline coefficient construction and input Gram formation are excluded.

C.4 Numerical pipeline and provenance

The root wrappers receive the covariance directly, without forming a second Gram matrix. The wrapper normalizes the covariance, adds the relative ridge, computes its Frobenius norm, and selects a covering coefficient-bank row before calling the shared solver. For an eigenvalue interval $[a, b]$, the existing coefficient API is called with endpoints $[\sqrt{a}, \sqrt{b}]$ and response exponent $1 + 2q$; its

Table 6: Complete ZPoles inverse-root computation on RTX 5090 with our solver or `cuSOLVER`. Each size uses one fixed matrix, processed individually; runtime is the mean of 20 timed repetitions after warmup. Error is measured on that matrix. Error is $100\|\widehat{\mathbf{Z}} - \mathbf{A}^q\|_F / \|\mathbf{A}^q\|_F$, using an `fp64` eigendecomposition reference. Bold marks the lower value for each power, size, and metric.

n	Runtime (ms)		Error (%)	
	`cuSOLVER`	Ours	`cuSOLVER`	Ours
Inverse square root: $\mathbf{A}^{-1/2}$				
1024	3.237	**0.322**	**0.0987**	0.1091
2048	9.656	**0.990**	**0.1252**	0.1355
4096	36.026	**4.632**	**0.1558**	0.1638
6144	97.060	**13.910**	**0.1751**	0.1821
8192	181.928	**29.341**	**0.1918**	0.1980
12288	533.823	**107.161**	**0.2148**	0.2206
Inverse fourth root: $\mathbf{A}^{-1/4}$				
1024	3.237	**0.318**	**0.0654**	0.0725
2048	9.649	**0.989**	**0.0836**	0.0894
4096	36.556	**4.626**	**0.1057**	0.1101
6144	97.205	**13.945**	**0.1278**	0.1314
8192	182.456	**29.461**	**0.1324**	0.1356
12288	535.555	**109.139**	**0.1585**	0.1622

multiplier is then an approximation to t^q. The API argument `p` counts poles in these legacy calls; the paper consistently uses k for this count and q for the inverse-root exponent.

Schur outputs and diagonal factorization/inversion use `fp32`. In the implementation used for the main measurements, a stored panel has two `fp16` parts; its products retain the high–high term and both high–residual cross terms, with scale 4096. Residual–residual terms are omitted. The recursive inverse consumes only the high factor and the diagonal inverses. The diagonal kernel stores both an `fp32` inverse for compensated panel multiplication and an `fp16` copy for recursive inversion. Both merge intermediates and the nonconstant combined inverse-root operator are stored in `fp16`; the wrapper's scaling and constant correction return `fp32`. The distinction is important for ports to other hardware.

A factor-storage ablation retains compensation for $\mathbf{P}_j\mathbf{D}_j^\top$ but uses only `fp16` factor blocks in Schur updates. An RTX 5090 control with 4 poles and one matrix per call changes runtime from 0.226/0.952/4.531 ms to 0.214/0.826/3.614 ms at $n = 768/2048/4096$, respectively. Approximation errors change from 0.0696/0.0893/0.1101% to 0.0844/0.1039/0.1224%. Each size uses one fixed matrix and 20 timed CUDA-graph repetitions; residual buffers are still written but not read. The main algorithm, tables, figure, and training results retain factor compensation. On three damped Gaussian inputs at $n = 768/2048$, dropping factor residuals increases relative inverse-solution error from 0.0725/0.0815% to 0.3457/0.3701%, despite the smaller changes in inverse-fourth-root error above.

The Gaussian experiment in Figure 3 uses one fixed random matrix per size and 20 timed repetitions after warmup. Explicit half-precision CN uses `fp16` operands with `fp32` accumulation; full-precision CN uses `fp32` matrix multiplications. The offline coefficient bank covers condition bounds up to 2^{24}. Coefficient preparation, compilation, and graph construction are excluded from steady-state timing for both methods; input-dependent scaling and root assembly are included.

Matrix benchmark protocol. A separate matrix control at $n = 768$ gives 0.226 ms and 0.0701% error for ZPoles-4, versus 0.466 ms and 0.2724% for CN-10 with `fp16` operands. This uses the same damped Gaussian construction with one fixed matrix and 20 timed repetitions, not sampled training covariances.

The primary measurements use RTX 5090 GPUs. Each size uses one fixed SPD matrix, processed individually. The dimensions are $n \in \{1024, 2048, 4096, 6144, 8192, 12288\}$. We sample $\mathbf{X} \in$

$\mathbb{R}^{n\times n}$ with independent standard Gaussian entries, without row or column scaling. Set $\mathbf{A} = \mathbf{X}\mathbf{X}^\top + \epsilon\mathbf{I}$, with $\epsilon = 0.01\,\mathrm{tr}(\mathbf{X}\mathbf{X}^\top)/n$; the target is $\mathbf{A}^q$. The reference uses `fp64` eigendecomposition. Approximation error is $\|\widehat{\mathbf{Z}} - \mathbf{A}^q\|_F / \|\mathbf{A}^q\|_F$.

The inverse-fourth-root comparison uses one random matrix per size, two warmup calls, and 20 timed repetitions per method. Method order is randomized; timing is synchronized wall-clock time. All routines include input-dependent scaling and final root assembly; the offline coefficient bank is excluded. CN uses 10 coupled-Newton iterations with the initialization used by Anil et al. (2020), $z = 5/(2\|\mathbf{A}\|_F)$, $\mathbf{M}_0 = z\mathbf{A}$, and $\mathbf{X}_0 = z^{1/4}\mathbf{I}$. There are no per-iteration stopping checks. CN is tested with explicit `fp16` operands and `fp32` accumulation, and separately with full `fp32` matrix multiplications. ZPoles retains its mixed-precision solver. We also use DASH's released NDB routine (Modoranu et al., 2026): two successive 10-iteration stages compute an inverse fourth root. Its multiple-vector power-iteration scaling is included in runtime. Both `fp16` and `fp32` variants are tested; implementation details appear in Appendix C.4. These are fixed-budget comparisons.

Iterative baseline implementations. The inverse-square-root panels use one fixed Gaussian matrix per size with the same damping and `fp64` reference construction. NS-5 applies the five Polar Express coefficient triples in Gram form, tracking a multiplier $\mathbf{Q}$ and the transformed Gram matrix. Every two iterations it recomputes the latter as $\mathbf{Q}\mathbf{A}_0\mathbf{Q}^\top$, where $\mathbf{A}_0$ is the trace-normalized damped input. All products use `fp16` operands and `fp32` outputs in Figure 3(a). Tables 8 and 10 additionally test `bf16` NS operands, keeping the coefficients, input matrix, and accumulation precision unchanged. The returned multiplier is rescaled to approximate $\mathbf{A}^{-1/2}$; no output normalization is applied. This is an inverse-root adaptation, not the released Gram-NS polar-factor API. ZPoles uses the compensated factor backend and Frobenius interval bound. Table 8 lists the measurements in Figure 3(a). Method order is shuffled per input; timings average 20 synchronized CUDA-graph executions, including scaling and root assembly.

CN error grows partly because Frobenius scaling shrinks the initial eigenvalues, increasing the iterations needed at a fixed budget; damping does not remove this scaling effect. NDB uses a spectral estimate instead.

We use DASH's released NDB inverse-root code (Modoranu et al., 2026), with 10 iterations per stage. The fourth root requires two stages: the first computes a square root, and the second its inverse square root. Each stage saves two matrix multiplications at initialization, giving 56 square matrix multiplications in total. The multiple-vector scaling routine uses 16 random vectors, 10 power iterations in `bf16`, and twice the largest estimated Rayleigh quotient. The second stage uses the square root of the first scaling factor. Scaling, precision conversions, and both stages are included in runtime.

The released `fp16` variant forms its first matrix product in `fp32`, then stores the two iterates in `fp16`; subsequent products return `fp32` accumulations before rounding the iterates. The `fp32` variant retains full-precision iterates and products. Both use the released `bf16` scaling routine. We supply the same damped matrix to every method and disable NDB's additional root damping. Its power-iteration safeguard remains at the released default and changes only the matrix used to estimate the scale. For CUDA-graph capture, an index vector is created directly on the GPU instead of copying it from the CPU; no numerical operation is changed. This comparison concerns inverse-root routines, not the complete DASH optimizer. At $n = 12288$, full `fp32` NDB reaches 0.153% error versus ZPoles' 0.162%, but takes 3.54 s rather than 112 ms. These are fixed iteration budgets; methods share the input and their order is shuffled before timing.

Speedup ratios use unrounded runtime means. At $n = 1024$, CN and ZPoles take 0.809770 and 0.315934 ms, respectively, giving $2.5631\times$ (reported as $2.6\times$).

C.5 Pole-count ablation

We vary the pole count while keeping the compensated solver, Frobenius interval, damping, and Gaussian construction used in Figure 3 fixed. Both exponents use the same fixed matrix per size, processed individually. Figure 4 reports runtime and relative Frobenius error against `fp64` spectral references. Increasing the pole count initially reduces error, but eventually arithmetic error limits the benefit: 8 poles do not uniformly improve on 6.

Table 7: Inverse-fourth-root measurements for Figure 3, with additional `fp32` controls. Each size uses one fixed random matrix, processed individually on an RTX 5090. Runtime is mean ± sample standard deviation over 20 timed repetitions after warmup; error is measured on that matrix. ZPoles uses 4 poles, CN 10 iterations, and NDB two 10-iteration stages. Precision details appear above.

n	Method	Runtime (ms)	Approximation error (%)
1024	ZPoles	**0.32 ± 0.01**	**0.0725**
1024	CN (`fp16`)	0.81 ± 0.00	0.3497
1024	CN (`fp32`)	1.89 ± 0.00	0.2473
1024	NDB (`fp16`)	1.34 ± 0.00	0.2431
1024	NDB (`fp32`)	2.94 ± 0.01	0.1703
2048	ZPoles	**0.95 ± 0.00**	**0.0894**
2048	CN (`fp16`)	4.89 ± 0.00	0.7420
2048	CN (`fp32`)	11.54 ± 0.31	0.6763
2048	NDB (`fp16`)	6.91 ± 0.01	0.2158
2048	NDB (`fp32`)	16.41 ± 0.51	0.1610
4096	ZPoles	**4.58 ± 0.01**	**0.1101**
4096	CN (`fp16`)	32.75 ± 0.13	1.5551
4096	CN (`fp32`)	93.17 ± 0.37	1.4812
4096	NDB (`fp16`)	43.99 ± 0.31	0.2086
4096	NDB (`fp32`)	131.47 ± 0.33	0.1561
6144	ZPoles	**13.85 ± 0.05**	**0.1314**
6144	CN (`fp16`)	107.91 ± 0.24	2.2736
6144	CN (`fp32`)	312.95 ± 0.69	2.1695
6144	NDB (`fp16`)	147.88 ± 0.43	0.2028
6144	NDB (`fp32`)	437.77 ± 1.98	0.1508
8192	ZPoles	**30.97 ± 0.57**	**0.1356**
8192	CN (`fp16`)	252.12 ± 0.13	2.8575
8192	CN (`fp32`)	738.20 ± 1.32	2.7551
8192	NDB (`fp16`)	361.02 ± 0.35	0.2027
8192	NDB (`fp32`)	1024.14 ± 6.71	0.1547
12288	ZPoles	**112.34 ± 0.48**	0.1622
12288	CN (`fp16`)	811.67 ± 0.29	3.9027
12288	CN (`fp32`)	2507.04 ± 13.23	3.7333
12288	NDB (`fp16`)	1184.46 ± 0.97	0.2170
12288	NDB (`fp32`)	3537.29 ± 1.60	**0.1530**

C.6 RTX 4090 REPRODUCTION

We repeat the RTX 5090 comparisons in Tables 4, 7, and 8 on a local RTX 4090 with 24 GiB memory. The matrix constructions, damping, pole counts, iteration budgets, and operand formats are unchanged. The solver comparison uses 10 independent matrices, each timed for 20 replays. Both root comparisons use one fixed matrix per size, processed individually and timed 20 times after warmup. ZPoles root routines use 4 poles. We retain the measured solver sources and the same CN, NDB, and Gram-form NS routines without hardware-specific retuning. CUDA graph preparation and coefficient construction are excluded; normalization and output assembly remain timed. All methods within each experiment run sequentially on the same GPU. At $n = 12288$, the `fp64` reference is constructed one matrix at a time and retained in host memory to fit the device. Reference construction and error calculation are outside the timed region. These measurements test portability of the implementation, not clock-matched performance across GPU models.

On this GPU, the compensated solver is 2.80–4.37× faster than `cuSOLVER`, at higher solution error. For inverse fourth roots, ZPoles is 2.74–7.83× faster than the `fp16` CN budget. For inverse square roots, it is 1.61–4.18× faster than Gram-form NS with `fp16` operands. The tables report all precision controls; these fixed-budget comparisons do not establish dominance at every accuracy requirement.

NDB's `fp16` error increases more on the RTX 4090 at large sizes, whereas its `fp32` errors remain close to the RTX 5090 values. The released implementation stores its iterates in `fp16` and uses

Table 8: Inverse-square-root comparison on RTX 5090. Each size uses one fixed random matrix, processed individually. Runtime is mean ± sample standard deviation over 20 timed repetitions after warmup; error is measured on that matrix. ZPoles uses 4 poles. NS uses 5 Polar Express iterations in Gram form, with the indicated operand precision and `fp32` accumulation. Error is $100\|\widehat{\mathbf{Z}} - \mathbf{A}^{-1/2}\|_F / \|\mathbf{A}^{-1/2}\|_F$. Bold marks the lowest value per size and metric.

n	Method	Runtime (ms)	Error (%)
1024	ZPoles	**0.32 ± 0.00**	**0.1098**
1024	NS (`fp16`)	0.46 ± 0.00	7.4864
1024	NS (`bf16`)	0.46 ± 0.00	8.2524
2048	ZPoles	**0.98 ± 0.00**	**0.1347**
2048	NS (`fp16`)	2.51 ± 0.00	10.2334
2048	NS (`bf16`)	2.50 ± 0.00	10.3859
4096	ZPoles	**4.58 ± 0.01**	**0.1640**
4096	NS (`fp16`)	16.94 ± 0.04	8.5546
4096	NS (`bf16`)	16.97 ± 0.39	7.9642
6144	ZPoles	**13.83 ± 0.02**	**0.1821**
6144	NS (`fp16`)	51.77 ± 0.20	10.2371
6144	NS (`bf16`)	51.66 ± 0.67	9.0733
8192	ZPoles	**29.69 ± 0.34**	**0.1978**
8192	NS (`fp16`)	114.28 ± 0.21	14.2575
8192	NS (`bf16`)	114.13 ± 0.30	13.2375
12288	ZPoles	**109.79 ± 0.30**	**0.2205**
12288	NS (`fp16`)	359.51 ± 0.87	21.9178
12288	NS (`bf16`)	361.10 ± 0.41	20.6669

`bf16` random probes for scaling. In an additional fixed-input $n = 8192$ check, three probe seeds give errors of 0.2646–0.2652%; disabling reduced-precision GEMM reduction leaves these results unchanged. These controls do not explain the cross-device gap. The machines use CUDA 13.2 and 13.0, respectively; a same-input, same-scale cross-device trace is needed to separate hardware and library effects. We report the measured discrepancy rather than assuming bitwise portability.

All speedup ratios use unrounded runtime means.

C.7 Fractional-power transform benchmarks

We compare ZPoles with DynMuon's NS–Taylor transform at fixed exponents; the training schedule and implementation are specified in Appendix E.

Transform runtime and approximation error. Table 11 compares the same negative-power implementations on Gaussian matrices, using the frozen training coefficients and identical input normalization and scale restoration. The reference is $\mathbf{U}\mathbf{\Sigma}^p\mathbf{V}^\top$, computed through an `fp64` eigendecomposition of $\mathbf{X}\mathbf{X}^\top$. Across these shapes and exponents, ZPoles has 0.043–0.052% error, compared with 15.0–56.5% for NS–Taylor; changing NS operands from `bf16` to `fp16` barely changes the latter error. ZPoles is 1.18–$2.26\times$ faster than the `fp16` NS control. The reference retains `fp32` Taylor operations in both controls, following the released implementation. We use PyTorch's `high` float32 matrix-multiplication setting, matching the training configuration; this permits reduced-precision Tensor Core products for the Taylor operations. This is not a comparison of fully half-precision implementations.

Table 12 repeats this protocol on an RTX 5090 with the same frozen coefficients and implementations. ZPoles has 0.044–0.052% error and is 1.34–$2.35\times$ faster than the `fp16` NS–Taylor control.

Taylor truncation and output scale. For $\widetilde{\mathbf{X}} = \mathbf{X}/\|\mathbf{X}\|_F$ and $\mathbf{A} = \widetilde{\mathbf{X}}\widetilde{\mathbf{X}}^\top$, DynMuon's correction approximates $\mathbf{A}^{p/2}$ around the identity (Wu et al., 2026). Its scalar response is

$$c_p(\lambda) = 1 + \frac{p}{2}(\lambda - 1) + \frac{p(p-2)}{8}(\lambda - 1)^2. \tag{17}$$

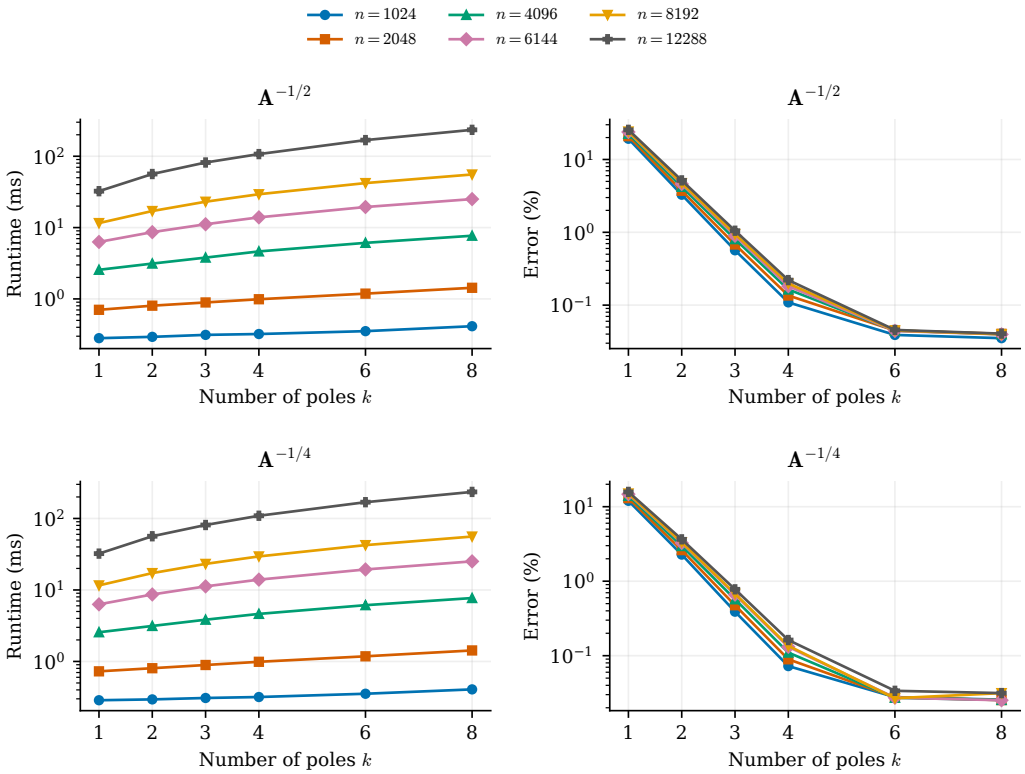


Figure 4: Runtime and error versus pole count for inverse square roots (top) and inverse fourth roots (bottom). Each curve uses $n \times n$ damped Gaussian Gram matrices constructed as in Section 5.3. Runtime points average 20 timed repetitions after warmup; errors are measured on the fixed input at each size. Runtime includes scaling and root assembly; error is $100\|\widehat{\mathbf{Z}} - \mathbf{A}^q\|_F / \|\mathbf{A}^q\|_F$ against an `fp64` spectral reference. Vertical axes use logarithmic scales.

However, $\mathrm{tr}(\mathbf{A}) = 1$, giving an average eigenvalue $1/n$. For the tested wide Gaussian inputs, the eigenvalues are therefore far from the expansion point 1. For example, at $p = -1/4$ and $\lambda = 1/1024$, the exact correction is $\lambda^{-1/8} \approx 2.378$, whereas $c_{-1/4}(\lambda) \approx 1.195$. A mildly negative exponent alone does not ensure a small truncation error on this normalized spectrum.

Table 13 isolates this effect on the first Gaussian input at each shape. Replacing NS with the exact polar factor still leaves 48.5–56.6% error, identifying Taylor truncation as the main source of the discrepancy in this test. To distinguish magnitude from direction, let $\mathbf{Y}_\star = \mathbf{U}\mathbf{\Sigma}^p\mathbf{V}^\top$ and fit a scalar to the released output $\widehat{\mathbf{Y}}$:

$$c_\star = \frac{\langle \widehat{\mathbf{Y}}, \mathbf{Y}_\star \rangle_F}{\|\widehat{\mathbf{Y}}\|_F^2}, \qquad e_{\mathrm{scaled}} = \frac{\|c_\star \widehat{\mathbf{Y}} - \mathbf{Y}_\star\|_F}{\|\mathbf{Y}_\star\|_F}. \tag{18}$$

This reduces the error to 10.7–11.1%: much of the raw discrepancy is an overall update-magnitude difference. The fitted scalar uses the exact reference and is only a diagnostic; it is not applied in the runtime or training comparisons. At $n = 512$, the `fp64` Gram-based reference agrees with a direct SVD to relative error 6.7×10^{-14}.

DynMuon's comparison with exact SVD reports similar validation-loss trajectories, rather than a matrix approximation-error bound (Wu et al., 2026). A large transform error can thus coexist with useful training behavior. These Gaussian diagnostics do not establish that ZPoles' smaller approximation error causes its training-loss improvement, and they do not characterize all training momentum spectra.

Table 9: RTX 4090 inverse-fourth-root reproduction of Table 7. One fixed matrix per size; runtime is mean ± sample standard deviation over 20 timed repetitions after warmup. Error is measured on that matrix. CN uses 10 iterations; NDB uses two 10-iteration stages. Error is the relative Frobenius distance to the `fp64` reference, in percent.

n	Method	Runtime (ms)	Error (%)
1024	ZPoles	**0.371 ± 0.001**	**0.07271**
1024	CN (`fp32`)	1.899 ± 0.001	0.25321
1024	CN (`fp16`)	1.015 ± 0.002	0.35258
1024	NDB (`fp32`)	2.980 ± 0.006	0.16690
1024	NDB (`fp16`)	1.538 ± 0.003	0.21836
2048	CN (`fp32`)	13.757 ± 0.205	0.67720
2048	NDB (`fp32`)	19.419 ± 0.293	0.16138
2048	ZPoles	**1.120 ± 0.002**	**0.08917**
2048	NDB (`fp16`)	7.377 ± 0.134	0.21495
2048	CN (`fp16`)	6.135 ± 0.016	0.73838
4096	CN (`fp16`)	47.660 ± 0.191	1.53833
4096	NDB (`fp16`)	61.068 ± 0.807	0.21260
4096	CN (`fp32`)	107.643 ± 0.307	1.48073
4096	NDB (`fp32`)	151.648 ± 0.400	0.15100
4096	ZPoles	**6.088 ± 0.005**	**0.10991**
6144	ZPoles	**22.376 ± 0.027**	**0.13120**
6144	NDB (`fp16`)	194.429 ± 1.512	0.20453
6144	NDB (`fp32`)	502.383 ± 0.801	0.15387
6144	CN (`fp16`)	144.004 ± 0.783	2.25044
6144	CN (`fp32`)	357.368 ± 0.317	2.16258
8192	NDB (`fp32`)	1166.529 ± 2.905	0.15071
8192	CN (`fp32`)	834.424 ± 1.580	2.75780
8192	CN (`fp16`)	327.182 ± 1.848	2.95504
8192	NDB (`fp16`)	453.710 ± 3.156	0.26549
8192	ZPoles	**55.150 ± 0.076**	**0.13541**
12288	CN (`fp32`)	2840.576 ± 4.199	3.73797
12288	NDB (`fp32`)	4009.306 ± 3.092	**0.14992**
12288	NDB (`fp16`)	1479.557 ± 1.679	0.22368
12288	CN (`fp16`)	1039.687 ± 1.058	3.90122
12288	ZPoles	**251.179 ± 2.110**	0.16240

Table 10: Inverse-square-root comparison on RTX 4090. Each size uses one fixed random matrix, processed individually. Runtime is mean ± sample standard deviation over 20 timed repetitions after warmup; error is measured on that matrix. ZPoles uses 4 poles. NS uses 5 Polar Express iterations in Gram form, with the indicated operand precision and `fp32` accumulation. Error is $100\|\widehat{\mathbf{Z}} - \mathbf{A}^{-1/2}\|_F / \|\mathbf{A}^{-1/2}\|_F$. Bold marks the lowest value per size and metric.

n	Method	Runtime (ms)	Error (%)
1024	ZPoles	**0.37 ± 0.00**	**0.1087**
1024	NS (`fp16`)	0.60 ± 0.01	7.4793
1024	NS (`bf16`)	0.58 ± 0.00	8.2332
2048	ZPoles	**1.24 ± 0.01**	**0.1351**
2048	NS (`fp16`)	3.39 ± 0.00	10.2576
2048	NS (`bf16`)	3.20 ± 0.00	10.4655
4096	ZPoles	**6.09 ± 0.01**	**0.1633**
4096	NS (`fp16`)	25.49 ± 0.04	8.5683
4096	NS (`bf16`)	25.75 ± 0.05	7.9381
6144	ZPoles	**22.49 ± 0.05**	**0.1820**
6144	NS (`fp16`)	74.11 ± 0.08	10.2239
6144	NS (`bf16`)	74.34 ± 0.05	9.0581
8192	ZPoles	**55.49 ± 0.06**	**0.1974**
8192	NS (`fp16`)	163.05 ± 0.27	14.2677
8192	NS (`bf16`)	161.58 ± 0.56	13.2094
12288	ZPoles	**248.23 ± 2.91**	**0.2205**
12288	NS (`fp16`)	496.17 ± 1.41	21.9042
12288	NS (`bf16`)	506.64 ± 1.48	20.6682

		Runtime (ms)			Error (%)		
n	p	NS–Taylor `bf16`	NS–Taylor `fp16`	ZPoles	NS–Taylor `bf16`	NS–Taylor `fp16`	ZPoles
512	$-1/16$	0.209	0.198	**0.167**	14.995	14.991	**0.048**
512	$-1/8$	0.209	0.198	**0.167**	27.818	27.818	**0.047**
512	$-1/4$	0.210	0.198	**0.167**	48.549	48.550	**0.052**
1024	$-1/16$	0.606	0.596	**0.377**	16.693	16.661	**0.044**
1024	$-1/8$	0.614	0.596	**0.383**	30.756	30.732	**0.043**
1024	$-1/4$	0.613	0.588	**0.382**	52.666	52.651	**0.048**
2048	$-1/16$	3.394	3.377	**1.494**	18.396	18.409	**0.049**
2048	$-1/8$	3.393	3.386	**1.495**	33.614	33.627	**0.046**
2048	$-1/4$	3.395	3.388	**1.496**	56.529	56.539	**0.051**

Table 11: DynMuon negative-power transforms on an RTX 4090. Inputs have shape $n \times 2n$ with independent standard Gaussian entries. We average results over 10 random matrices per shape, timing each matrix 20 times after warmup. Error is $\|\widehat{\mathbf{Y}} - \mathbf{U}\mathbf{\Sigma}^p\mathbf{V}^\top\|_F / \|\mathbf{U}\mathbf{\Sigma}^p\mathbf{V}^\top\|_F$, using an `fp64` reference. NS–Taylor uses 5 NS iterations and the released second-order correction; precision labels refer to NS operands, while the Taylor operations retain `fp32`. ZPoles uses 4 poles. Runtime includes input normalization, the complete transform, and scale restoration.

n	p	Runtime (ms)			Error (%)		
		NS–Taylor `bf16`	NS–Taylor `fp16`	ZPoles	NS–Taylor `bf16`	NS–Taylor `fp16`	ZPoles
512	$-1/16$	0.280	0.281	**0.160**	14.989	14.981	**0.048**
512	$-1/8$	0.280	0.281	**0.159**	27.812	27.808	**0.047**
512	$-1/4$	0.281	0.281	**0.159**	48.543	48.541	**0.052**
1024	$-1/16$	0.469	0.471	**0.352**	16.687	16.660	**0.044**
1024	$-1/8$	0.470	0.471	**0.353**	30.752	30.733	**0.044**
1024	$-1/4$	0.470	0.472	**0.352**	52.666	52.654	**0.048**
2048	$-1/16$	2.821	2.833	**1.207**	18.398	18.409	**0.049**
2048	$-1/8$	2.821	2.831	**1.208**	33.616	33.627	**0.046**
2048	$-1/4$	2.820	2.830	**1.207**	56.530	56.539	**0.051**

Table 12: DynMuon negative-power transforms on an RTX 5090. Inputs have shape $n \times 2n$ with independent standard Gaussian entries. We average results over 10 random matrices per shape, timing each matrix 20 times after warmup. Error is $\|\widehat{\mathbf{Y}} - \mathbf{U}\mathbf{\Sigma}^p\mathbf{V}^\top\|_F / \|\mathbf{U}\mathbf{\Sigma}^p\mathbf{V}^\top\|_F$, using an `fp64` reference. NS–Taylor uses 5 NS iterations and the released second-order correction; precision labels refer to NS operands, while the Taylor operations retain `fp32`. ZPoles uses 4 poles. Runtime includes input normalization, the complete transform, and scale restoration.

n	Raw error (%)	Exact-polar error (%)	$c_\star$	Scaled error (%)
512	48.56	48.53	1.9073	11.07
1024	52.66	52.74	2.0771	10.75
2048	56.52	56.62	2.2628	10.82

Table 13: NS–Taylor error diagnostics at $p = -1/4$, using one Gaussian input of shape $n \times 2n$ per row. Raw error uses the released `bf16` NS implementation. Exact-polar error replaces NS with the exact polar factor and computes the Taylor response in `fp64`, isolating truncation error. Scaled error optimally rescales the released output using Equation (18). These diagnostics are separate from the 10-input means in Table 11.

D TRAINING PROTOCOL AND ADDITIONAL EXPERIMENTS

The main runs use GPT-2-Small/Medium (124M/355M parameters), paired initialization, and the same data order. Each update processes 32,768 tokens through microbatches of eight sequences of length 1024. The matrix learning rate is 0.002, momentum 0.95, and weight decay 0.1. Attention and MLP weight matrices use the compared optimizer. Token and positional embeddings (including the tied output head), biases, and LayerNorm parameters use AdamW with betas $(0.95, 0.95)$, $\epsilon = 10^{-8}$, and weight decay 0.1. AdamW uses the same base learning rate and schedule as the matrix optimizer, including in the learning-rate comparisons. The learning rate stays constant for the first 40% of training, then decreases linearly to one tenth of its initial value. The budgets are 30,510/30,517 updates for Small/Medium, approximately 1B tokens each. Intermediate validation uses 1,048,576 tokens; final validation uses the full held-out shard of approximately 100M tokens. Each configuration uses one seed. These runs do not provide a reliable estimate of seed variability. Per-step runtimes include forward, backward, and optimizer work, excluding validation and the first 100 updates. Runs use RTX 5090 cards; timings are not hardware-isolated alternating measurements.

Shampoo recipe and provenance. The two-sided inverse-fourth-root preconditioner follows Shampoo (Gupta et al., 2018). Blocking, exponential moving averages, momentum, and layerwise grafting follow the scalable Shampoo framework (Anil et al., 2020; Shi et al., 2023). Grafting preserves the preconditioned direction while matching its norm to a coordinate-wise adaptive update. We implement these components in our training framework; the recipe below is not an unchanged reproduction of a released Shampoo configuration. The decay coefficients, relative damping, block sizes, and additional dimension-dependent scaling are experimental choices.

Let $\mathbf{W}_t \in \mathbb{R}^{n\times m}$ be a parameter matrix and $\mathbf{G}_t$ its current gradient after the training loop's global gradient clipping. Partition $\mathbf{G}_t$ into blocks $\mathbf{G}_{t,b}$ of shape $b_n \times b_m$. Small uses blocks up to 768 per dimension; Medium uses blocks up to 512. For each block, initialize the statistics to zero and update

$$\mathbf{L}_{t,b} = 0.95\mathbf{L}_{t-1,b} + 0.05\mathbf{G}_{t,b}\mathbf{G}_{t,b}^\top/b_m, \qquad \mathbf{R}_{t,b} = 0.95\mathbf{R}_{t-1,b} + 0.05\mathbf{G}_{t,b}^\top\mathbf{G}_{t,b}/b_n.$$

Set $\epsilon_{L,t,b} = 0.01\,\mathrm{tr}(\mathbf{L}_{t,b})/b_n$ and $\epsilon_{R,t,b} = 0.01\,\mathrm{tr}(\mathbf{R}_{t,b})/b_m$. The root routine supplies approximations to $(\mathbf{L}_{t,b} + \epsilon_{L,t,b}\mathbf{I})^{-1/4}$ and $(\mathbf{R}_{t,b} + \epsilon_{R,t,b}\mathbf{I})^{-1/4}$, denoted $\widehat{\mathbf{P}}_{L,t,b}$ and $\widehat{\mathbf{P}}_{R,t,b}$. They act on the *current gradient block*, before momentum:

$$\mathbf{P}_{t,b} = \widehat{\mathbf{P}}_{L,t,b}\,\mathbf{G}_{t,b}\,\widehat{\mathbf{P}}_{R,t,b}.$$

Reassemble these blocks into $\mathbf{P}_t \in \mathbb{R}^{n\times m}$. For grafting, maintain a separate coordinate-wise second moment, initialized to zero, and form

$$\mathbf{V}_t = 0.95\mathbf{V}_{t-1} + 0.05\mathbf{G}_t \odot \mathbf{G}_t, \qquad \mathbf{J}_t = \frac{\mathbf{G}_t}{\sqrt{\mathbf{V}_t/(1-0.95^t)} + 10^{-8}}.$$

$$\mathbf{D}_t = \frac{\|\mathbf{J}_t\|_F}{\max(\|\mathbf{P}_t\|_F, 10^{-30})\sqrt{\max(n,m)}}\mathbf{P}_t.$$

Division and square root in $\mathbf{J}_t$ are entrywise. Grafting uses the whole parameter matrix's norm, not individual block norms. The additional dimension scaling is our recipe's choice, not a requirement of Shampoo or grafting.

Nesterov momentum follows preconditioning and grafting. With $\beta_1 = 0.95$ and $\mathbf{B}_0 = 0$, the implementation computes

$$\mathbf{B}_t = \beta_1\mathbf{B}_{t-1} + \mathbf{D}_t, \qquad \mathbf{U}_t = (1-\beta_1)(\mathbf{D}_t + \beta_1\mathbf{B}_t).$$

The factor $1-\beta_1 = 0.05$ converts the sum-based momentum to EMA scaling; $\mathbf{B}_t$ is distinct from the second moment $\mathbf{V}_t$. Finally, with scheduled learning rate η_t and decoupled weight decay $\lambda = 0.1$, the parameter update is

$$\mathbf{W}_{t+1} = (1-\eta_t\lambda)\mathbf{W}_t - \eta_t\sqrt{n/m}\,\mathbf{U}_t. \tag{19}$$

Here n, m are the stored parameter matrix's row and column counts. The factor $\sqrt{n/m}$ is retained from the training framework's matrix-update learning-rate scaling; it does not scale weight decay. Thus the complete order is gradient statistics, two-sided preconditioning, whole-matrix grafting and dimension scaling, Nesterov momentum, then the parameter update above. The main comparisons refresh roots every update. All root backends use the same rule; parameters assigned to the AdamW fallback follow the separate update described in the training setup.

Root backends in training. CN uses 10 iterations with `fp16` matrix-multiplication operands and `fp32` accumulation; its state and normalization are stored in `fp32`. NDB uses the released `fp32` routine, with two 10-iteration stages and multi-power-iteration scaling. ZPoles uses 4 poles and the mixed-precision routine in Section 3.2.

Selection and controls. The main Shampoo comparison uses exponent $q = -1/4$, ridge 0.01, and learning rate 0.002 for every backend. All runs start from the same initialization and data order and refresh roots every update. Their step-time differences are observational, with the protocol above. Appendix D.3 reports additional full-budget learning-rate comparisons.

Scope of the training evidence. The same-frequency comparisons isolate the root routine within the tested grafted Shampoo optimizer. They do not establish superiority over a tuned Muon optimizer: the Shampoo runs have higher loss and step cost than the Muon rows in Table 2. Shampoo has one paired seed per configuration; a full-budget eigendecomposition run with fresh roots every step, warm-started CN, and per-backend tuning beyond the learning-rate grid are not included.

D.1 Completed Small/Medium comparisons

The curves below accompany the final losses in Table 2. Muon controls replace the momentum transform; Shampoo controls replace the inverse-root routine within the grafted optimizer described above. Comparisons across optimizer families also change the update rule. NS uses `bf16` operands; Gram-NS and ZPoles use `fp16` matrix multiplications. Appendix D.2 specifies the Muon learning-rate selection.

Failures and provenance. Both `fp16` NDB training runs terminated with nonfinite preconditioned gradients; the reported NDB results use `fp32` iterates and the released multi-power-iteration scaling. We did not identify the cause of NDB's higher training loss. Its comparable accuracy on the Gaussian matrix benchmarks does not establish comparable accuracy on the evolving training statistics or explain the observed loss gap. The completed Medium Gram-NS run uses pinned Quack/CUTLASS dependencies and direct kernel calls. All main-table ZPoles runs use 4 poles.

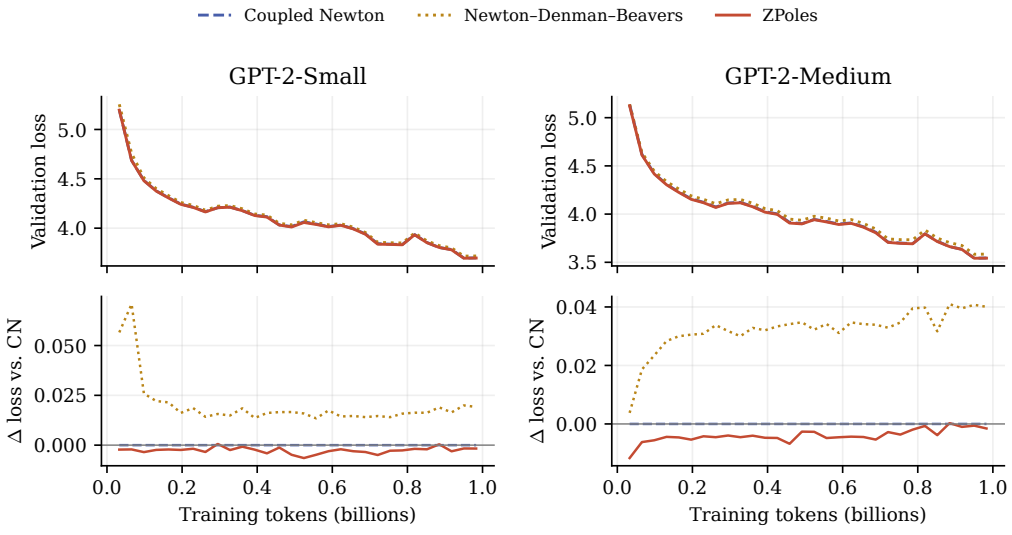


Figure 5: Validation loss during Shampoo training in Section 5.4. Left: GPT-2-Small; right: GPT-2-Medium. Curves show the 30 periodic measurements using 1,048,576 validation tokens, from 1,000 to 30,000 updates. Final full-shard losses appear in Table 2 and are not joined to these curves. All methods refresh roots every update. Bottom panels show paired loss differences relative to CN at the same checkpoint; negative values indicate lower loss.

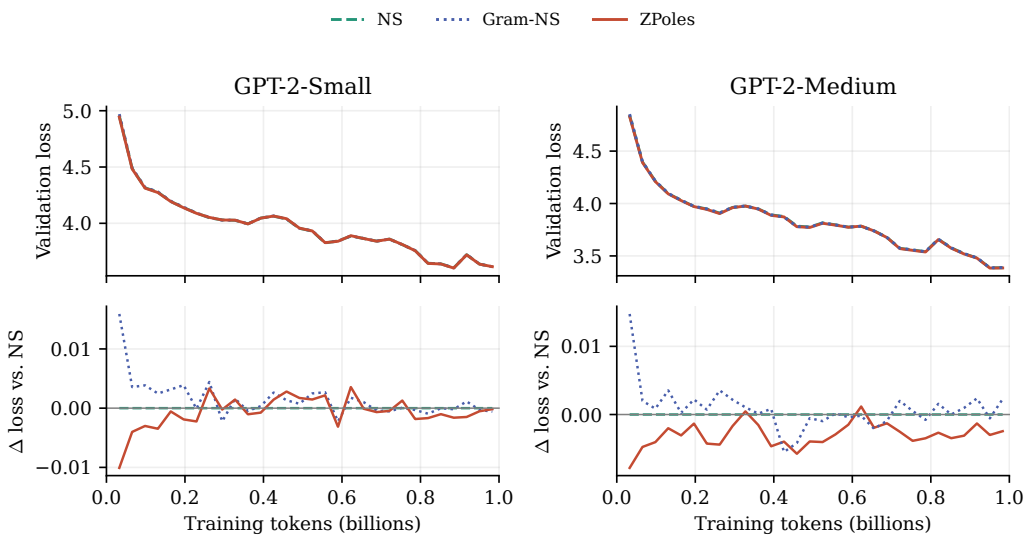


Figure 6: Validation loss during Muon training in Section 5.4. Left: GPT-2-Small; right: GPT-2-Medium. Curves show the 30 periodic measurements using 1,048,576 validation tokens, from 1,000 to 30,000 updates. Final full-shard losses appear in Table 2 and are not joined to these curves. All three curves use the learning-rate-search cohort; ZPoles uses 4 poles. Bottom panels show paired loss differences relative to NS at the same checkpoint; negative values indicate lower loss.

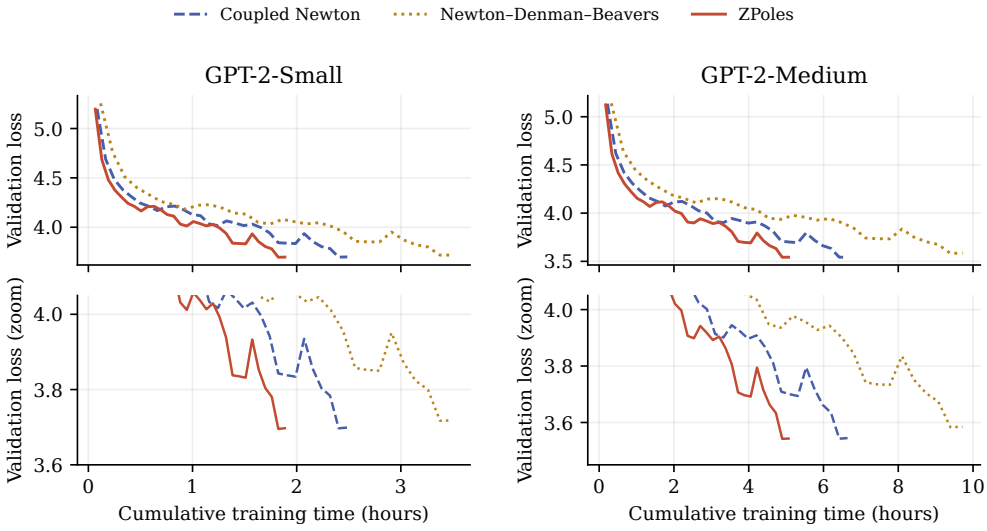


Figure 7: Shampoo validation loss versus cumulative recorded training-step time. Left: GPT-2-Small; right: GPT-2-Medium. Top panels show the full trajectory; bottom panels magnify late-training losses. Each time coordinate sums all recorded update times up to its validation checkpoint, including initial updates but excluding validation. Curves use the same periodic validation measurements as Figure 5; final full-shard losses are not joined. All methods refresh every update. Runs were timed separately on RTX 5090 GPUs; these are observed time–loss curves, not hardware-isolated timing comparisons.

D.2 MUON LEARNING-RATE SELECTION

Each method receives the same learning-rate grid, 0.001, 0.002, and 0.004, in 3,000-update GPT-2-Small pilots with shared initialization and data order. All select 0.002. Small resumes the selected checkpoint. The main Medium runs use the same selected rate from a shared initialization. Appendix D.3 subsequently compares three learning rates with approximately 1B tokens per run on both model sizes. Table 2 reports this complete cohort, with 4 poles for ZPoles. Small's runtime combines the pilot and continuation, excluding the first 100 updates overall. These separately run timings are not hardware-isolated alternating comparisons. The single-seed loss differences do not establish a reproducible advantage.

D.3 LEARNING-RATE COMPARISONS

We test maximum learning rates $\eta_{\max} \in \{0.001, 0.002, 0.005\}$ for every backend on GPT-2-Small and Medium (Table 14). Each run uses approximately 1B tokens, matching the main experiments. Within each optimizer and model size, initialization, data order, and all settings other than the backend and $\eta_{\max}$ are held fixed. The learning rate remains at $\eta_{\max}$ for the first 40% of training, then decreases linearly to $0.1\eta_{\max}$. All ZPoles runs use 4 poles.

For Shampoo, the backend ranking depends on the learning rate: CN gives lower loss at 0.001, whereas ZPoles gives lower loss at 0.002 and 0.005 on both models. Selecting each backend's best rate from this grid gives Small losses of 3.7099 for CN, 3.7288 for NDB, and 3.7049 for ZPoles. Thus the ZPoles–CN gap is 0.0050 after rate selection, compared with 0.0197 when both use 0.005. On Medium, all three backends perform best at 0.002, giving losses of 3.5593, 3.5983, and 3.5572, respectively. Two additional Medium runs at 0.01 give 3.7710 for NDB and 3.7115 for ZPoles; neither improves on 0.002.

For Muon, all backends perform best at 0.002 on both models. ZPoles gives slightly lower loss than NS and Gram-NS at every tested rate, but the differences remain small. At the best rate, the ZPoles–NS differences are 0.0003 on Small and 0.0016 on Medium.

For DynMuon, ZPoles gives lower loss than NS–Taylor at all three rates on both models. Both backends perform best at 0.002 on Small and 0.001 on Medium. At these selected rates, ZPoles reaches 3.5429 versus 3.5542 on Small and 3.3634 versus 3.3823 on Medium. The preferred learning rate therefore depends on the optimizer and model size; comparisons at a single shared rate need not represent each backend's best result.

Table 14: Validation loss after approximately 1B training tokens on GPT-2-Small/Medium. $\eta_{\max}$ is the maximum learning rate, held constant for the first 40% of training and then reduced linearly to $0.1\eta_{\max}$. All ZPoles runs use 4 poles. Bold marks the lowest loss within each optimizer, model, and learning-rate comparison.

		GPT-2-Small			GPT-2-Medium		
		$\eta_{\max}$	$\eta_{\max}$	$\eta_{\max}$	$\eta_{\max}$	$\eta_{\max}$	$\eta_{\max}$
Optimizer	Backend	0.001	0.002	0.005	0.001	0.002	0.005
Shampoo	CN	**3.8133**	3.7099	3.7246	**3.6755**	3.5593	3.5962
	NDB	3.8316	3.7288	3.7438	3.6885	3.5983	3.6541
	ZPoles	3.8197	**3.7090**	**3.7049**	3.6790	**3.5572**	**3.5869**
Muon	NS	3.6046	3.5821	3.6257	3.4217	3.4033	3.4810
	Gram-NS	3.6048	3.5828	3.6282	3.4209	3.4059	3.4817
	ZPoles	**3.6019**	**3.5818**	**3.6254**	**3.4196**	**3.4017**	**3.4762**
DynMuon	NS–Taylor	3.5730	3.5542	3.7045	3.3823	3.3860	3.6114
	ZPoles	**3.5465**	**3.5429**	**3.6766**	**3.3634**	**3.3832**	**3.5917**

E DYNAMICALLY SCHEDULED FRACTIONAL POWERS

The DynMuon comparison imports the released spectral strategy into the same GPT-2 training envelope as our other runs; it does not reproduce the authors' complete training recipe. All variants

share initialization, data order, seed, momentum, weight decay, and learning-rate schedule. Small and Medium train for 30,510 and 30,517 updates, respectively, with 32,768 tokens per update and learning rate 0.002. With update index t and total update count T, we use the logistic schedule $p(t) = -1/4 + (5/4)/(1 + \exp[(t/T - 0.02)/0.01])$. The released dispatch returns raw momentum for $p \geq 1/4$, a polar update for $0 \leq p < 1/4$, and the negative-power transform for $p < 0$. The reference computes the latter with NS and a second-order Taylor correction. ZPoles uses 4 poles in both nontrivial branches and retains the same input scaling and restoration of the input norm raised to p. Neither branch adds output RMS normalization.

The original reference uses `bf16` NS operands; a second control changes these to `fp16`, matching ZPoles. Coefficients for the negative regime are fitted offline at 33 response exponents $p \in [-1/4, 0]$ and 57 condition settings, then interpolated in log-coefficient space. This transform implementation uses a heuristic momentum-interval selector, unlike the covering interval for damped inverse roots in Algorithm 1. Its approximation accuracy on arbitrary momentum spectra is not certified by those covering bounds.

Table 2 reports final full-shard validation losses and mean runtime over all updates after the first 100, excluding validation. Runs use separate RTX 5090 GPUs; timing differences are observational. The matched-precision comparison removes the NS operand-format difference, but still changes both the approximation and its numerical implementation; it does not identify approximation accuracy as the cause of lower loss.

The additional `fp16` NS–Taylor control reaches validation loss 3.5560 at 195.8 ms/step on Small and 3.3865 at 549.2 ms/step on Medium. Against this control, ZPoles reduces loss by 0.0131 and 0.0033 and observed runtime by 1.9% and 4.7%, respectively.

Figure 8 shows the corresponding loss trajectories.

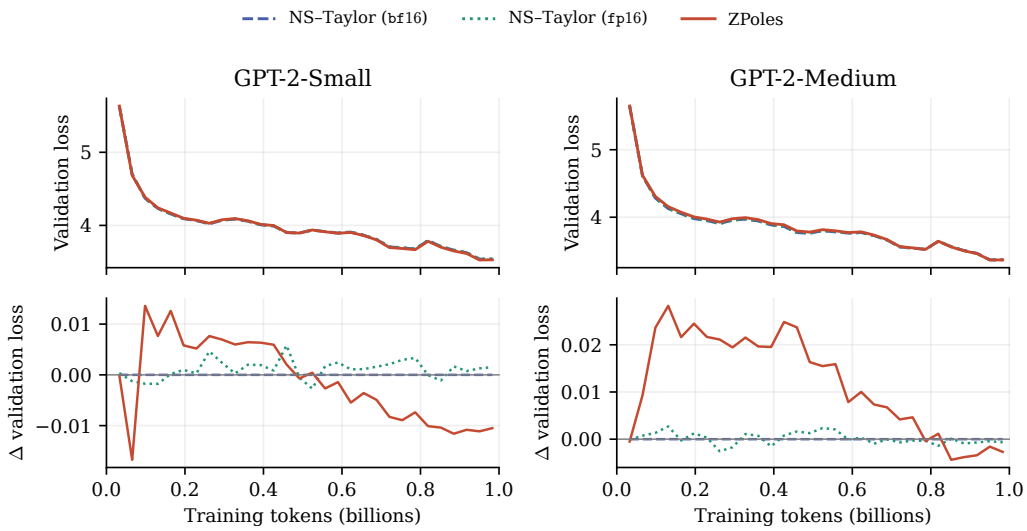


Figure 8: Dynamically scheduled fractional-power training with the runs reported in Table 2 and the additional matched-operand-precision control. Left: GPT-2-Small; right: GPT-2-Medium. The 30 periodic measurements each use 1,048,576 validation tokens; final full-shard losses are excluded from the curves. Bottom panels show paired loss differences from the released `bf16` NS–Taylor baseline. All runs share the exponent schedule, learning rate, initialization, and data order. These are single-seed trajectories, not estimates of seed variability.