DION3: FULL-STACK ORTHOGONAL UPDATES

Anonymous authors
Paper under double-blind review

ABSTRACT

The Muon optimizer incurs a significant overhead cost due to its cubic-time Newton-Schulz orthogonalization step. When weights are sharded, communication overhead compounds this computational cost, eroding the benefits of Muon in many settings. We present Dion3, a revision of Muon that targets this overhead at every level of the stack. Our Gram Newton-Schulz algorithm reduces the FLOP cost of orthogonalization, our CuteDSL kernels accelerate it by exploiting symmetry, and our megabatching strategy reduces communication overhead. Moreover, we propose a simple change to the update rule that cuts costs even further: selecting only a fraction of the momentum matrix's rows to orthogonalize at each step. This update rule improves on Dion (another "compressed" version of Muon), in both speed and performance (Ahn et al., 2025). Overall, Dion3 matches or improves on the loss achieved by Muon but reduces optimizer step time by up to $6\times$. Dion3 is available via the `dion` package as a drop-in replacement for Muon.

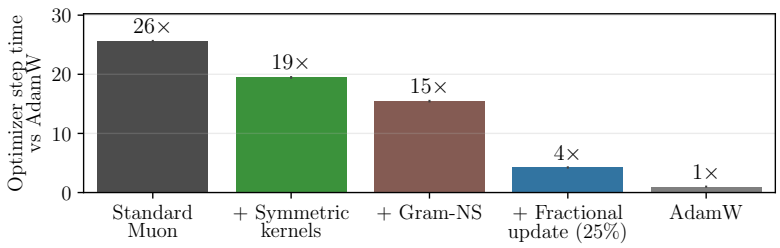


Figure 1: Optimizer step time of Muon (excluding forward/backward pass) relative to AdamW for a 7B-parameter language model trained on four GH200s. Each bar adds one of our contributions on top of the previous one. Together they reduce Muon's cost from $26\times$ AdamW to just $4\times$.

1 INTRODUCTION

Muon is becoming the method of choice for training frontier LLMs like Kimi K2 and GLM-5 (GLM-5 Team et al., 2026; Kimi Team et al., 2025). Compared to AdamW, Muon needs fewer optimizer steps to reach a given loss, but each step is more expensive. This overhead is due to Muon's Newton-Schulz orthogonalization procedure, a cubic-time matrix operation not present in older optimizers. As model size increases, the overhead of computing each Muon step grows rapidly. Moreover, Muon is more difficult than traditional optimizers to parallelize, introducing additional communication overhead in distributed settings. Thus, scaling Muon presents a range of algorithmic and systems-level challenges that diminish its effectiveness in the most demanding settings and hinder its adoption to new problems.

This paper gives a comprehensive answer to these challenges. We provide a full-stack solution that performs well across a wide range of model sizes, architectures, cluster sizes, and parallelism strategies. We package our contributions into a single optimizer, Dion3, built on four improvements that each reduce the cost of Muon's orthogonalization step and compound when used together:

1. **Gram Newton-Schulz**, a mathematically equivalent reformulation of Newton-Schulz that iterates on the small symmetric Gram matrix and cuts the FLOP cost of each orthogonalization dramatically (Section 3).

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.

2. Custom **GPU kernels for symmetric matrix multiplication** written in CuteDSL, which speed up Gram Newton-Schulz and take maximal advantage of its symmetric structure (Section 4).
3. A new **optimizer update rule** that subsamples the rows or columns of the momentum matrix before orthogonalizing it to make each step faster (Section 5). Our update rule is simpler and faster than Dion (Ahn et al., 2025) while matching or improving the optimization quality of Muon and NorMuon.
4. **Megabatched communication**, which eliminates a significant source of overhead by reducing the number of rounds of communication per optimizer step to a small constant.

Our improvements combine to accelerate the optimizer step at every level of the stack. The GPU kernels at the lowest level speed up Gram Newton-Schulz, which in turn reduces the FLOP cost of executing our optimizer's update rule. At the top level, the compression achieved by our update rule joins with the megabatch strategy to limit the communication cost. Each of our contributions is of independent value, but combining them gives Dion3 maximum flexibility to handle different parallelism strategies, architectures, and model sizes efficiently. Moreover, Dion3 benefits from synergies between them that further reduce computational cost. Gram Newton-Schulz uses more symmetric multiplications than standard Newton-Schulz does, compounding the benefits of our CuteDSL kernels. Likewise, Gram Newton-Schulz is especially fast for matrices with highly asymmetric shapes—like those produced by our update rule's subsampling strategy. Surprisingly, Dion3 appears to benefit the update *quality* as well as the computational footprint in some settings, even with a strong optimized baseline.

To realize these gains in practice, we provide open-source implementations in the form of two interoperable packages: `gram-newton-schulz` is a drop-in replacement for Muon's Newton-Schulz routine that also incorporates our CuteDSL kernels, while `dion` implements our optimizer update rule (along with baselines like Muon and NorMuon), and handles parallelism in the distributed setting (megabatching, FSDP, DDP, etc.). In all, Dion3 allows practitioners working in a wide range of settings to realize the benefits of Muon and related optimizers with almost no overhead.

2 The Challenge of Scaling Muon

2.1 Muon and NorMuon Recap

The Muon optimizer (Jordan et al., 2024) is best described as steepest-direction descent with respect to the spectral norm (Bernstein, 2025). At a given training step, let $\boldsymbol{W} \in \mathbb{R}^{n \times m}$ be a weight matrix and let $\boldsymbol{G}$ be the gradient of the loss with respect to $\boldsymbol{W}$. The Muon update rule is

$$\boldsymbol{M} \leftarrow \mu \boldsymbol{M} + \boldsymbol{G}, \qquad \boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \operatorname{polar}(\boldsymbol{M}) \tag{1}$$

where μ is the momentum coefficient, η is the learning rate, and $\boldsymbol{M}$ is the momentum matrix (with $\boldsymbol{M}_0 := \boldsymbol{0}$). In many ways, Muon resembles basic stochastic gradient descent (SGD) with momentum. Its key innovation is the polar operation, which is defined as follows:

Definition 1 (Polar Decomposition). *If $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$ is the singular value decomposition (SVD) of a matrix, then* $\operatorname{polar}(\boldsymbol{X}) = \boldsymbol{U}\boldsymbol{V}^\top$.

This operation—the "orthogonalization step"—balances the spectrum of the update, ensuring that it has full numerical rank. NorMuon (Li et al., 2026) is a popular variant of Muon that aims to combine its strengths with those of Adam. The cost of NorMuon's extra normalization steps is negligible compared to that of computing $\operatorname{polar}(\boldsymbol{M})$, so NorMuon's benefits come almost for free. Our methods, which speed up the computation of $\operatorname{polar}(\boldsymbol{M})$, apply to Muon and NorMuon alike.

2.2 Standard Newton-Schulz

The main challenge to scaling Muon is the need to compute $\operatorname{polar}(\boldsymbol{M})$ for each weight matrix at every step. Since $\operatorname{polar}(\cdot)$ is expensive to compute exactly, Muon uses an iterative method called Newton-Schulz to approximate it. Beginning with $\boldsymbol{X}_0 = \boldsymbol{M}/\|\boldsymbol{M}\|_{\mathsf{F}}$, each iteration improves the approximation $\boldsymbol{X}_t \approx \operatorname{polar}(\boldsymbol{M})$ according to the update rule

$$\boldsymbol{X}_{t+1} = a_t \boldsymbol{X}_t + b_t \boldsymbol{X}_t \boldsymbol{X}_t^\top \boldsymbol{X}_t + c_t \left(\boldsymbol{X}_t \boldsymbol{X}_t^\top\right)^2 \boldsymbol{X}_t. \tag{2}$$

We can interpret (2) via the SVD. Let $\boldsymbol{X}_t = \boldsymbol{U}\boldsymbol{\Sigma}_t\boldsymbol{V}^\top$. Each iteration of (2) preserves the singular vectors $\boldsymbol{U}$ and $\boldsymbol{V}$ and transforms the singular values according to the polynomial $p_t(x) = a_t x + b_t x^3 + c_t x^5$. The initialization ensures all singular values of $\boldsymbol{X}_0$ lie in $[0, 1]$, and Jordan et al. (2024) identified a sequence of polynomials for which $(p_T \circ \cdots \circ p_1)(x) \approx 1$ on $[0, 1]$. Therefore,

$$\boldsymbol{X}_T = \boldsymbol{U}\left(p_T \circ \cdots \circ p_1\right)\Big(\boldsymbol{\Sigma}_0\Big)\,\boldsymbol{V}^\top \approx \boldsymbol{U}\boldsymbol{V}^\top =: \operatorname{polar}(\boldsymbol{M}).$$

Algorithm 1 gives the standard implementation of Newton-Schulz. We now analyze its runtime. Each iteration does three matrix multiplications ($\boldsymbol{X}\boldsymbol{X}^\top$, $\boldsymbol{A}^2$, and $\boldsymbol{B}\boldsymbol{X}$) costing $2mn^2$, $2n^3$, and $2mn^2$ FLOPs, respectively. We ignore the other operations, which cost $O(mn)$. Let T be the number of iterations and let $\alpha := m/n$ be the aspect ratio of $\boldsymbol{X}$ ($\alpha = 1$ when X is square and $\alpha \gg 1$ when it is very rectangular). The total cost of Newton-Schulz is $T(4mn^2 + 2n^3) = 2T(2\alpha + 1)n^3$ FLOPs.

Algorithm 1
Standard Newton-Schulz

Input: $\boldsymbol{X} \in \mathbb{R}^{n\times m}$ with $n \le m$, coefficients $\{(a_t, b_t, c_t)\}_{t=1}^5$
// Normalize sing. vals. to $[0, 1]$
$\boldsymbol{X} \leftarrow \boldsymbol{X} \,/\, (\|\boldsymbol{X}\|_{\mathsf{F}} + 10^{-7})$
// Use half precision for speed
$\boldsymbol{X} \leftarrow \texttt{bfloat16}(\boldsymbol{X})$
for $t = 1, \ldots, 5$ **do**
  // Update $\boldsymbol{X} \leftarrow p_t(\boldsymbol{X})$
  $\boldsymbol{A} \leftarrow \boldsymbol{X}\boldsymbol{X}^\top$
  $\boldsymbol{B} \leftarrow b_t\boldsymbol{A} + c_t\boldsymbol{A}^2$
  $\boldsymbol{X} \leftarrow a_t\boldsymbol{X} + \boldsymbol{B}\boldsymbol{X}$
end for
return $\boldsymbol{X}$

While Muon comfortably outperforms traditional optimizers like Adam on a per-step basis, Newton-Schulz makes it harder to scale for two reasons:

- **Super-linear complexity.** SGD and Adam only do element-wise operations (addition, scalar multiplication), so their runtimes scale linearly with the number of parameters. In contrast, orthogonalizing an $n \times n$ weight takes $O(n^3)$ time.
- **Distributed training.** When weights are sharded across several GPUs, element-wise optimizers like Adam simply update each shard separately. Muon must gather the shards of $\boldsymbol{M}$ together to compute $\operatorname{polar}(\boldsymbol{M})$, incurring a large communication overhead.

Our runtime analysis above reveals two further shortcomings of standard Newton-Schulz:

- **Symmetric Matrix Multiplication.** The matrices $\boldsymbol{A} = \boldsymbol{X}\boldsymbol{X}^\top$ and $\boldsymbol{B} = b_t\boldsymbol{A} + c_t\boldsymbol{A}^2$ computed at each iteration are symmetric by definition, but standard Newton-Schulz does not exploit this structure. Eliminating the redundant computations would save half the FLOP cost of these multiplications, reducing the overall cost to $T(3\alpha + 1)n^3$ FLOPs.
- **Dependence on Aspect Ratio.** Newton-Schulz's runtime is dominated by the large rectangular matrix multiplications needed to compute $\boldsymbol{X}\boldsymbol{X}^\top$ and $\boldsymbol{B}\boldsymbol{X}$, which together cost $3\alpha n^3$ FLOPs per iteration even when exploiting symmetry. A typical implementation with $T = 5$ requires 10 of these expensive rectangular multiplications. This strong dependence on α is unfortunate, because most weight matrices in transformer architectures are rectangular. The latest MoE architectures are trending towards finer-grained, sparser experts, whose weights have even higher α (Guo et al., 2026; Kimi Team et al., 2025; OpenAI et al., 2025; Yang et al., 2025). Thus, we want our orthogonalization method to avoid expensive rectangular multiplications.

2.3 Prior Work on Scaling Muon

Appendix A surveys previous work that scales Muon or attempts to remedy its high overhead costs. To summarize, Muon has been scaled successfully (e.g., by Kimi Team et al. 2025), but it has relied on a brittle alignment of architectural, parallelism, and framework factors. Hardly any work has addressed directly the four issues described above. By shrinking the size of the orthogonalized matrix, Ahn et al. (2025) *did* address them and inspired several subsequent papers; our update rule simplifies their approach. Some parts of Gram Newton-Schulz and symmetric multiplication kernels have appeared before, but their potential as applied to Muon has not been fully realized.

3 Gram Newton-Schulz

Our first contribution is an orthogonalization algorithm that uses fewer rectangular matrix multiplications than standard Newton-Schulz. Instead of iterating directly on the large input matrix $\boldsymbol{X}$, we

iterate on the small symmetric Gram matrix $\boldsymbol{X}\boldsymbol{X}^\top$. The output of our algorithm is mathematically identical to that of standard Newton-Schulz, but it is significantly cheaper to compute.

At a high level, our strategy is based on the following formula. If $\boldsymbol{X} \in \mathbb{R}^{n\times m}$ with $n \le m$, then $\mathrm{polar}(\boldsymbol{X}) = (\boldsymbol{X}\boldsymbol{X}^\top)^{-1/2}\boldsymbol{X}$. Rather than use an iterative method to approximate $\boldsymbol{X}_T \approx \mathrm{polar}(\boldsymbol{X})$ directly, we instead **(1)** compute the $n \times n$ Gram matrix $\boldsymbol{X}\boldsymbol{X}^\top$; **(2)** use an iterative method to approximate $\boldsymbol{Q}_T \approx (\boldsymbol{X}\boldsymbol{X}^\top)^{-1/2}$; and **(3)** output $\boldsymbol{Q}_T\boldsymbol{X}$. Because it is based on the Gram matrix, we call this strategy "Gram Newton-Schulz" (GNS). Step 2, the core of the algorithm, works entirely with small $n \times n$ symmetric matrices. Thus, GNS requires just two rectangular matrix multiplications: $\boldsymbol{X}\boldsymbol{X}^\top$ in the beginning and $\boldsymbol{Q}_T\boldsymbol{X}$ at the end. GNS also synergizes well with our symmetric GEMM kernels (Section 4), since almost all of its multiplications are symmetric.

How do we implement step 2? It turns out that any sequence of polynomials designed for *standard* Newton-Schulz can be adapted for this purpose. Formally, GNS is based on the following theorem, which is the scalar analogue of the three-step strategy described above: **(1)** $r_0 = x^2$, **(2)** compute q_T from r_0 via a polynomial iteration, **(3)** output $q_T x$. Theorem 2 guarantees that this process gives the same result as applying standard Newton-Schulz to x directly. For the proof, see Appendix B.

Theorem 2. *Let $p_1, \dots, p_T$ be a sequence of odd polynomials with $p_t(x) = xh_t(x^2)$. Define* $\begin{pmatrix} r_0 \\ q_0 \end{pmatrix} = \begin{pmatrix} x^2 \\ 1 \end{pmatrix}$ *and* $\begin{pmatrix} r_t \\ q_t \end{pmatrix} = \begin{pmatrix} r_{t-1}h_t(r_{t-1})^2 \\ q_{t-1}h_t(r_{t-1}) \end{pmatrix}$ *for $t \in \{1, \dots, T\}$. Then $q_T x = (p_T \circ \cdots \circ p_1)(x)$.*

To obtain our initial version of Gram Newton-Schulz (Algorithm 2), we simply lift the iteration of Theorem 2 from scalars to matrices. All operations are matrix polynomials which preserve the singular vectors of $\boldsymbol{X}$. Each singular value of $\boldsymbol{R}_t$ and $\boldsymbol{Q}_t$ evolves independently according to the scalar iteration, so the output of Algorithm 2 equals that of standard Newton-Schulz. In effect, GNS computes the final iterate of standard Newton-Schulz, $\boldsymbol{X}_T$, without ever constructing the intermediate iterates $\boldsymbol{X}_1, \dots, \boldsymbol{X}_{T-1}$.

If we use symmetric matrix multiplications (and skip computing $\boldsymbol{R}_5$), the total cost of Algorithm 2 is $(4T + 3\alpha - 3)n^3$ FLOPs. Compare this to standard Newton-Schulz's $T(3\alpha + 1)n^3$ FLOPs when using symmetric GEMMs. For $\alpha = 1$, they are equal.[1] For $\alpha > 1$, GNS is cheaper, often significantly so. In a typical Muon application ($T = 5, \alpha = 4$), GNS saves **55%** of the FLOPs used by standard Newton-Schulz with symmetric GEMMs, or **68%** compared to a typical implementation without symmetric GEMMs. For larger T and α, the savings are even greater, as the leading order term is $O((T + \alpha)n^3)$ instead of $O(T\alpha n^3)$.

Algorithm 2
Naive Gram Newton-Schulz

Input: $\boldsymbol{X} \in \mathbb{R}^{n\times m}$ with $n \le m$, coefficients $\{(a_t, b_t, c_t)\}_{t=1}^5$

// Normalize sing. vals. to $[0, 1]$

$\boldsymbol{X} \leftarrow \boldsymbol{X} \,/\, (\|\boldsymbol{X}\|_{\mathsf{F}} + 10^{-7})$

$\boldsymbol{R}_0 = \boldsymbol{X}\boldsymbol{X}^\top$

$\boldsymbol{Q}_0 = \boldsymbol{I}$

for $t = 1, \dots, 5$ **do**

  // $Z_t = h_t(\boldsymbol{R}_{t-1})$

  $\boldsymbol{Z}_t \leftarrow a_t\boldsymbol{I} + b_t\boldsymbol{R}_{t-1} + c_t\boldsymbol{R}_{t-1}^2$

  $\boldsymbol{Q}_t \leftarrow \boldsymbol{Q}_{t-1}\boldsymbol{Z}_t$

  $\boldsymbol{R}_t \leftarrow \boldsymbol{Z}_t\boldsymbol{R}_{t-1}\boldsymbol{Z}_t$

end for

return $\boldsymbol{Q}_5\boldsymbol{X}$

Stabilizing Gram Newton-Schulz As written, Algorithm 2 destabilizes Muon and can even diverge. The leading cause of this instability is the presence of spurious negative eigenvalues in the Gram matrix $\boldsymbol{R}_0 = \boldsymbol{X}\boldsymbol{X}^\top$ due to rounding errors in half-precision arithmetic. In theory, Gram matrices do not have negative eigenvalues. Theorem 2's trick of adapting polynomials from standard Newton-Schulz to compute $(\boldsymbol{R}_0)^{-1/2}$ only works when $r_0 = x^2$; otherwise, it diverges to infinity. We provide a thorough theoretical analysis and numerical experiments demonstrating this phenomenon in Appendix B.1.

In practice, we can fully mitigate this instability using a restarting strategy. Instead of running all five iterations of the loop in a single pass and outputting $\boldsymbol{Q}_5\boldsymbol{X}$, we run only the first two iterations and compute $\boldsymbol{X}_2 = \boldsymbol{Q}_2\boldsymbol{X}$. We then "restart" the algorithm treating $\boldsymbol{X}_2$ as the new input, reinitializing $\boldsymbol{R}_2 = \boldsymbol{X}_2\boldsymbol{X}_2^\top$, $\boldsymbol{Q}_2 = \boldsymbol{I}$ and proceeding with $t = 3, 4, 5$. This resets any spurious negative eigenvalues to near zero and restores commutativity of $\boldsymbol{X}$, $\boldsymbol{Q}$, and $\boldsymbol{R}$ at the cost of $3(\alpha - 1)n^3$ FLOPs. Our stabilized, ready-to-use version of Gram Newton-Schulz is presented in Algorithm 4 of Appendix B and implemented in a pip-installable package called `gram-newton-schulz`.

[1] In practice, when $\alpha = 1$, we fall back to standard Newton-Schulz with our symmetric GEMMs (Appendix C.3), since it launches fewer GEMMs and has a faster wall clock time.

4 SYMMETRIC GEMM KERNELS IN CUTEDSL

Our second contribution is a set of custom GPU kernels for the operations $\boldsymbol{AB}$ and $\alpha\boldsymbol{AB} + \beta\boldsymbol{C}$ that assume $\boldsymbol{AB}$ and $\boldsymbol{C}$ are symmetric. These *symmetric GEMM* kernels compute the lower triangle of the output matrix in the usual way and copy the results to the upper triangle (Figure 2, left), saving about half the floating point operations used by standard matrix multiplication routines. Our kernels allow us to take advantage of the symmetric structure of Gram Newton-Schulz. As noted above, they also accelerate standard Newton-Schulz, but their impact is far greater when combined with Gram Newton-Schulz. We target the Hopper and Blackwell GPU architectures. Figure 2 (right) shows that our kernels achieve superb performance on both architectures, significantly outperforming the standard GEMM routine from cuBLAS across a range of matrix sizes.

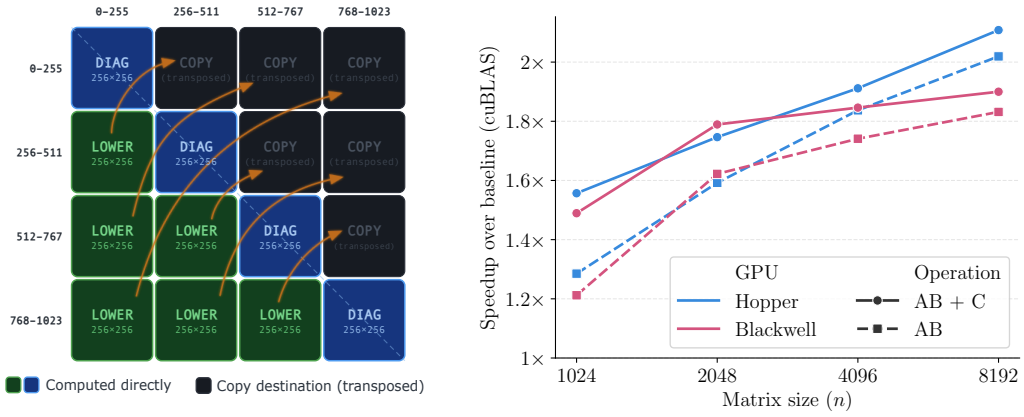


Figure 2: **Left:** Symmetric GEMM computes tiles from the lower triangle and main diagonal directly, then transposes and copies each lower tile to the corresponding upper tile. **Right:** Our CuteDSL symmetric GEMM kernels benchmarked against cuBLAS GEMM kernels on Hopper and Blackwell GPUs. Input matrices $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$ have dimensions $n \times n$. For large enough n, our kernels achieve a $\sim 2\times$ speedup over cuBLAS, both with and without an epilogue addition of $\boldsymbol{C}$.

Most general matrix multiplication (GEMM) kernels have the following form:

1. Scheduler: The output matrix is divided into tiles and each is assigned to a group of workers.
2. Computing each output tile of $\boldsymbol{AB}$ or $\alpha\boldsymbol{AB} + \beta\boldsymbol{C}$ requires three steps:
   (a) Prologue: The rows of $\boldsymbol{A}$ and columns of $\boldsymbol{B}$ needed for the current tile are loaded in from general memory (high-bandwidth memory) to shared memory (SRAM).
   (b) Matrix-Multiply Accumulate: The rows and columns are multiplied and written to the register file (Hopper) or tensor memory (Blackwell).
   (c) Epilogue: Additional tensors needed for fused operations (e.g. $\boldsymbol{C}$, α, β) are loaded, fused operations are executed, and the final output is written from the register file to shared memory and then to general memory.

Our symmetric GEMM kernels differ from the standard one only in their schedulers and epilogues. In the standard GEMM, the entire output matrix is partitioned into work tiles that are load balanced and evenly partitioned amongst clusters of thread blocks. Our "triangular scheduler" is almost identical, except that it only partitions work tiles from the lower triangle of the matrix (including the main diagonal); those in the upper triangle are left unassigned. This ensures that clusters are load balanced and no unnecessary work is performed. In the epilogue, when tiles from the lower triangle (excluding the main diagonal) are written to their assigned tile in general memory, they are also written to their transposed tile location in the upper triangle. The left panel of Figure 2 illustrates this process. We describe the details of our symmetric kernels and of our kernelized implementation of standard Newton-Schulz in Appendix C.

5 THE DION3 UPDATE RULE

Our third contribution is a variant of Muon that avoids orthogonalizing the full momentum matrix, making the optimizer even faster. At each step, we select only a fraction of the rows from the momentum matrix and perform the Muon update (including the orthogonalization) as if the other rows did not exist. In words, the main steps of the Dion3 update rule are as follows:

1. **Select** a fraction of the rows (or columns) of the momentum matrix.
2. **Orthogonalize** the selected submatrix using Gram Newton-Schulz.
3. **Update** the selected rows of the weight matrix using the output of the previous step. Do not update the other rows.
4. **Error Feedback** Decay *only* the selected rows of the momentum matrix by a multiplicative factor.

Unlike our other contributions, this procedure changes the optimization trajectory, but experiments in Section 7.1 show that this change is benign. When *all* rows are selected (i.e., the fraction equals 1), our update rule reduces to Muon in exact arithmetic. We now describe each step of the optimizer in detail. Algorithm 3 gives the full pseudocode, and Appendix D discusses some important implementation details.

Algorithm 3 Dion3 update rule (single weight matrix)

Input: Weight $\boldsymbol{W} \in \mathbb{R}^{n\times m}$, gradient $\boldsymbol{G}$, momentum buffer $\boldsymbol{M}$, row-wise 2nd moment buffer $\boldsymbol{v} \in \mathbb{R}^n$, fraction $f \in (0, 1]$, momentum μ, decay β_2, learning rate η.

$\boldsymbol{M} \leftarrow \boldsymbol{M} + \boldsymbol{G}$
$\mathcal{S} \leftarrow$ indices of the $k = \lceil fn \rceil$ rows of $\boldsymbol{M}$ with largest ℓ_1 norm ▷ Selection
$\boldsymbol{O} \leftarrow \mathrm{polar}(\boldsymbol{M}[\mathcal{S}, :])$ ▷ Orthogonalize $k \times m$ submatrix
$\boldsymbol{v}_i \leftarrow \beta_2 \boldsymbol{v}_i + (1-\beta_2)\frac{1}{m}\sum_j \boldsymbol{O}_{ij}^2$ for $i \in \mathcal{S}$ ▷ Optional: NorMuon steps...
$\widehat{\boldsymbol{O}}_{ij} \leftarrow \boldsymbol{O}_{ij}/(\sqrt{\boldsymbol{v}_i} + \epsilon)$
$\widehat{\boldsymbol{O}} \leftarrow \widehat{\boldsymbol{O}}\, \|\boldsymbol{O}\|_{\mathsf{F}}/\|\widehat{\boldsymbol{O}}\|_{\mathsf{F}}$
$\boldsymbol{W} \leftarrow (1 - \eta \cdot \mathtt{wd})\boldsymbol{W}$ ▷ Optional: weight decay
$\boldsymbol{W}_{ij} \leftarrow \boldsymbol{W}_{ij} - \eta\, \widehat{\boldsymbol{O}}_{ij}$ for $i \in \mathcal{S}$ ▷ Update selected rows
$\boldsymbol{M}_{ij} \leftarrow \mu\, \boldsymbol{M}_{ij}$ for $i \in \mathcal{S}$ ▷ Error Feedback

Selection Our update rule introduces a new hyperparameter $f \in (0, 1]$ that controls the fraction of rows (or columns) to select. We think of f as a compression factor; $f = 1$ corresponds to Muon, and decreasing f speeds up the algorithm. We recommend $f = 1/4$ or $f = 1/8$. We could subselect either the rows or the columns. When the weight matrix is sharded across one of these dimensions, we subselect along that same dimension. Otherwise, we pick whichever dimension is smaller. For brevity, we refer only to row selection throughout this section. We use a simple selection strategy: we pick the $k = \lceil fn \rceil$ rows of largest ℓ_1 norm. Initial experiments showed that optimization quality when using random selection was not much worse, so other selection strategies may perform well too. In the distributed setting, we select the top-f fraction of rows from each shard independently to avoid extra rounds of synchronization. True global selection is available as an option in our package.

Orthogonalization The benefit of our selection strategy is realized in this step. For an input of size $n \times m$, each symmetric matrix multiplication in Newton-Schulz costs either mn^2 or n^3 FLOPs. Selection reduces the dimensions to $fn \times m$, slashing the cost of these multiplications by a factor of at least $1/f^2$. This benefit is compounded by using Gram Newton-Schulz, which is faster than standard Newton-Schulz precisely when the aspect ratio $\alpha = m/n$ is large. Happily, selection increases α by a factor of $1/f$. Our optimizer also benefits the *communication* cost of Muon in the distributed setting. Rather than assemble the entire momentum matrix on a single device to perform the orthogonalization, we only assemble the selected rows, reducing the communication volume by a factor of $1/f$. We describe our communication strategy in detail in Section 6.

Weight Update We experiment with two flavors of the update rule: Muon and NorMuon. For the Muon flavor, we use the orthogonalized submatrix $\boldsymbol{O} = \mathrm{polar}(\boldsymbol{M}[\mathcal{S}, :])$ to update the matching

submatrix of the weight: $\boldsymbol{W}[\mathcal{S},:] \leftarrow \boldsymbol{W}[\mathcal{S},:] - \eta \boldsymbol{O}$. The NorMuon flavor is similar, but with extra steps that update the row-wise second moments and use them to rescale $\boldsymbol{O}$; see Algorithm 3. If weight decay is enabled, it is applied to all rows of $\boldsymbol{W}$ before updating the selected rows.

Error Feedback We adapt the *error feedback* mechanism from Dion (Ahn et al., 2025) to our simpler selection-based strategy for approximating $\boldsymbol{M}$. The usual rule for updating the momentum matrix is $\boldsymbol{M} \leftarrow \boldsymbol{M} + \boldsymbol{G}$; $\boldsymbol{M} \leftarrow \mu \boldsymbol{M}$, which ensures that $\boldsymbol{M}$ is an exponential moving average of the gradients across iterations. Error feedback decays *only* the component of $\boldsymbol{M}$ that was selected: $\boldsymbol{M}[\mathcal{S},:] \leftarrow \mu \boldsymbol{M}[\mathcal{S},:]$. Compared to standard momentum, this boosts the magnitude of the rows that were *not* selected, nudging future iterations to select them. Even if a given row of $\boldsymbol{G}$ is consistently small, error feedback allows it to build up in $\boldsymbol{M}$ over many iterations, so it eventually gets selected and participates in the update.

6 Megabatching and Communication

Our final contribution concerns the distributed setting. Under fully-sharded data parallelism (FSDP), each momentum matrix is partitioned across GPUs and must be assembled onto a single device before Newton-Schulz can run. The cost of these communications is a major obstacle to scaling Muon (Ahn et al., 2025; Essential AI, 2025). To help users easily adopt Dion3 in a wide range of scenarios, we release `dion`, a pip-installable package that gracefully manages this communication for FSDP2, DDP, and mixed sharding strategies with minimal overhead. It supports our GNS with symmetric GEMM kernels, the Dion3 update rule, Muon, NorMuon, and standard Newton-Schulz. We believe `dion` provides a full-stack, production-ready solution for distributed orthogonal optimization.

A key feature of our implementation is **megabatching**: we group all matrices of the same shape into a single batch, which we assemble and orthogonalize in parallel. Megabatching greatly reduces the number of *rounds* of communication per Muon step, since transformers contain only a few distinct weight shapes regardless of scale. This simple change can significantly reduce communication overhead. Appendix E describes the motivation and implementation of megabatching in detail and benchmarks its performance. It also outlines how Dion3 could enable further communication savings at very large scales by combining model parallelism and data parallelism.

7 Experiments

7.1 Model Quality Is Preserved

Of our four contributions, only our update rule explicitly changes the optimizer trajectory. Gram Newton-Schulz and our symmetric GEMM kernels do introduce numerical differences due to finite precision arithmetic; however, experiments in Appendix F.2 show that these differences do not affect training. We also confirmed that, as expected, Dion3 with selection fraction $f = 1$ matches Muon/NorMuon almost exactly (see Figure 23). In contrast, Dion3 with $f < 1$ is a genuinely new optimizer. In this section, we demonstrate that Dion3 achieves nearly the same loss as the baseline, and even a slightly better one.

Smaller-Scale Runs In our first experiment, we train dense transformers of various sizes on the FineWeb dataset (Lozhkov et al., 2024) using our `dion` package. We try four training budgets (2.5B, 5B, 10B, and 20B tokens), each with a learning rate cooldown in the last 25%. For further details of our architecture and setup, see Appendix G.1. Throughout this section, we compare NorMuon to the NorMuon flavor of Dion3, as initial experiments gave it a slight edge over Muon. We set the learning rate ($\eta = 0.01$) and momentum coefficient ($\mu = 0.95$) based on an initial sweep. Figure 3 shows the results. For all three model sizes, Dion3 with $f = 1/8$ lags behind NorMuon at the beginning of training, but eventually closes the gap. For the 225M model, Dion3 actually wins. For another perspective, we measure the "horizontal gap" between the convergence curves at different target loss values (right panel). E.g., for the 418M model, Dion3 takes 20% longer to achieve loss ≤ 3.2, but both optimizers reach loss $= 3.0$ at the same time. The horizontal gap decreases steadily with target loss (and drops to nearly zero during cooldown).

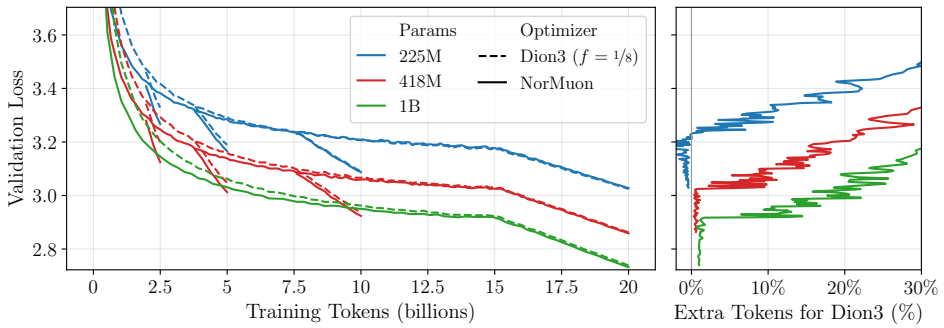


Figure 3: Training dense transformers on FineWeb with NorMuon and Dion3 at different model scales and training budgets. Right panel shows the "horizontal gap" between convergence curves for the longest (20B token) runs. For a given model and target loss value (e.g. green, 3.0), we note when each optimizer first attains that loss (6.3B vs 7.4B tokens) and plot the relative difference on the x-axis (17%). Dion3 lags in early training (equivalently, at high target loss) but eventually catches up.

Larger-Scale Runs The rest of this section describes larger-scale runs that use the ClimbMix dataset (Diao et al., 2025) and different training code (see Appendix G.1). We still use the `dion` package to implement the optimizers. For now, we train 1B-parameter models on 100B tokens. We retune NorMuon's learning rate ($\eta = 0.01$) and momentum ($\mu = 0.9$) for fairness. In this setting, we find that Dion3's optimal learning rate is higher than NorMuon's; it scales as $0.01/\sqrt{f}$. Appendix G.2 motivates this learning rate transfer rule theoretically and empirically. Figure 4 shows the validation loss curves for NorMuon and for Dion3 with various $f < 1$ and tuned η. Surprisingly, Dion3 consistently beats NorMuon throughout training, including after the cooldown phase.

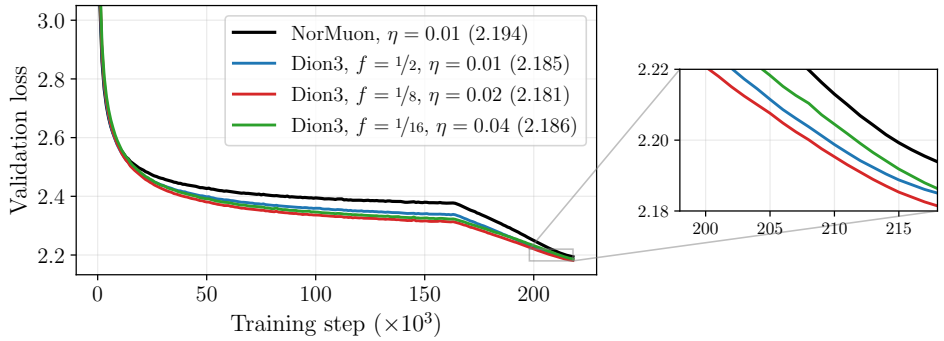


Figure 4: Validation loss curves for 1B-parameter models trained on 100B tokens of ClimbMix with NorMuon or Dion3. Each optimizer uses its best learning rate. Parentheses show final loss. All Dion3 variants beat the fully-tuned NorMuon baseline throughout training, finishing ≈ 0.01 points lower.

To strengthen this finding, we scale up the model size to between 3B and 14B parameters. For reasons of cost, we now train on 10B tokens. We compare Dion3 with $f = 1/4$ against NorMuon, each at their best learning rate from above. Table 4 reports final validation loss on ClimbMix and downstream accuracy on a suite of 12 standard benchmarks. In validation loss, Dion3 continues to beat NorMuon at every scale, achieving the largest improvement (-0.027) at the largest scale (14B). In downstream accuracy, it wins at three of the four scales.

7.2 Dion3 Accelerates the Optimizer

We now measure how much Dion3 reduces the cost of Muon's optimizer step. Starting with standard Muon, we successively add our improvements—symmetric kernels, Gram Newton-Schulz, and fractional updates with $f = 0.5$ or $f = 0.25$—stacking each on top of the previous ones. (All versions use megabatching.) We also compare against plain AdamW. We run each version of the

optimizer on models of various sizes either on 1 GPU or using FSDP on 4 GPUs. We use CUDA events to measure the GPU time of the optimizer step. For fidelity, we run full training steps with forward and backward passes (on synthetic data) but these are excluded from the recorded timings.

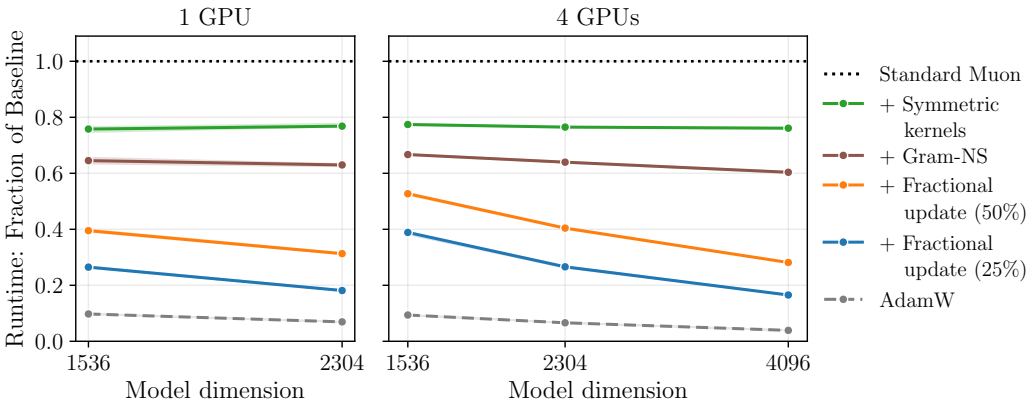


Figure 5: Optimizer step time (excluding forward/backward pass) relative to standard Muon across model scales when training on 1 GH200 (left) and FSDP over 4 GH200s (right). Each colored line adds one of our contributions on top of the previous one; AdamW is shown for reference. Lines show median over 25 steps and bands show interquartile range. Adding each of our contributions consistently reduces runtime; together they achieve a $6\times$ speedup for larger models.

Results are shown in Figure 5. A subset of these results appears in Figure 1, and the corresponding timings for the NorMuon family are given in Appendix G.5 (Figure 24). Each part of Dion3 yields a consistent speedup across scales and parallelism configurations. Our symmetric kernels and GNS give a combined speedup of $1.5\times$ or more over standard Muon. The fractional update rule cuts the runtime dramatically. As expected, its impact grows with model size, as the cubic computational cost of Newton-Schulz increasingly dominates the other operations. For sufficiently large models, setting $f = 1/2$ and $f = 1/4$ gives further $2\times$ and $3.7\times$ reductions respectively, for overall speedups of $3.6\times$ and $6.5\times$. Thus, while Dion3 is still slower than AdamW, the gap is *significantly* smaller.

In Appendix G.4, we report CPU time and communication volume in addition to GPU time, showing that CPU overhead is small and that fractional updates reduce communication by a factor of $1/f$. In Appendix F, we conduct an additional suite of benchmarks on a wider range of architectures (Gemma, MoE), including some whose weights have higher aspect ratios. For such architectures, GNS and the symmetric kernels alone achieve a speedup of $2\times$ (see Figures 19 and 20).

8 Conclusion

For LLMs trained with Muon, estimates suggest that optimizer step time accounts for between 1% (e.g., Abadji et al. (2026)) and 17% of total training time (see Appendix H). While Muon has been scaled successfully, doing so has demanded particular choices of architecture and parallelism strategy, along with *significant* engineering effort. As interest in Muon continues to spread, this paradigm is too brittle. A flexible, general-purpose remedy for Muon's overhead is needed.

Dion3 allows practitioners to easily realize the benefits of Muon without paying the high cost of its orthogonalization step, even for large-scale models in highly distributed settings. Our Gram Newton-Schulz algorithm and CuteDSL kernels speed up Muon by $1.5\times$ for dense models and $2\times$ for MoEs, a rare case of free lunch performance. Megabatching has an equally large effect in certain distributed settings. Dion3's fractional update rule ($f = 1/4$) provides an *additional* $3.7\times$ speedup. Though this update rule changes the optimizer trajectory, we find that it preserves and sometimes actually improves training quality. We did not expect such improvements when designing Dion3, but we note a similar recent finding of Joo et al. (2026): randomly masking blocks of the SGD update can improve the trajectory. Overall, Dion3 as implemented in our `dion` and `gram-newton-schulz` packages makes orthogonal optimizers practical and accessible in a wide range of settings.

AI USE STATEMENT

We used generative AI tools regularly for implementing and testing our methods, creating our packages, running experiments, organizing results, and creating figures. We used it occassionally for suggesting best practices, configurations, and ablations in LLM training and explaining expected results (e.g., the finite-precision issues described in Appendix C.1). We did not use AI to identify the research problem, to come up with our algorithms (GNS, update rule, ...), to prove theorems, or to write the text of the paper. In this project, we did not process or generate any datasets, propose new research methodologies, or use LLM as a judge, translator, or transcriber. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

As described in the main text, the primary contributions of this project are implemented in the `gram-newton-schulz` and `dion` packages. We provide these packages in the supplementary material. Most of the code needed to reproduce our experiments is also provided. Specifically:

1. Benchmarking GEMM kernels (Figure 2): Benchmarking scripts in gram-newton-schulz.
2. "Smaller-Scale Runs" from Section 7.1: `experiments/launch_size_budget.sh` in `dion` package.
3. "Larger-Scale Runs" from Section 7.1: These mix `dion` with a closed-source training pipeline, so the code is not released. But the sufficient details are provided in Appendix G.1 to reproduce them.
4. Timing results in Section 7.2: An alternative branch of `dion`, provided in a separate folder in the supplementary material (`dion-benchmark-branch`). Use the script `profile_training_step.py`.
5. Stability experiments of Appendices B.1 and B.2: Stand-alone Jupyter notebook `gram_newton_schulz_stability.ipynb` in supplementary material.
6. "Alternative experiments" of Appendix F: gram-newton-schulz package plus the details provided in Appendix G.1.

REFERENCES

Julien Abadji, Marah Abdin, Connor Adams, Eric Alcaide, Mustafa Altun, Michele Artoni, Junze Bao, Uday Barar, Vassilis Bekiaris, Arkadii Bessonov, et al. Laguna m.1/xs.2 technical report, 2026. URL https://arxiv.org/abs/2605.27605.

Kwangjun Ahn, Byron Xu, Natalie Abreu, Ying Fan, Gagik Magakyan, Pratyusha Sharma, Zheng Zhan, and John Langford. Dion: Distributed orthonormalized updates, 2025. URL https://arxiv.org/abs/2504.05295.

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 *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=yRtgZ1K8hO.

Jacob Austin, Sholto Douglas, Roy Frostig, Anselm Levskaya, Charlie Chen, Sharad Vikram, Federico Lebron, Peter Choy, Vinay Ramasesh, Albert Webson, and Reiner Pope. *How to Scale Your Model*. Google DeepMind, 2025. URL https://jax-ml.github.io/scaling-book/.

Jeremy Bernstein. Deriving Muon, 2025. URL https://jeremybernste.in/writing/deriving-muon.

Thibaut Boissin, Thomas Massena, Franck Mamalet, and Mathieu Serrurier. Turbo-muon: Accelerating orthogonality-based optimization with pre-conditioning, 2025. URL https://arxiv.org/abs/2512.04632.

Valentyn Boreiko, Zhiqi Bu, and Sheng Zha. Towards understanding of orthogonalization in muon. In *Tiny Titans: The next wave of On-Device Learning for Foundation Models (TTODLer-FM)*, 2025. URL https://openreview.net/forum?id=4vzhqq5hpX.

Franz Louis Cesista, Jiacheng You, and Keller Jordan. Squeezing 1-2% efficiency gains out of muon by optimizing the newton-schulz coefficients, 2025. URL http://leloykun.github.io/ponder/muon-opt-coeffs/.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, et al. Deepseek-v3 technical report, 2025. URL https://arxiv.org/abs/2412.19437.

Shizhe Diao, Yu Yang, Yonggan Fu, Xin Dong, Dan Su, Markus Kliegl, Zijia Chen, Peter Belcak, Yoshi Suhara, Hongxu Yin, Mostofa Patwary, Yingyan Celine Lin, Jan Kautz, and Pavlo Molchanov. Nemotron-CLIMB: Clustering-based iterative data mixture bootstrapping for language model pre-training. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2025. URL https://openreview.net/forum?id=aBlqKPkc4a.

Essential AI. Layer Sharding for Large-Scale Training with Muon, 2025. URL https://www.essential.ai/research/infra.

Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, et al. Gemma 3 technical report, 2025. URL https://arxiv.org/abs/2503.19786.

GLM-5 Team, Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, et al. Glm-5: from vibe coding to agentic engineering, 2026. URL https://arxiv.org/abs/2602.15763.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Ekaterina Grishina, Matvey Smirnov, and Maxim Rakhuba. Accelerating newton-schulz iteration for orthogonalization via chebyshev-type polynomials, 2026. URL https://arxiv.org/abs/2506.10935.

Wentao Guo, Mayank Mishra, Xinle Cheng, Ion Stoica, and Tri Dao. Sonicmoe: Accelerating moe with io and tile-aware optimizations, 2026. URL https://arxiv.org/abs/2512.14080.

Nicholas J. Higham. *Functions of Matrices: Theory and Computation*. Society for Industrial and Applied Mathematics, 2008. doi: 10.1137/1.9780898717778.

Taejong Joo, Wenhan Xia, Cheolmin Kim, Ming Zhang, and Eugene Ie. On surprising effectiveness of masking updates in adaptive optimizers, 2026. URL https://arxiv.org/abs/2602.15322.

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

Ahmed Khaled, Kaan Ozkara, Tao Yu, Mingyi Hong, and Youngsuk Park. MuonBP: Faster muon via block-periodic orthogonalization. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=mHouLSUQP5.

Kimi Team, Yifan Bai, Yiping Bao, Y. Charles, Cheng Chen, Guanduo Chen, Haiting Chen, Huarong Chen, Jiahao Chen, Ningxin Chen, et al. Kimi k2: Open agentic intelligence, 2025. URL https://arxiv.org/abs/2507.20534.

Slobodan Lakić. On the computation of the matrix k-th root. *Zeitschrift für Angewandte Mathematik und Mechanik*, 78(3):167–172, 1998. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291521-4001%28199803%2978%3A3%3C167%3A%3AAID-ZAMM167%3E3.0.CO%3B2-R.

Zichong Li, Liming Liu, Chen Liang, Weizhu Chen, and Tuo Zhao. Normuon: Making muon more efficient and scalable. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=m1IRWFAMsa.

Junghwan Lim, Sungmin Lee, Dongseok Kim, Taehyun Kim, Eunhwan Park, Jeesoo Lee, Jeongdoo Lee, Junhyeok Lee, Wai Ting Cheung, Dahye Choi, et al. Motif 2 12.7B technical report, 2025. URL https://arxiv.org/abs/2511.07464.

Tianyang Lin. Flash-muon: An efficient implementation of muon optimizer, 2025. URL https://github.com/nil0x9/flash-muon.

Jingyuan Liu. A proof of concept for Distributed Muon. Pull Request #1428, NVIDIA/Megatron-LM, 2025. URL https://github.com/NVIDIA/Megatron-LM/pull/1428. Accessed: 2026-07-10.

Jingyuan Liu, Jianlin Su, Xingcheng Yao, Zhejun Jiang, Guokun Lai, Yulun Du, Yidao Qin, Weixin Xu, Enzhe Lu, Junjie Yan, et al. Muon is scalable for llm training, 2025. URL https://arxiv.org/abs/2502.16982.

Anton Lozhkov, Loubna Ben Allal, Leandro von Werra, and Thomas Wolf. Fineweb-edu: the finest collection of educational content, 2024. URL https://huggingface.co/datasets/HuggingFaceFW/fineweb-edu.

Ionut-Vlad Modoranu, Mher Safaryan, Erik Schultheis, Max Ryabinin, Artem Chumachenko, and Dan Alistarh. Trion: FFT-based dynamic subspace selection for low-rank adaptive optimization of LLMs. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=TkHjRwbMNl.

Laker Newhouse, Dakota Goldberg, and Ricardo Ruiz. Faster symmetric matrix multiplication with ThunderKittens, 2024. URL https://www.lakernewhouse.com/assets/writing/faster-symmul-with-thunderkittens.pdf.

OpenAI, Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, et al. gpt-oss-120b & gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

Ziyuan Tang, Tianshi Xu, Yousef Saad, and Yuanzhe Xi. Hierarchical muon: Tiled newton-schulz updates for efficient muon optimization, 2026. URL https://arxiv.org/abs/2606.27216.

Byron Xu. Transpose one of the MLP matrices + add Triton kernel for symmetric matmul. Pull Request #109, KellerJordan/modded-nanogpt, 2025. URL https://github.com/KellerJordan/modded-nanogpt/pull/109. Accessed: 2026-07-10.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. PyTorch FSDP: experiences on scaling fully sharded data parallel. *Proceedings of the VLDB Endowment*, 16(12):3848–3860, 2023.

CONTENTS

1 Introduction **1**

2 The Challenge of Scaling Muon **2**
2.1 Muon and NorMuon Recap . . . 2
2.2 Standard Newton-Schulz . . . 2
2.3 Prior Work on Scaling Muon . . . 3

3 Gram Newton-Schulz **3**

4 Symmetric GEMM Kernels in CuteDSL **5**

5 The Dion3 Update Rule **6**

6 Megabatching and Communication **7**

7 Experiments **7**
7.1 Model Quality Is Preserved . . . 7
7.2 Dion3 Accelerates the Optimizer . . . 8

8 Conclusion **9**

Appendices **14**

A Comparison with Related Work **14**
A.1 Using Standard Newton-Schulz . . . 14
A.2 Improving Newton-Schulz . . . 14
A.3 Orthogonalizing a Smaller Matrix . . . 15

B Details and Stability Analysis of Gram Newton-Schulz **15**
B.1 Instability of Naive Gram Newton-Schulz . . . 16
B.2 Stabilizing Gram Newton-Schulz by Restarting . . . 20

C Kernel Implementation Details **23**
C.1 Symmetric GEMM Kernel Details . . . 23
C.2 Implementation Strategy in Code . . . 24
C.3 Kernel Optimizations for Standard Newton-Schulz . . . 24

D Implementation Details of Dion3 Update Rule **24**

E Motivating and Benchmarking Megabatching **25**

F Alternative Experiments on Gram Newton-Schulz **26**
F.1 Setup . . . 26
F.2 Kernelized Gram Newton-Schulz preserves quality . . . 27
F.3 Kernelized Gram Newton-Schulz speeds up the optimizer step . . . 27

G Additional Experiments **28**
G.1 Architecture and Optimization Setup . . . 28
G.2 Learning-Rate Transfer Rule for Dion3 . . . 30
G.3 Dion3 versus NorMuon Across Model Sizes . . . 31
G.4 Finer-Grained Timing Metrics . . . 31
G.5 Ablations . . . 33

H Case Studies of End-to-End Training Time **34**

Appendices

A Comparison with Related Work

A.1 Using Standard Newton-Schulz

As Muon has gained popularity, successive work has sought to improve it in several ways. Most of these proposals (e.g. NorMuon) modify Muon's update rule so as to reach a given loss in fewer training steps; however, they use the same Newton-Schulz routine described above and generally suffer from the same scaling challenges.

Despite these challenges, Muon has been adopted in real-world training pipelines. Early implementations handled distributed weights by assembling a full copy of the momentum matrix on each device and computing $\mathrm{polar}(\boldsymbol{M})$ locally. This approach is simple but wasteful; it duplicates identical Newton-Schulz steps across devices, increasing Muon's already considerable computational cost. Essential AI (2025) proposed using all-to-all communication along the sharding dimension, with each device processing a different weight in parallel. Ahn et al. (2025) implemented this strategy in PyTorch FSDP2, and Lim et al. (2025) examined its compute-communication overlap characteristics. While this approach makes the overhead of distributed Newton-Schulz manageable in some settings, it does not fully resolve the scalability limitations.

Muon has been scaled successfully by Moonshot AI (Kimi Team et al., 2025; Liu et al., 2025). Their success was enabled by an alignment of several factors:

1. Earlier releases of PyTorch and Megatron-LM used DP-sharding strategies for optimizer states that were, by chance, favorable for Muon (Liu, 2025). Model and optimizer states were stored in large contiguous flat buffers, and data-parallel shards were produced by splitting this buffer. As a result, only tensors crossing a DP boundary required an additional gather. These advantageous strategies, however, have been deprecated in more recent releases.
2. They adopt a fine-grained MoE architecture with only a single dense layer (even fewer than in DeepSeek-V3 (DeepSeek-AI et al., 2025)), so most matrices remain small even at the one-trillion-parameter scale, keeping the Newton-Schulz overhead manageable.
3. Their main distributed training strategy combines pipeline parallelism with expert parallelism, which naturally distributes Muon's computation across devices with minimal communication.

In sum, Moonshot AI's success relied on particular choices of architecture, parallelism strategy, and framework. For Muon to serve as a general-purpose replacement for Adam, it would benefit from cheaper, more flexible scaling properties that relax these constraints. This is the goal of the present work.

A.2 Improving Newton-Schulz

A few papers *have* attempted to improve the Newton-Schulz step itself—e.g., by optimizing the sequence of polynomials (a_t, b_t, c_t) or the normalization step (Amsel et al., 2026; Boissin et al., 2025; Grishina et al., 2026)—but they retain the form of Algorithm 1. In contrast, our Gram Newton-Schulz algorithm departs from this form. Since its output is still mathematically identical to the standard version, it remains compatible with nearly all varieties of Muon, including prior improvements to Newton-Schulz.

Gram Newton-Schulz is closely akin to a method proposed in Appendix J of Amsel et al. (2026). Both aim to reduce the FLOP cost of Newton-Schulz, both form the Gram matrix to reduce the number of $m \times n$ matrix multiplications, and both are mathematically identical to standard Newton-Schulz. However, our work supersedes Amsel et al. (2026) in several ways. First, the precise formulas of Gram Newton-Schulz are different and, we believe, more stable. Second, we use symmetric matrix multiplication kernels; the opportunity to use these kernels more is an essential advantage of Gram Newton-Schulz not studied previously. Third, we undertake a thorough stability analysis and provide practical recommendations that allow Gram Newton-Schulz to be used in practice with minimal ad-hoc hyperparameter tuning. The iterative part of the method, which approximates the inverse square root of the Gram matrix $\boldsymbol{Q}_T \approx \boldsymbol{R}_0^{-1/2} := (\boldsymbol{X}\boldsymbol{X}^\top)^{-1/2}$, generalizes and speeds up

the method of Lakić (1998) (see also Higham (2008, Eq. 7.18)), but Lakić does not consider the symmetric kernels, the stability issues, or the application to $\mathrm{polar}(\boldsymbol{X})$.

The idea of exploiting symmetry to reduce the arithmetic cost of computing $\boldsymbol{A}^\top \boldsymbol{A}$ has appeared before, both in standard linear algebra packages like BLAS and in the context of Muon's Newton-Schulz routine (Lin, 2025; Newhouse et al., 2024; Xu, 2025). However, our results show that the true potential of this trick is only realized when combined with Gram Newton-Schulz, which requires more general symmetric operations like $\boldsymbol{AB} + \beta \boldsymbol{C}$. Furthermore, our symmetric matrix multiplication kernels are written in CuteDSL, exploiting the advanced features of NVIDIA's Hopper and Blackwell GPUs.

A.3 Orthogonalizing a Smaller Matrix

Our update rule is part of a second line of work that reduces the cost of the Newton-Schulz step by shrinking the size of its input, an approach pioneered by Dion (Ahn et al., 2025). Dion constructs a low-rank approximation of the momentum matrix and orthogonalizes only this approximation. If $\boldsymbol{M} \approx \boldsymbol{M}\widehat{\boldsymbol{V}} \cdot \widehat{\boldsymbol{V}}^\top$ is a rank-k approximation with $\widehat{\boldsymbol{V}}^\top \widehat{\boldsymbol{V}} = \boldsymbol{I} \in \mathbb{R}^{k \times k}$, then

$$\mathrm{polar}(\boldsymbol{M}) \approx \mathrm{polar}(\boldsymbol{M}\widehat{\boldsymbol{V}}\widehat{\boldsymbol{V}}^\top) = \mathrm{polar}(\boldsymbol{M}\widehat{\boldsymbol{V}})\widehat{\boldsymbol{V}}^\top. \tag{3}$$

Because the dimensions of $\boldsymbol{M}\widehat{\boldsymbol{V}}$ are much smaller than those of $\boldsymbol{M}$, this approximation greatly reduces the runtime of Newton-Schulz. In Dion, $\widehat{\boldsymbol{V}}$ is found by a warm-started power-iteration procedure. If $\widehat{\boldsymbol{V}}$ spans the top-k right singular vectors of $\boldsymbol{M}$, then this approximation gives the optimal rank-k approximation of $\mathrm{polar}(\boldsymbol{M})$. Ahn et al. (2025) found that approximating $\widehat{\boldsymbol{V}}$ from only a single warm-started power-iteration suffices for good downstream performance. A key element of Dion's success is *error feedback*, a technique that helps offset the error introduced by low-rank approximation in future iterations. We adopt error feedback and describe it fully in Section 5.

Trion (Modoranu et al., 2026) adopts the main ideas of Dion, but changes how the low-rank approximation is computed. Instead of constructing $\widehat{\boldsymbol{V}}$ via power iteration, it selects k columns from the discrete cosine transform matrix. Surprisingly, this simpler approximation performs even better than Dion, suggesting that finding a *good* low-rank approximation is not essential; the error-feedback mechanism compensates even for large differences between $\boldsymbol{M}$ and $\boldsymbol{M}\widehat{\boldsymbol{V}}\widehat{\boldsymbol{V}}^\top$.

We push this simplification further by using the simplest possible "low-rank approximation" of $\boldsymbol{M}$: we select k of its rows or columns and do not operate on the others. This achieves the same savings in the runtime of Newton-Schulz as Dion and Trion, but makes all the other steps of the algorithm (construction of $\boldsymbol{V}$, error feedback, etc.) much easier to implement, especially in the distributed setting.

Besides low-rank approximation, block orthogonalization is an alternative way to shrink the size of the input to Newton-Schulz. This approach partitions the momentum matrix into blocks—each typically corresponding to a shard in weight-sharded settings—and orthogonalizes each block separately. Several papers have proposed versions of this approach (Boreiko et al., 2025; Khaled et al., 2026; Tang et al., 2026). In particular, Khaled et al. (2026) found success by periodically alternating block-wise steps with full steps, a method they call MuonBP.

Both low-rank and block orthogonalization methods use Newton-Schulz as a black box. Therefore, they are trivial to combine with improvements to Newton-Schulz.

B Details and Stability Analysis of Gram Newton-Schulz

We now provide a proof of Theorem 2 and a full description of stabilized Gram Newton-Schulz, which were omitted from Section 3. Then, in Appendices B.1 and B.2, we analyze the instability of naive Gram Newton-Schulz (Algorithm 2) in depth, demonstrate how restarting helps, and address several practical considerations in the implementation of the stable version.

Theorem 2. *Let $p_1, \ldots, p_T$ be a sequence of odd polynomials with $p_t(x) = x h_t(x^2)$. Define $\begin{pmatrix} r_0 \\ q_0 \end{pmatrix} = \begin{pmatrix} x^2 \\ 1 \end{pmatrix}$ and $\begin{pmatrix} r_t \\ q_t \end{pmatrix} = \begin{pmatrix} r_{t-1} h_t(r_{t-1})^2 \\ q_{t-1} h_t(r_{t-1}) \end{pmatrix}$ for $t \in \{1, \ldots, T\}$. Then $q_T x = (p_T \circ \cdots \circ p_1)(x)$.*

Proof. Define $x_0 := x$ and $x_t := p_t(x_{t-1})$ for $t \in \{1, \dots, T\}$. We will show by induction that $r_t = x_t^2$ and $q_t = x_t/x_0$ for all t. The base case $t = 0$ holds by the definition $r_0 = x^2, q_0 = 1$. Now assume the hypothesis holds for $t-1$. By assumption,

$$x_t := p_t(x_{t-1}) = x_{t-1} h_t(x_{t-1}^2) \tag{4}$$

Squaring both sides of (4), applying the inductive hypothesis $r_{t-1} = x_{t-1}^2$, and recalling the definition of r_t, we have

$$x_t^2 = x_{t-1}^2 h_t(x_{t-1}^2)^2 = r_{t-1} h_t(r_{t-1})^2 =: r_t.$$

If we instead divide both sides of (4) by x_0, apply the inductive hypothesis, and recall the definition of q_t, we have

$$\frac{x_t}{x_0} = \frac{x_{t-1}}{x_0} h_t(x_{t-1}^2) = q_{t-1} h_t(r_{t-1}) = q_t.$$

Thus, both parts of the hypothesis hold for t. Finally, conclude $(p_T \circ \cdots \circ p_1)(x) =: x_T = q_T x_0$. □

Algorithm 4 gives the final, stabilized version of Gram Newton-Schulz that we implement in our `gram-newton-schulz` package. It uses the restart strategy described in the main text and developed in Appendix B.2. To choose when to restart, we sweep across iterations for the restart location that provides the best bound on the condition number of $\boldsymbol{Q}_t$, assuming that forming the Gram matrix introduces spurious eigenvalues as negative as $-4 \cdot 10^{-4}$. We walk through an example of how to automate this sweep in Appendix B.2.1. Algorithm 4 also slightly reformulates the arithmetic of the intermediate polynomials and switches to `float16` instead of `bfloat16`. These two changes are motivated in Appendices B.2.2 and B.2.3.

Algorithm 4 Stabilized Gram Newton-Schulz

Input: $\boldsymbol{X} \in \mathbb{R}^{n \times m}$ with $n \le m$, coefficients $\{(a_t, b_t, c_t)\}_{t=1}^5$

$\boldsymbol{X} \leftarrow \boldsymbol{X} \,/\, (\|\boldsymbol{X}\|_{\mathsf{F}} + 10^{-7})$ *// Normalize sing. vals. to* $[0, 1]$
$\boldsymbol{X} \leftarrow \texttt{float16}(\boldsymbol{X})$ *// Use half precision for speed*
$\boldsymbol{R}_0 \leftarrow \boldsymbol{X}\boldsymbol{X}^\top$
$\boldsymbol{Q}_0 \leftarrow \boldsymbol{I}$
for $t = 1, \dots, 5$ **do**
  if $t = 3$ **then** *// Restart to stabilize*
    $\boldsymbol{X} \leftarrow \boldsymbol{Q}_2 \boldsymbol{X}$
    $\boldsymbol{R}_2 \leftarrow \boldsymbol{X}\boldsymbol{X}^\top$
    $\boldsymbol{Q}_2 \leftarrow \boldsymbol{I}$
  end if
  $\boldsymbol{Z}_t \leftarrow b_t \boldsymbol{R}_{t-1} + c_t \boldsymbol{R}_{t-1}^2$
  $\boldsymbol{Q}_t \leftarrow \boldsymbol{Q}_{t-1} \boldsymbol{Z}_t + a_t \boldsymbol{Q}_{t-1}$ *// $\boldsymbol{Q}_t = \boldsymbol{Q}_{t-1} h_t(\boldsymbol{R}_{t-1})$*
  $(\mathbf{RZ})_t \leftarrow \boldsymbol{R}_{t-1} \boldsymbol{Z}_t + a_t \boldsymbol{R}_{t-1}$
  $\boldsymbol{R}_t \leftarrow \boldsymbol{Z}_t (\mathbf{RZ})_t + a_t (\mathbf{RZ})_t$ *// $\boldsymbol{R}_t = \boldsymbol{R}_{t-1} h_t(\boldsymbol{R}_{t-1})^2$*
end for
$\boldsymbol{X} \leftarrow \boldsymbol{Q}_5 \boldsymbol{X}$
return $\boldsymbol{X}$

B.1 Instability of Naive Gram Newton-Schulz

We trained Llama-430M with Muon using Naive Gram Newton-Schulz (Algorithm 2). The loss curve is shown in Figure 6. Clearly, training is very unstable. Not only do we observe loss spikes, but eventually, the outputs of Gram Newton-Schulz become full of Infs. The problem is due to floating point arithmetic. While Gram Newton-Schulz is mathematically equivalent to standard Newton-Schulz in exact arithmetic, it behaves differently in finite precision, especially in half precision. As half precision is essential for good performance, this makes Naive Gram Newton-Schulz unworkable in practice.

We now analyze the source of the instability in order to motivate our solution. A Jupyter notebook reproducing the experiments in this section is available in the supplementary material.

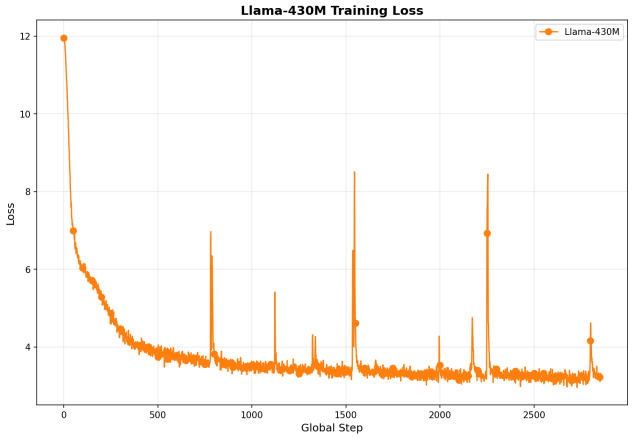


Figure 6: Naive Gram Newton-Schulz used to train Llama-430M in half precision. Numerical instability wrecks training.

B.1.1 Tracking Eigenvalues of Intermediate Matrices

To understand how the matrices in Naive Gram Newton-Schulz evolve and why they diverge, we track their eigenvalues and singular values. Recall that the entries of any matrix are upper bounded by its largest singular value, so if we can bound the singular values, we can prevent blowups.

As a baseline, we start by running Naive Gram Newton-Schulz in full `float64` precision for 8 steps to simulate its behavior in exact arithmetic. We use a synthetic 128×512 matrix with an exponentially decaying spectrum. To make our plots more readable, the experiments in this section use the coefficients $(a_t, b_t, c_t) = (\frac{15}{8}, -\frac{10}{8}, \frac{3}{8})$ at every iteration, but our conclusions will generalize to the coefficients used in practice. If $\boldsymbol{X}_0 = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$ is the SVD of the input matrix, then the intermediate matrices of Algorithm 2 ($\boldsymbol{R}_t$, $\boldsymbol{Q}_t$, $\boldsymbol{Z}_t$) are square symmetric with eigenvectors $\boldsymbol{U}$. We therefore plot the diagonal entries of $\boldsymbol{U}^\top \boldsymbol{R}_t \boldsymbol{U}$ and $\boldsymbol{U}^\top \boldsymbol{Q}_t \boldsymbol{U}$ against the corresponding singular values in $\boldsymbol{\Sigma}$ to track how each evolves according to the polynomial update rules—or diverges from them. Even though Gram Newton-Schulz does not need to compute $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_{T-1}$, we do so here for demonstration using the formula $\boldsymbol{X}_t = \boldsymbol{Q}_t \boldsymbol{X}_0$ and plot $\boldsymbol{U}^\top \boldsymbol{X}_t \boldsymbol{V}$ against $\boldsymbol{\Sigma}$. The top panel of Figure 7 shows that the eigenvalues evolve as predicted by Theorem 2. Initially, $x_0 \in [0, 1]$, $r_0 = x_0^2 \in [0, 1]$ and $q_0 = 1$. As the algorithm progresses, $r_t \to 1$, $q_t = x_t / x_0 \to 1/x_0$, and $x_t \to 1$. For $x_0 \approx 1$, convergence is faster; for $x_0 \ll 1$, it is slower.

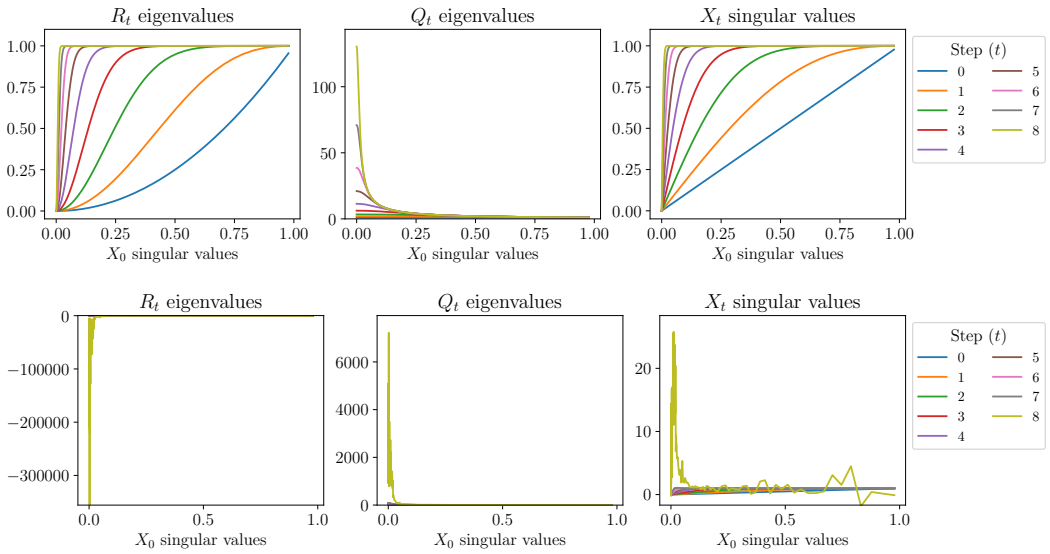


Figure 7: Evolution of eigenvalues of $\boldsymbol{R}_t$, $\boldsymbol{Q}_t$, and $\boldsymbol{X}_t$ in Naive Gram Newton-Schulz with $(a_t, b_t, c_t) = (15/8, 10/8, 3/8)$ as a function of the corresponding singular value of $\boldsymbol{X}_0$. **Top:** `float64`. **Bottom:** `bfloat16`.

Now we rerun the experiment in `bfloat16` arithmetic (Figure 7, bottom). The first few iterations look correct, but by step 7, we see noisy and unexpected behavior ($r_t < 0$, $x_t > 1$). By step 8, the spectrum diverges completely, and by step 10 the algorithm returns Infs. We now describe two key causes of divergence: spurious negative eigenvalues in the Gram matrix $\boldsymbol{X}\boldsymbol{X}^\top$, and eigenvector drift.

B.1.2 Spurious Negative Eigenvalues

The leading cause of divergence is the presence of negative eigenvalues in the Gram matrix due to half-precision arithmetic. These negative eigenvalues blow up after too many iterations of Gram Newton-Schulz.

By construction, $r_t = x_t^2 \geq 0$, so $\boldsymbol{R}_t$ should be positive semidefinite (Theorem 2), but Figure 7 shows that is has negative eigenvalues. In fact, even $\boldsymbol{R}_0$ has tiny negative eigenvalues introduced in the first matrix multiplication $\boldsymbol{X}_0\boldsymbol{X}_0^\top$. Because $\boldsymbol{X}_0$ has many singular values that are nearly zero (as is the case in Muon), $\boldsymbol{R}_0 = \boldsymbol{X}_0\boldsymbol{X}_0^\top$ has many eigenvalues that are *numerically equal* to zero. In `bfloat16`, even a slightly negative number can be numerically equal to zero. Later computations can introduce additional negative eigenvalues into $\boldsymbol{R}_t$ as well. These eigenvalues represent nothing about the original problem; they are just an artifact of floating point arithmetic. Therefore, we call them "spurious eigenvalues".

These spurious negative eigenvalues start small, but Figure 7 shows that their magnitude grows quickly. Recall the update rule:

$$r_t = r_{t-1} z_t^2 = r_{t-1} h_t(r_{t-1})^2 \tag{5}$$

For our choice of coefficients, $h_t(x) = \frac{15}{8} - \frac{10}{8}x + \frac{3}{8}x^2$. Figure 8 plots this update rule. As it shows, $r_t < \left(\frac{15}{8}\right)^2 r_{t-1}$. Thus, if any $r_t < 0$, the spurious negative eigenvalues grow exponentially, diverging to $-\infty$! This sets off a chain reaction that causes $\boldsymbol{Q}_t$ and $\boldsymbol{X}_t$ to diverge as well. This problem cannot be fixed by choosing different polynomials; while the main loop of Gram Newton-Schulz approximates the inverse square root of $r_0 > 0$, it diverges for negative inputs.

To prove that the tiny spurious negative eigenvalues of $\boldsymbol{R}_0$ suffice to cause divergence, we rerun the experiment in `float64` precision, but cast $\boldsymbol{R}_0$ from `float64` to `bfloat16` and then back to `float64` to induce a small floating point error. As Figure 9 shows, this is enough to cause a blowup.

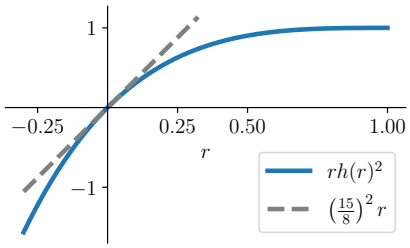


Figure 8: When $\boldsymbol{R}_t$ evolves according to (5), negative eigenvalues diverge to $-\infty$.

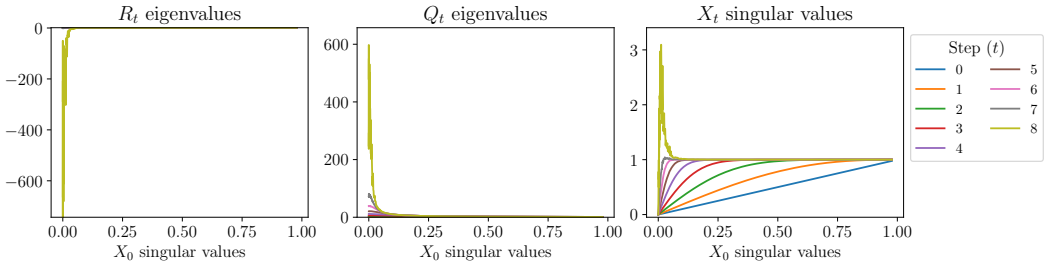


Figure 9: Evolution of eigenvalues of $\boldsymbol{R}_t$, $\boldsymbol{Q}_t$, and $\boldsymbol{X}_t$ when all operations use `float64` except $\boldsymbol{R}_0 = \boldsymbol{X}_0\boldsymbol{X}_0^\top$, which uses `bfloat16`.

B.1.3 Eigenvector Drift

Spurious negative eigenvalues are not the only source of numerical instability. If we take the input matrix $\boldsymbol{X}_0$ to have no small singular values (i.e., all ≥ 0.017), then we do not observe any negative eigenvalues in $\boldsymbol{R}_t$, but $\boldsymbol{X}_t$ still fails to converge. The culprit seems to be "eigenvector drift".

In exact arithmetic, the eigenvectors of all intermediate matrices match $\boldsymbol{U}$, the left singular vectors of $\boldsymbol{X}_0$, but in finite precision they do not. We demonstrate this effect by measuring how far $\boldsymbol{U}^\top\boldsymbol{R}_t\boldsymbol{U}$, $\boldsymbol{U}^\top\boldsymbol{Q}_t\boldsymbol{U}$, and $\boldsymbol{U}^\top\boldsymbol{X}_t\boldsymbol{V}$ are from being diagonal matrices. That is, we take the Frobenius norm of the off-diagonal entries as a fraction of the total Frobenius norm. Figure 10 shows that after several iterations, the eigenvectors / singular vectors of $\boldsymbol{Q}_t$ and $\boldsymbol{X}_t$ have drifted significantly from those of $\boldsymbol{X}_0$. At the same time, the eigen*values* of $\boldsymbol{Q}_t$ (and by extension, those of $\boldsymbol{X}_t$) diverge from where they would be in exact arithmetic. The growing eigenvalues of $\boldsymbol{Q}_t$ seem to spill into one another. The strength of this effect is less consistent than that of negative eigenvalues, but it is still harmful.

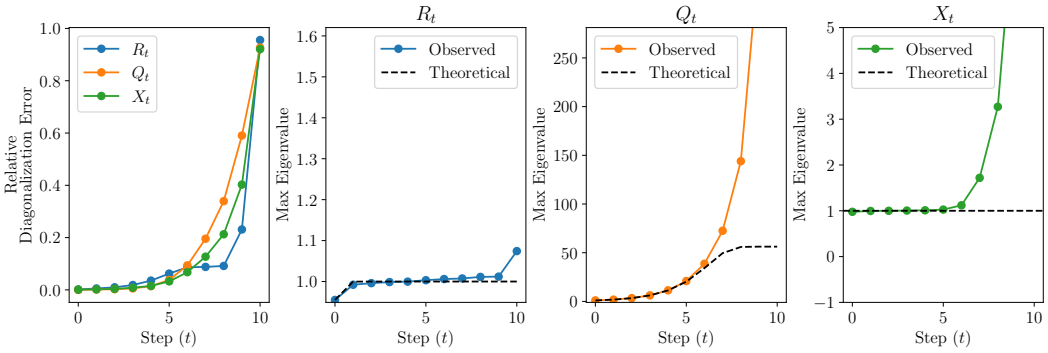


Figure 10: As the eigenvectors drift, the spectral norms of $\boldsymbol{R}_t$, $\boldsymbol{Q}_t$, and $\boldsymbol{X}_t$ diverge.

B.2 Stabilizing Gram Newton-Schulz by Restarting

As we have seen, if we run Gram Newton-Schulz for more than a few iterations, the spurious negative eigenvalues of $\boldsymbol{R}_t$ diverge to negative infinity and $\boldsymbol{Q}_t$ blows up. Our solution is simple: run Gram Newton-Schulz for only a few iterations. For instance, rather than using Gram Newton-Schulz to compute $\boldsymbol{X}_T$ directly, we can use it to compute $\boldsymbol{X}_5$ in a stable manner. While $\boldsymbol{X}_5$ is not a good approximation to $\mathrm{polar}(\boldsymbol{X}_0)$, it is closer than where we started. Now we apply Gram Newton-Schulz a second time on the input $\boldsymbol{X}_5$ to compute $\boldsymbol{X}_{10}$ stably. This process can be repeated to reach any desired $\boldsymbol{X}_T$. This restarting technique sacrifices some of the performance gains of Gram Newton-Schulz, but it still offers a significant speedup over standard Newton-Schulz.

Figure 11 repeats the experiment from above with a restart every five iterations. While $\boldsymbol{R}_t$ develops some negative eigenvalues, unlike before, the growth of these eigenvalues is controlled. Each time we restart, we re-initialize $\boldsymbol{R}_t = \boldsymbol{X}_t\boldsymbol{X}_t^\top$, eliminating any large negative eigenvalues. By restarting regularly, we prevent any from growing too large. As expected, $\boldsymbol{Q}_t$ resets to the identity at iterations $5, 10, 15, 20$, and 25. Therefore, the eigenvalues of $\boldsymbol{Q}_t$ never grow beyond ≈ 12, despite the negative eigenvalues in $\boldsymbol{R}_t$. Since the eigenvalues of $\boldsymbol{Q}_t$ remain controlled, those of $\boldsymbol{X}_t = \boldsymbol{Q}_t\boldsymbol{X}_{t-5}$ stay at or below 1.

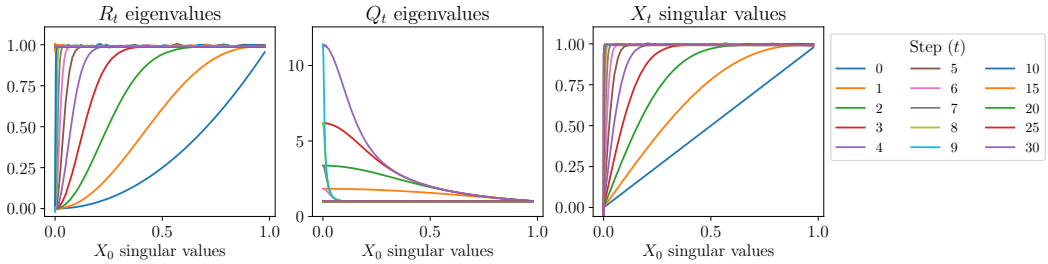


Figure 11: Restarting prevents the divergence of $\boldsymbol{R}_t$ in half-precision.

Restarting also helps control eigenvector drift. We repeat the experiment from Figure 10 on the same matrix (with all singular values > 0.017) with a restart after step 5. Diagonalization error always remains ≤ 0.05, and the maximum eigenvalues now closely track their predicted values. Note that we always measure eigenvector drift relative to the original input $\boldsymbol{X}_0$, not the restarted $\boldsymbol{X}_5$.

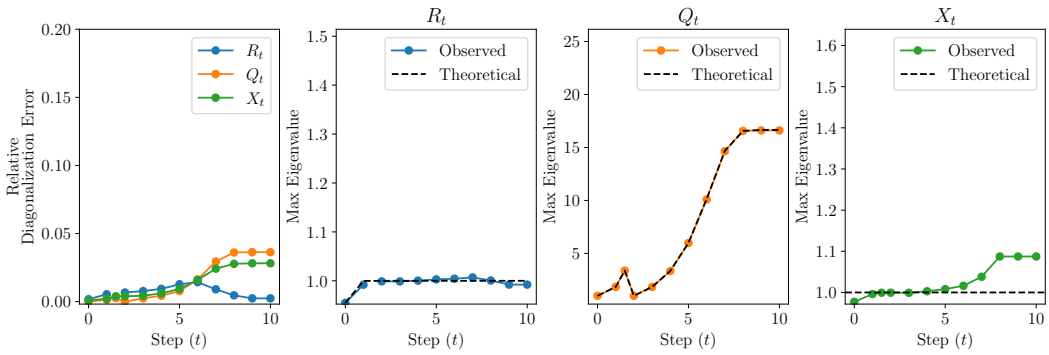


Figure 12: Restarting curbs eigenvector drift.

B.2.1 When to Restart: Polar Express Coefficients for Muon

We now derive the optimal restarting schedule for a given total number of iterations T. To prevent $\boldsymbol{X}_t = \boldsymbol{Q}_t\boldsymbol{X}_0$ from blowing up, we must control the condition number of $\boldsymbol{Q}_t$, even when $\boldsymbol{R}_0$ has spurious negative eigenvalues. (Because $q_t \to 1/x_0 \geq 1$, this also controls the maximum eigenvalue

of $\boldsymbol{Q}_t$.) The growth of $\boldsymbol{Q}_t$ in turn depends on the size of the spurious negative eigenvalues and the specific sequence of polynomials defined by $\{(a_t, b_t, c_t)\}_{t=1}^{T}$. Fixing a desired number of restarts, we sweep over all possible restart schedules and pick the one that minimizes the maximum condition number of $\boldsymbol{Q}_t$ across ts.

For the application to Muon, we analyze five iterations of the Polar Express coefficients (Amsel et al., 2026). In the experiments of the previous section, we observe that the most negative spurious eigenvalue of $\boldsymbol{R}_0$ is about $-4 \cdot 10^{-4}$. Therefore, we simulate Gram Newton-Schulz in full precision with Polar Express coefficients and track how the eigenvalues of $\boldsymbol{R}_t$ and $\boldsymbol{Q}_t$ evolve when $\boldsymbol{R}_0$ has eigenvalues in the range $[-4 \cdot 10^{-4}, 1]$. The left panels of Figure 13 show that without restarting, they blow up. We then repeat the simulation with one restart, sweeping all possible choices of when to restart. Every time we restart and form $\boldsymbol{R} = \boldsymbol{X}\boldsymbol{X}^\top$, we subtract $4 \cdot 10^{-4}\boldsymbol{I}$ to simulate potentially dangerous corruption of the eigenvalues due to floating point error. As the right panel shows, restarting after the *second* iteration provides the best bounds, ensuring that the eigenvalues of $\boldsymbol{R}_t$ stay well above -0.4 and those of $\boldsymbol{Q}_t$ stay below ≈ 100 for all iterations. Figure 14 shows that for Gram Newton-Schulz with Polar Express coefficients and one restart after the second iteration, all eigenvalues of $\boldsymbol{X}_t$ converge stably to 1 .

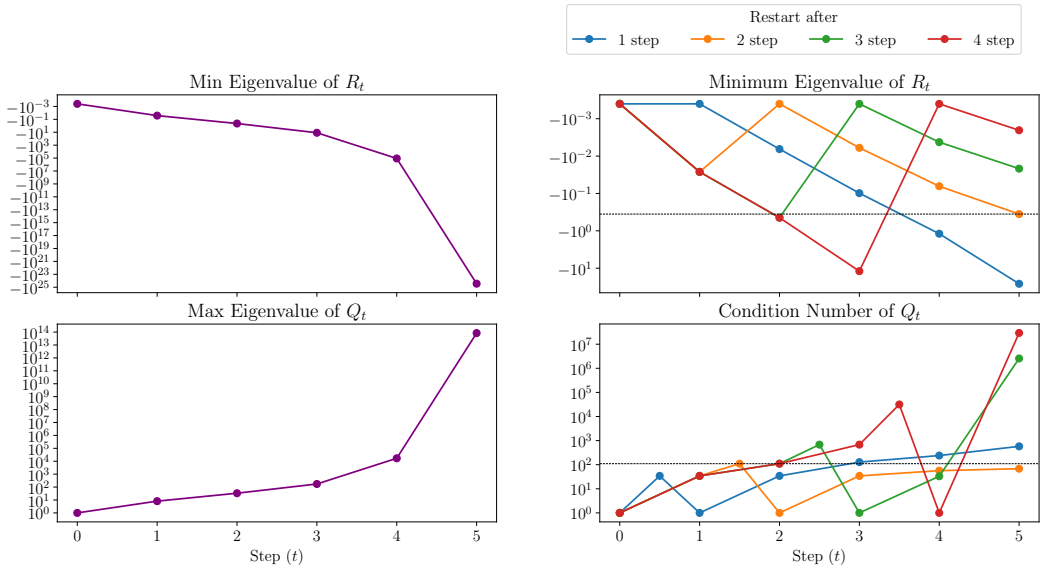


Figure 13: **Left:** Min/max eigenvalue of $\boldsymbol{R}_t$ and $\boldsymbol{Q}_t$ without restarts. $\boldsymbol{R}_0$ starts with a negative eigenvalue at $-4 \cdot 10^{-4}$. **Right:** Minimum eigenvalue of $\boldsymbol{R}_t$ and condition number of $\boldsymbol{Q}_t$ if restart is placed after iteration 1, 2, 3, or 4. Restarting after iteration 2 best controls the size of $\boldsymbol{Q}_t$.

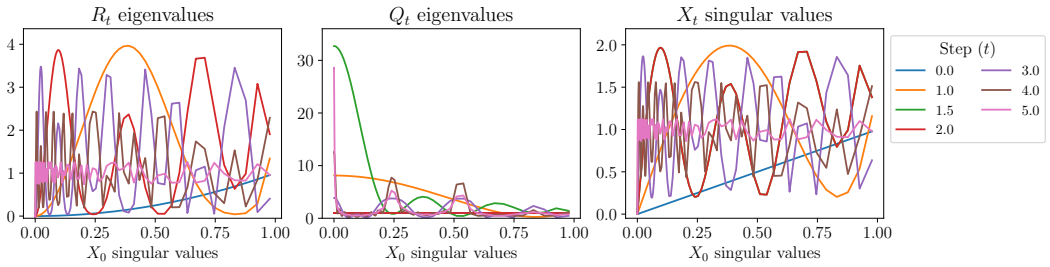


Figure 14: Gram Newton-Schulz with Polar Express coefficients and a restart after 2 iterations converges stably.

B.2.2 Further Precautions

While restarting greatly improves stability, it is not absolutely foolproof. A second or third restart may be required if running for more than five iterations or using a numerically sensitive set of coefficients. Moreover, the usual numerical precautions for standard Newton-Schulz still apply.

Safety Factors Most choices of Newton-Schulz polynomials are designed to converge only when the input lies in $[0, 1]$; any singular values larger than 1 may diverge rapidly, as Figure 15 shows. Even when $\boldsymbol{X}_0$ is properly normalized, singular values greater than 1 can arise due to numerical error. This problem affects standard Newton-Schulz too, so the Polar Express polynomials are typically adjusted according to the formula $\tilde{p}_t(x) = p_t(x/1.02)$. This ensures convergence even for singular values as large as 1.02. When using Gram Newton-Schulz, roundoff errors like this can worsen due to computations like $\boldsymbol{X}\boldsymbol{X}^\top$, which do not have such numerical buffers; however, we have never seen this happen when using our recommended setup (`float16` arithmetic with restarting after 2 iterations). It is wise to be conservative in the choice of safety factor, for instance, by replacing 1.02 with 1.05.

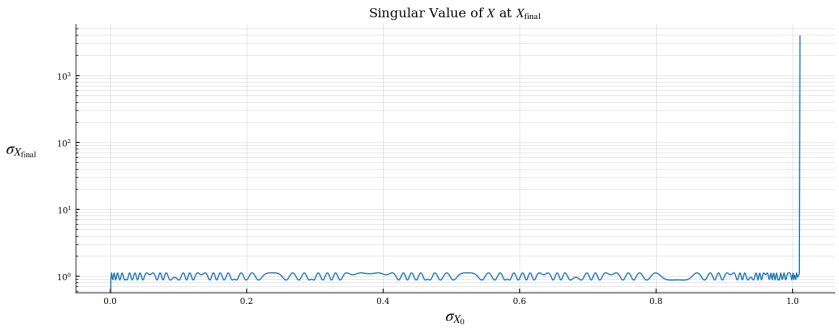


Figure 15: Theoretical behavior of both standard and Gram Newton-Schulz (Polar Express coefficients) when $\boldsymbol{X}_0$ has singular values slightly above one.

Float16 vs BFloat16 in Newton-Schulz In addition, we argue for using `float16` instead of `bfloat16` to implement Newton-Schulz. Both use 16 bits and thus run with the same performance; however, the distribution of the 16 bits across the mantissa and exponent bits differs. Compared to `bfloat16`, `float16` represents values from a narrower range, but it has greater precision within that range. For Newton-Schulz, however, the range of `float16` (roughly $6.1 \cdot 10^{-5}$ to $6.5 \cdot 10^4$) suffices because the magnitudes of its intermediate matrices are not too large. On certain test matrices, we see more accurate $\mathrm{polar}(\boldsymbol{X})$ approximations with `float16`, but in practice, we have not found a case where the training quality is meaningfully different between `float16` and `bfloat16`. Still, we default to `float16`.

High Accuracy Setting In the application to Muon, we do not need to compute $\mathrm{polar}(\boldsymbol{X})$ to high accuracy, and Appendix F.2 shows that Muon with Gram Newton-Schulz yields effectively identical results to Muon with standard Newton-Schulz in terms of training quality. However, when high accuracy is desired, the usual warnings about forming the Gram matrix apply. Since forming $\boldsymbol{X}\boldsymbol{X}^\top$ immediately squares the condition number, Gram Newton-Schulz may not be appropriate in these cases.

B.2.3 Computing Matrix Quadratics

A key step in Gram Newton-Schulz is computing the matrix quadratic $\boldsymbol{Z}_t \leftarrow a_t\boldsymbol{I} + b_t\boldsymbol{R}_{t-1} + c_t\boldsymbol{R}_{t-1}^2$. Standard Newton-Schulz implicitly computes matrix quadratics too, but PyTorch implementations typically avoid adding the identity explicitly. Rather, they compute $\boldsymbol{X}(a_t\boldsymbol{I} + b_t\boldsymbol{A} + c_t\boldsymbol{A}^2)$ in two steps, each with a single GEMM:

1. $\boldsymbol{B} \leftarrow b_t\boldsymbol{A} + c_t\boldsymbol{A}^2$
2. $\boldsymbol{X} \leftarrow a_t\boldsymbol{X} + \boldsymbol{B}\boldsymbol{X}$

Our symmetric GEMM kernel can compute matrix quadratics in one launch, fusing the addition of $a_t\boldsymbol{I}$ by adding a_t to all diagonal entries of the output when they are at the register level. This optimization completely obviates any I/O operations needed for the $a_t\boldsymbol{I}$ addition, typically outspeeding `gemm_symmetric(A, B, C, c_t, b_t) + a_t * I`, which would require loading $\boldsymbol{I}$ from general memory to shared memory to registers. Once $\boldsymbol{Z}_t$ is assembled, Gram Newton-Schulz can perform its three subsequent multiplications without the need for further fused additions.

However, our tests show that adding $a_t\boldsymbol{I}$ explicitly can be less stable than distributing it into those later three multiplications. If we stress-test our method by ignoring some of our own advice—restarting after three iterations instead of two, using a Polar Express safety factor of 1.02 instead of 1.05, and computing the quadratic with $a_t\boldsymbol{I}$ explicitly—we observe instability. Interestingly, this instability disappears if we use non-symmetric GEMMs (either from PyTorch or Quack) instead of our symmetric kernels; however, if we force symmetry after calling standard PyTorch GEMMs, we see instability again. We conclude that fusing $+a_t\boldsymbol{I}$ into a symmetric GEMM is numerically unfavorable; this is not just a kernel bug.

We believe this effect can be explained as follows. While the fused kernel computes $a_t\boldsymbol{I} + b_t\boldsymbol{R}_{t-1} + c_t\boldsymbol{R}_{t-1}^2$ in `float32` arithmetic under the hood, the result $\boldsymbol{Z}_t$ is rounded back down to `float16` at the end of the GEMM. Future computations like $\boldsymbol{Q}_t\boldsymbol{Z}_t$ suffer from this loss of precision in a_t. In contrast, if the $a_t\boldsymbol{I}$ term is handled implicitly, all arithmetic involving a_t takes place in `float32`. Therefore, it is more stable to compute $a_t\boldsymbol{Q}_t + \boldsymbol{Q}_t\left(b_t\boldsymbol{R}_{t-1} + c_t\boldsymbol{R}_{t-1}^2\right)$ than $\boldsymbol{Q}_t\left(a_t\boldsymbol{I} + b_t\boldsymbol{R}_{t-1} + c_t\boldsymbol{R}_{t-1}^2\right)$.

We reiterate that in all our experiments, this instability can be avoided entirely by restarting correctly or using a safety factor of 1.05. Out of an abundance of caution, we rearrange Naive Gram Newton-Schulz (Algorithm 2) to avoid adding $a_t\boldsymbol{I}$ explicitly. We change

1. $\boldsymbol{Z}_t \leftarrow a_t\boldsymbol{I} + b_t\boldsymbol{R}_{t-1} + c_t\boldsymbol{R}_{t-1}^2$ // Apply $h_t(\boldsymbol{R}_{t-1})$
2. $\boldsymbol{Q}_t \leftarrow \boldsymbol{Q}_{t-1}\boldsymbol{Z}_t$
3. $\boldsymbol{R}_t \leftarrow \boldsymbol{Z}_t\boldsymbol{R}_{t-1}\boldsymbol{Z}_t$

to

1. $\boldsymbol{Z}_t \leftarrow b_t\boldsymbol{R}_{t-1} + c_t\boldsymbol{R}_{t-1}^2$
2. $\boldsymbol{Q}_t \leftarrow \boldsymbol{Q}_{t-1}\boldsymbol{Z}_t + a_t\boldsymbol{Q}_{t-1}$
3. $(\mathbf{RZ})_t \leftarrow \boldsymbol{R}_{t-1}\boldsymbol{Z}_t + a_t\boldsymbol{R}_{t-1}$
4. $\boldsymbol{R}_t \leftarrow \boldsymbol{Z}_t(\mathbf{RZ})_t + a_t(\mathbf{RZ})_t$

This change fixes all collected training examples in which symmetric GEMMs were less stable than non-symmetric GEMMs.

C Kernel Implementation Details

C.1 Symmetric GEMM Kernel Details

We implement all of our symmetric GEMM kernels with square cluster work tiles. Hopper uses cluster size $(2, 1)$ and thread block tile size $(128, 256)$, and Blackwell uses cluster size $(2, 1)$ and 2-CTA collaboration, in which the 2 thread blocks in the cluster collaborate on the same big $(256, 256)$ tile. Notably, highly optimized custom GEMM kernels on Hopper typically use Ping Pong Scheduling, in which the matrix-multiply accumulate (MMA) of tile i and the epilogue of tile $i - 1$ are overlapped in two consumer warp groups.[2] However, Ping Pong Scheduling uses more registers at once, and $(128, 256)$ is too large of a tile size for Ping Pong Scheduling, leading to register spillage. This is much slower than standard single producer warp, single consumer warp scheduling. Thus, our Hopper symmetric kernels do not use Ping Pong Scheduling. Blackwell GEMM kernels have no explicit conception of Ping Pong Scheduling, since by default in cuBLAS and most kernel libraries, two accumulators are kept in the new tensor memory hierarchy, and MMA is computed on one accumulator while the epilogue is computed on the other.

[2] https://pytorch.org/blog/cutlass-ping-pong-gemm-kernel/

As a small implementation detail, note that the main diagonal of 256×256 cluster work tiles is part of the work assigned by the triangular scheduler. Since their transposed locations are identical to their current locations, we only write those values to general memory once—writing twice can cause inaccurate values or NaNs.

C.2 Implementation Strategy in Code

There are only two differences between the symmetric GEMM kernel and the standard GEMM kernel: the triangular scheduler and the transposed tile write in the epilogue. We design our symmetric kernel such that it is abstracted around the standard GEMM kernel to enable lightweight but maximally performant GEMM epilogue fusions. Using these abstractions, we are able to implement the symmetric GEMM kernel for both Hopper and Blackwell in just 160 lines, while achieving state-of-the-art performance.

We override the standard tile scheduler with our triangular scheduler and wrap the symmetric GEMM class around the GEMM with activation class. GEMM with activation itself is a wrapper around the Blackwell and Hopper default GEMMs. It supports writing two output tensors—the standard GEMM output (the preactivation) and the standard GEMM output with an activation function such as SwiGLU or ReLU applied (the postactivation). We define the activation function to be the identity and the postactivation tensor to be the inplace transpose of the preactivation tensor. Then, when the GEMM with activation class writes to the postactivation, it is really writing to the upper triangle with a transposed layout—this is exactly the intent of the symmetric GEMM kernel. We override the epilogue of GEMM with activation just to ensure we do not write twice to the diagonal tiles, for the correctness reasons mentioned previously.

C.3 Kernel Optimizations for Standard Newton-Schulz

Using just optimized CuteDSL kernels, we can accelerate standard Newton-Schulz with two changes.

Symmetric Matrix Multiplication As discussed above, the matrices $\boldsymbol{A} = \boldsymbol{X}\boldsymbol{X}^\top$ and $\boldsymbol{B} = b_t\boldsymbol{A} + c_t\boldsymbol{A}^2$ computed at each iteration of Newton-Schulz are symmetric by definition. Therefore, we use our symmetric GEMM kernels for these operations, reducing their FLOP cost by half.

Fused GEMM + Add The typical way to implement the non-symmetric multiplication $\boldsymbol{X} \leftarrow a_t\boldsymbol{X} + \boldsymbol{B}\boldsymbol{X}$ is to use `torch.baddbmm`, which calls cuBLAS under the hood. However, we offer a much faster implementation of this "Fused GEMM + Add" operation for Hopper. Unlike cuBLAS, our "Fused GEMM + Add" supports Ping Pong Scheduling for Hopper, which better hides the epilogue addition of $a_t\boldsymbol{X}$.

D Implementation Details of Dion3 Update Rule

Avoiding Downcasting Our initial implementation of Dion3 suffered from a nefarious numerical issue that worsened the final loss. Normally, the compiler fuses the weight update into a single operation $\boldsymbol{W} \leftarrow (1-\eta\cdot\texttt{wd})\boldsymbol{W} - \eta\boldsymbol{O}$. Tensors are cast from `bfloat16` to `float32`, the arithmetic is performed, and the result is cast back to `bfloat16`. When we introduced the row selection operator, it broke this automatic fusion, creating multiple rounds of upcasting and downcasting. To restore the original numerical behavior, we implemented a custom Triton kernel for Lines 7 and 8 of Algorithm 3. Furthermore, we found that NorMuon's normalization step should be performed in `float32` to avoid similar issues.

Finite Precision At $f = 1$, every row is selected and Dion3 reduces to Muon/NorMuon up to four numerical details: the rows get permuted / unpermuted, momentum damping is applied *after* forming the update[3], NorMuon's normalization steps run in `float32`, and the update step uses our custom Triton kernel. We confirmed that, despite these differences, the convergence curve of our optimizer matches that of Muon / NorMuon almost exactly (see Appendix G.5).

[3]That is, Newton-Schulz sees $\boldsymbol{M} + \boldsymbol{G}$ rather than $\mu\boldsymbol{M} + \boldsymbol{G}$. Since $\boldsymbol{M}$ is initialized to zero, these matrices differ by a constant factor of μ, which is removed by orthogonalization anyway.

Kernel launch minimization Dion3's row selection process necessarily requires more kernel launches than standard Muon does. At small scale, the cost of these extra kernel launches can dominate. Since the kernel dependency structure is the same from round to round, we use a standard CUDA graph capture and replay to avoid this overhead and realize the full potential of our optimizer on smaller-scale models.

E Motivating and Benchmarking Megabatching

This section describes the benefits of our megabatching strategy for implementing the orthogonalization step in distributed settings. While the update rule of Section 5 reduces the communication *volume* (the number of bits that must be assembled and scattered), megabatching reduces the number of *rounds* of communication. A naive implementation of Muon performs the update in batches of `world_size` matrices at a time; each matrix in the batch is assembled and orthogonalized in parallel on a different GPU without any GPU sitting idle. For a model with N orthogonalized matrices, this requires $O(N/\texttt{world_size})$ rounds of communication (all-to-alls) per optimizer step. Unfortunately, launching and synchronizing each round incurs overhead. Since this overhead is independent of the communication volume, our update rule does not mitigate it. Furthermore, each peer-to-peer message is small—just one shard of one parameter. These messages may fail to saturate the interconnect, so bandwidth utilization is poor. We demonstrate these effects below.

Our solution is megabatching; we group all matrices of the same shape into a *single* batch. For a given shape, the local momentum shards are packed into one all-to-all, assembled, orthogonalized as a batch, and scattered back together. Transformers contain only a handful of distinct weight shapes, so megabatching reduces the number of communication rounds to $O(1)$, independent of model depth. We implement this megabatching strategy in the `dion` package, where it powers Muon, NorMuon, and Dion3 alike. Note that megabatching only changes *when* data moves; the assembly, momentum-state layout, checkpoint format, and orthogonalization math are all the same.

Benchmarking all-to-all communications Above, we claim that reducing the number of communication rounds via megabatching is beneficial in two ways: (1) each round adds a fixed overhead not dependent on the size of the payload, and (2) small messages fail to saturate the interconnect. Here, we verify both claims directly with a standalone microbenchmark of NCCL `all_to_all` over NVLink, independent of any optimizer. On a single node of four H100s (SXM, NVLink-4 mesh, NCCL 2.29), we sweep the size of the payload per rank from 1 KiB to 1 GiB. We time each all-to-all with CUDA events (10 warmup, 50 timed iterations), taking the per-iteration time to be the maximum over ranks, and measure the bandwidth. We confirmed that the transport is a pure NVLink peer-to-peer communication, with no PCIe or network fallback and that the measured bandwidths agree with the `nccl-tests alltoall_perf` tool to within 1%. Figure 16 shows the results. Both effects are clearly visible. The latency floor is $\sim$25 μs (left panel), of which $\sim$15–17 μs is host-side dispatch (as measured separately with `perf_counter` and no device synchronization). Without megabatching, we pay this cost over and over. Bandwidth climbs by more than an order of magnitude as the per-link payload grows (right panel), so coalescing many small all-to-alls into one megabatch increases the effective bandwidth of each transfer.

Benchmarking Optimizer Step We compare megabatching against the previous communication strategy by measuring the wall-clock time of the optimizer step. We hold the model, data, and optimizer (Muon) fixed, and time both strategies back-to-back on the same GPUs. We test two model sizes (1B and 14B parameters) and two FSDP configurations (8 or 32 GPUs). Results are shown in Table 1. For the 14B model, the computational cost of Newton-Schulz dominates, so reducing communication overhead has little impact. For the 1B model with 32 shards, each rank holds only one or two matrices of each shape anyway (see Table 3), so the batch size is about the same for both strategies. However, for the 1B model with 8 shards, the optimizer is communication-bound and each rank holds many matrices. Here, megabatching has a major impact, reducing step time by 35%.

Compressed Data Parallelism There is a prospect for further savings at larger scales, where data parallelism spans multiple pods in a data-center network (DCN) or hybrid sharding is used (Zhao et al., 2023). A common deployment combines model parallelism within an inter-chip interconnect (ICI) with pure data parallelism across pods (Austin et al., 2025). Because DCN bandwidth is typically

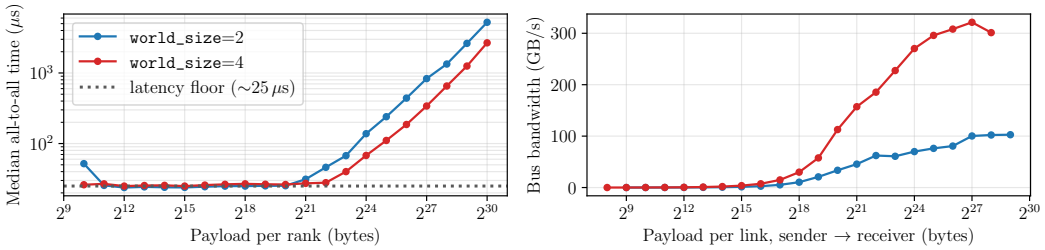


Figure 16: NCCL `all_to_all` over NVLink on 4×H100. **Left:** median all-to-all time versus payload per rank (sender) over 50 trials. For small payloads, runtime is dominated by a fixed latency. **Right:** bus bandwidth versus payload per link (sender → receiver). Bandwidth is near zero for small messages and does not reach 80–90% of its peak until the per-link payload is ∼16–32 MiB. At `world_size` = 4, a 256 KiB per-link payload achieves only ∼10% of the peak bandwidth.

Table 1: Effect of megabatched communication on the per-GPU optimizer step time (Muon, median over steps). "Batching" orthogonalizes matrices in groups whose size equals the shard count; "megabatch" uses a single batch per shape group. When the weights are small (so Newton-Schulz is cheap) and each rank holds several matrices, megabatching substantially reduces optimizer time. Each node has four GH200s.

Model size	Nodes	FSDP Shards	Batching (ms)	Megabatching (ms)	Change
1B	1	8	80.7	52.1	−35%
1B	4	32	61.9	59.3	−4%
14B	1	8	144.0	140.8	−2%
14B	4	32	94.7	89.1	−6%

far smaller than ICI bandwidth, shrinking the volume of data-parallel communication is especially valuable in this regime. Here, Dion3 allows for compressed data-parallel synchronizations analogous to those of DION (Ahn et al., 2025, §3.3). Ordinarily, gradients are averaged across data-parallel replicas at every step so that the momentum buffers, and hence the weights, stay consistent. With Dion3, however, we need not synchronize the entire momentum matrix at every step, only the selected submatrix $\boldsymbol{M}[\mathcal{S}, :]$. When the rows $\mathcal{S}$ can be selected without first synchronizing $\boldsymbol{M}$ (e.g., by picking $\mathcal{S}$ at random), this strategy performs an identical update at a fraction of the communication cost.

F ALTERNATIVE EXPERIMENTS ON GRAM NEWTON-SCHULZ

This section presents additional experimental results about Gram Newton-Schulz and the symmetric GEMM kernels. Complementing Section 7, these experiments study a wider range of architectures but at smaller scale, using only one GPU at a time.

F.1 SETUP

We train four architectures: Llama-430M, Qwen-600M, Gemma-1B, and a custom MoE architecture with 1B parameters, of which ∼20% are active (Gemma Team et al., 2025; Grattafiori et al., 2024; Yang et al., 2025). We train on FineWeb-Edu (Lozhkov et al., 2024). The number of training tokens for each dense model is given by the Chinchilla scaling law; for MoE-1B it is twice the Chinchilla scaling law with respect to the active parameters. We use a cosine learning rate scheduler with the base learning rates given in Table 2.

We use Muon on most matrix parameters—the attention layer's projection matrices ($\boldsymbol{W}_Q, \boldsymbol{W}_K, \boldsymbol{W}_V$), the projection following attention ($\boldsymbol{W}_O$), the SwiGLU MLP weights ($\boldsymbol{W}_{\text{MLP,UP}}$, $\boldsymbol{W}_{\text{MLP,GATE}}$, $\boldsymbol{W}_{\text{MLP,DOWN}}$), and the token router of the MoE ($\boldsymbol{W}_{\text{router}}$)—but exclude the embedding and unembedding layers. Although we are not in the distributed setting, we use megabatching (Section 6), as batching Newton-Schulz's GEMM operations makes them faster. As is standard for Muon, we adjust the learning rate by scaling the update for each weight matrix based on its dimensions. For these

experiments, we found that Moonshot AI's strategy of scaling by $0.2\sqrt{\max(\text{fan_out}, \text{fan_in})}$ yields the best loss curves (Liu et al., 2025, §2.2).

Table 2: Architecture, base learning rate, and training-token budget of each model.

Model	Hidden dim	Layers	MLP dim	Learning rate	Training tokens
Llama-430M	1024	24	4096	3×10^{-3}	10B
Qwen-600M	1024	28	3072	1.5×10^{-3}	12B
Gemma-1B	2048	8	16384	3×10^{-4}	22B
MoE-1B	1024	9	(per-expert) 256	2.5×10^{-3}	11B

F.1.1 Splitting the Weights

We draw special attention to the fact that we split $\boldsymbol{W}_{\text{MLP,UP}}$ from $\boldsymbol{W}_{\text{MLP,GATE}}$ and orthogonalize them separately. Ordinary implementations of SwiGLU MLPs concatenate these into a single weight matrix; however, their contributions to the activation are fundamentally different, so gradients are different too. We find that orthogonalizing them separately improves the final loss; for example, in Llama-430M, we observe an improvement of ≈ 0.2 in perplexity. For MoE architectures—whose intermediate size is typically smaller than the hidden size—separating them also reduces the FLOP cost of orthogonalization by a factor of two for standard Newton-Schulz and even more for Gram Newton-Schulz. Likewise, while earlier implementations of Muon orthogonalized the combined matrix $[\boldsymbol{W}_Q \mid \boldsymbol{W}_K \mid \boldsymbol{W}_V]$, we orthogonalize each piece separately.

We are also aware that in some settings, including pretraining of GLM-5, Muon benefits from splitting the Multi-Latent Attention weights ($\boldsymbol{W}^{UQ}$, $\boldsymbol{W}^{UK}$, and $\boldsymbol{W}^{UV}$) by attention head before orthogonalizing (GLM-5 Team et al., 2026). This choice is principled, since the actual function computed by attention treats each head separately; it does not see the concatenated weights. Inspired by this, we experimented with splitting $\boldsymbol{W}_Q$, $\boldsymbol{W}_K$, $\boldsymbol{W}_V$, and $\boldsymbol{W}_O$ by attention head to form H matrices each of size $\frac{d}{H} \times d$, where d is the embedding dimension and H is the number of heads. However, this design led to higher losses throughout training, so we did not adopt it.

Still, we believe that there are other settings like GLM-5 where this strategy works well. Such cases would benefit *immensely* from Gram Newton-Schulz, since the aspect ratio of these weight matrices would be the number of heads H. For a standard attention weight like $\boldsymbol{W}_Q$ with $H = 16$ and $T = 5$, Gram Newton-Schulz on the little matrices would use $\mathbf{80}\times$ fewer FLOPs than orthogonalizing the big matrix!

F.2 Kernelized Gram Newton-Schulz preserves quality

We first verify that switching from standard Newton-Schulz to our kernelized implementation of Gram Newton-Schulz does not affect training quality. We try each implementation with both the Polar Express coefficients (Amsel et al., 2026) and those of Cesista et al. (2025). As Figure 17 shows, the loss curves are identical in all cases, and the final validation perplexity is preserved to within 0.01. Figure 17 uses a Hopper GPU, but we got the same results on Blackwell.

We see loss preserved as follows, when both using the Polar Express coefficients and the coefficients derived by Cesista et al. (2025):

F.3 Kernelized Gram Newton-Schulz speeds up the optimizer step

Newton-Schulz Performance We observe that our method speeds up the runtime of the Newton-Schulz step by 1.5–$2\times$. Figure 18 reports these speed-ups for each model, benchmarked on both H100 and B300 GPUs. As expected, savings are greatest for weights that are highly rectangular, like Gemma's MLP weights ($\alpha = 8$) and MoE-1B's expert weights ($\alpha = 4$). Note that these experiments use standard Newton-Schulz as the fallback when $m = n$.

End-to-End Optimizer Performance Figure 19 shows the end-to-end wall clock time of the Muon optimizer step for each version of Newton-Schulz, along with that of AdamW. For Muon, these timings include updating the momentum matrix, learning rate scaling, applying the weight update,

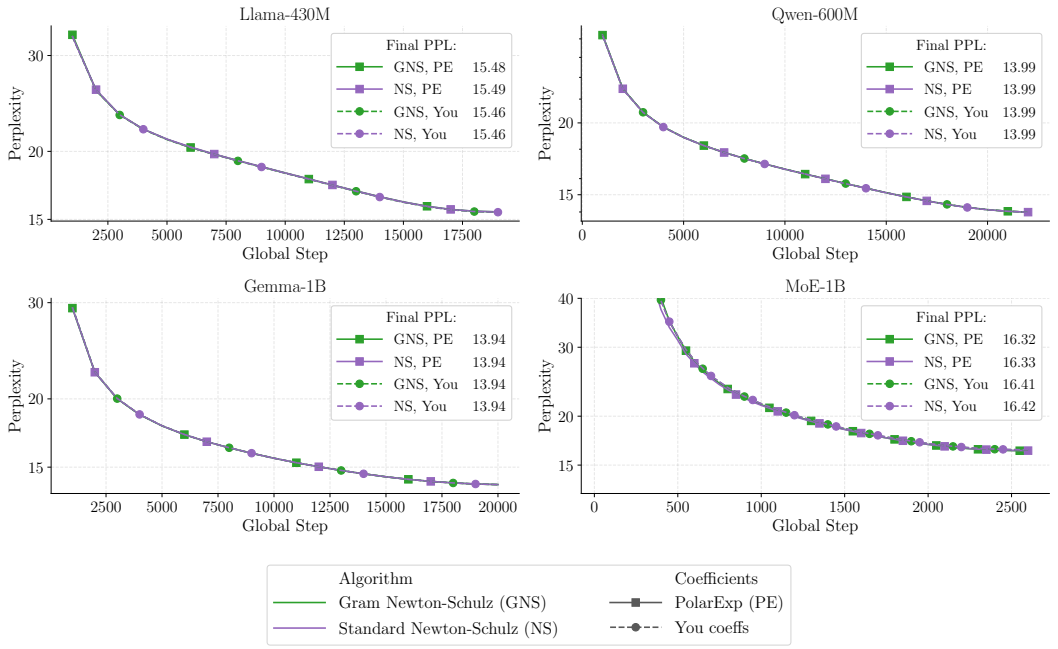


Figure 17: When training with Muon on a Hopper GPU, switching from standard Newton-Schulz to Gram Newton-Schulz preserves the validation perplexity throughout training (up to a 0.01 difference in GNS's favor). Setup is described in Appendix F.1.

and the AdamW updates for weights not assigned to Muon (such as the embedding layer and the vector-valued weights). Our method yields a 1.3–2× speedup, with both Gram Newton-Schulz and the kernels contributing significantly. As before, the impact of Gram Newton-Schulz is greatest for architectures with highly rectangular weights. Due to its MLP's higher $8\times$ aspect ratio, Gemma-1B sees the largest speedup.

Estimating Gram Newton-Schulz time in Kimi K2 Kimi K2 is a trillion parameter sparse, fine-grained MoE model with 384 experts per layer, a hidden size of 7168, and a small expert intermediate dimension of 2048. Since models are trending towards finer-grained MoE architectures and Kimi K2 was trained with Muon, this is a perfect setting to benchmark Gram Newton-Schulz. Due to Kimi's sophisticated pipeline parallel strategy, the optimizer steps of many of its weights are completely hidden behind the backward pass of the next pipeline stage. We estimate that orthogonalization steps of following weights are exposed: 216 expert up/gate/down weights of shape 2048×7168 and 1 dense up/gate/down weight of shape 7168×18432. Therefore, we estimate the exposed Newton-Schulz time by orthogonalizing weights of these shapes on a single GPU. Figure 20 shows the results: our method yields a $2\times$ speedup. As before, both our kernels and the Gram Newton-Schulz algorithm contribute significantly.

G Additional Experiments

G.1 Architecture and Optimization Setup

We now describe details of the experiments in Section 7.1. (The benchmarks in Section 7.2 are similar, but with minor architectural differences and synthetic data.)

Architecture. We use slightly different recipes for the "smaller-scale runs" and "larger-scale runs" described in Section 7.1. Both architectures are decoder-only dense transformers (not mixtures of experts) that use RMS normalization layers and rotary positional embeddings. Smaller-scale runs use full attention at sequence length 1024, vocabulary size 50,304, and ReLU^2 non-linearities.

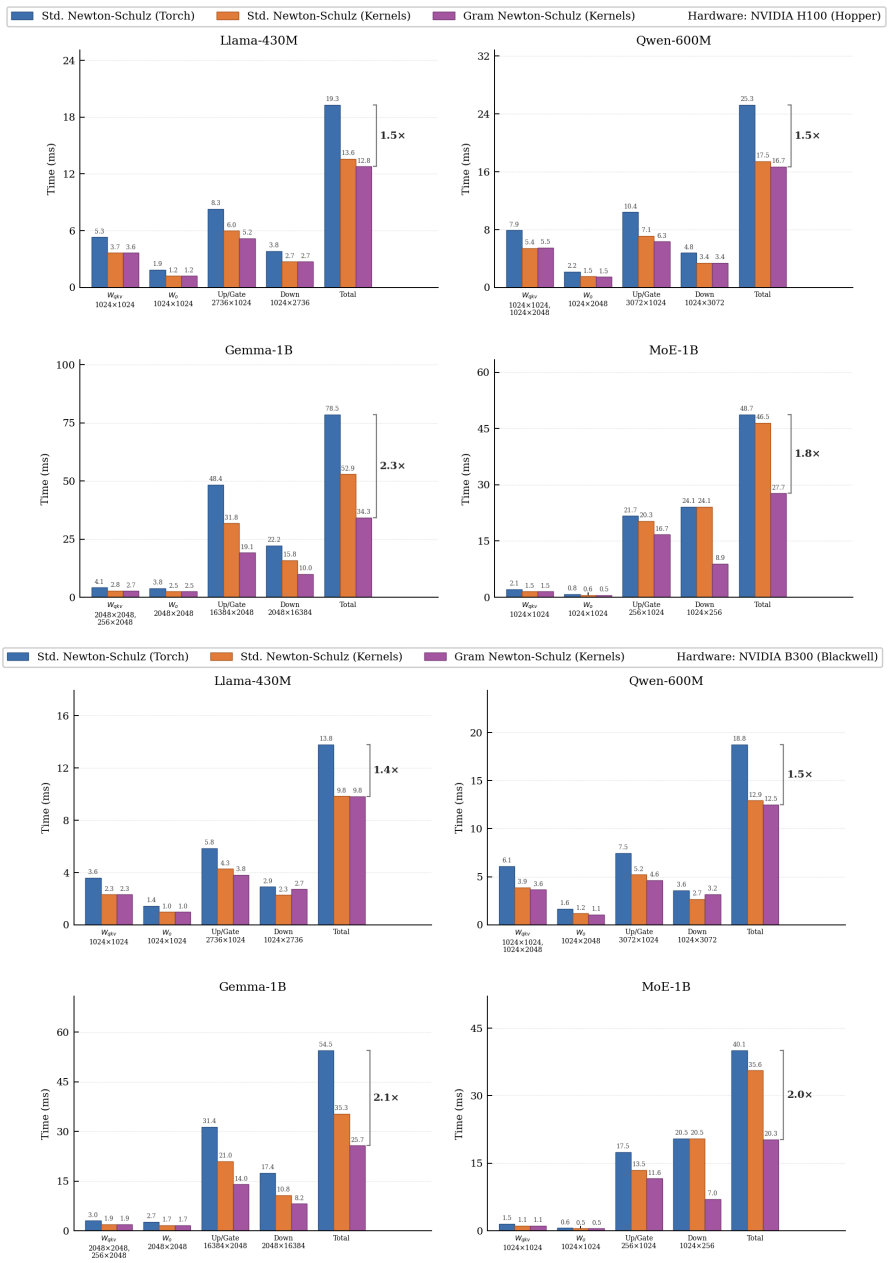


Figure 18: Newton-Schulz time per model weight for (1) standard Newton-Schulz implemented in pure PyTorch, (2) standard Newton-Schulz with our symmetric GEMM kernels, and (3) Gram Newton-Schulz with our kernels. We test four architectures on a single Hopper (top) or Blackwell (bottom) GPU. Square weights ($\boldsymbol{W}_Q$ or $\boldsymbol{W}_K$) benefit from the kernels. Rectangular weights—especially those with a high aspect ratio (e.g. Up/Gate in Gemma-1B)—additionally benefit from Gram Newton-Schulz.

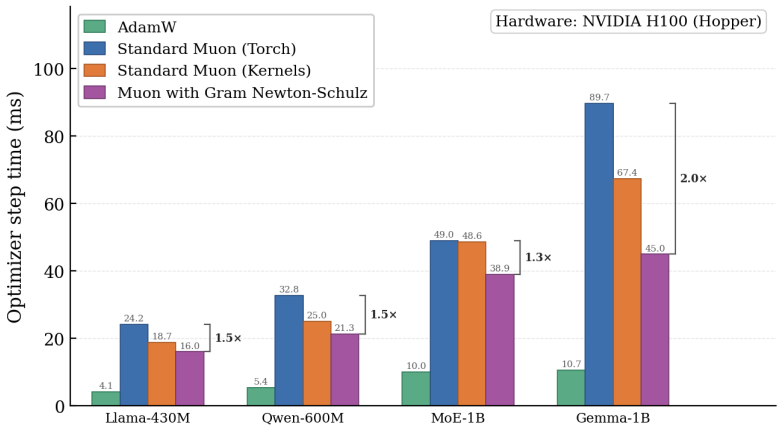


Figure 19: Optimizer step time for AdamW and Muon with three different Newton-Schulz routines on a single H100. Gram Newton-Schulz with our symmetric kernels gives a 1.3–2× speedup over standard Newton-Schulz. Timings include matrix splitting and recombination for QKV and MLP, learning rate scaling, weight updates, and the scalar optimizer (AdamW) step for non-2D weights.

Larger-scale runs use sliding-window grouped-query attention (window 2048) on sequences of length 8192, vocabulary size 100,352, SwiGLU non-linearities, and gated attention; in addition, weights are stored and updated in MXFP8. Table 3 lists the per-scale dimensions for both recipes.

Table 3: Model dimensions by scale. First three rows correspond to smaller-scale runs; remaining rows correspond to larger-scale runs.

Scale	Hidden dim	Layers	Heads	KV heads	MLP hidden dim
225M	896	14	7	7	3584
418M	1280	18	10	10	5120
1B	1792	24	16	16	7168
1B	1536	24	16	8	6656
3B	2304	32	18	6	9856
4B	2560	36	32	8	10752
7B	4096	32	32	8	11008
14B	5120	40	40	8	14592

Optimization. Matrix parameters are updated by the orthogonalizing optimizer under study—Muon, NorMuon, or their filtered variants. As always, Newton-Schulz and Gram Newton-Schulz run in half-precision arithmetic. For matrix parameters, we multiply the base learning rate by $\sqrt{d_{\text{out}}/d_{\text{in}}}$. Embeddings, the language-model head, norm scales, and biases are updated by AdamW ($\beta_1 = 0.9$, $\beta_2 = 0.95$) at the base learning rate, without the $\sqrt{d_{\text{out}}/d_{\text{in}}}$ scaling applied to the matrix parameters (the language-model head, in particular, also uses the base rate). We apply weight decay 0.01 to the matrix parameters only; the AdamW (non-matrix) groups use no weight decay. The learning-rate schedule is warmup–stable–cooldown: a 200-step linear warmup to the peak learning rate, a constant phase, and a cooldown over the final 25% of training. All reported held-out cross-entropies are taken at the end of cooldown.

G.2 Learning-Rate Transfer Rule for Dion3

Since Dion3 updates only a fraction of the rows in each step, it reduces the effective step size. This in turn changes the optimal learning rate. To understand this effect, we sweep both the fraction $f \in \{1/2, 1/4, 1/8, 1/16\}$ and the learning rate η of Dion3, along with our earlier sweep for NorMuon ($f = 1$). The results in Figure 21 show a clear trend: the optimal learning rate for a given f scales as $\eta \propto \sqrt{1/f}$. We can explain this transfer rule as follows. For Muon, the size

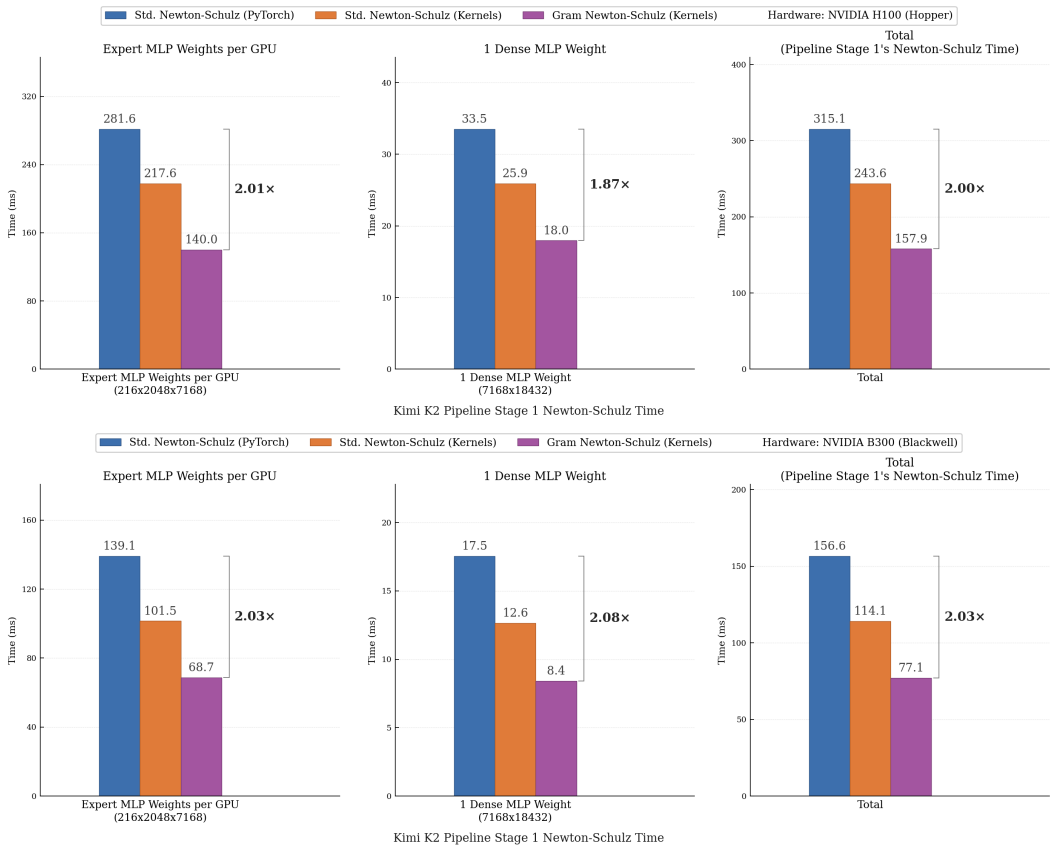


Figure 20: Estimated exposed Newton-Schulz time for one step of Kimi K2 with pipeline parallelism. We measure the runtime of the exposed operations on a single Hopper (top) or Blackwell (bottom) GPU. Gram Newton-Schulz with kernels is $2\times$ faster than standard Newton-Schulz.

of the update is $\|\eta \cdot \mathrm{polar}(\boldsymbol{M})\|_{\mathsf{F}} = \eta\sqrt{n}$, where n is the smaller dimension. For Dion3, it is $\|\eta' \cdot \mathrm{polar}(\boldsymbol{M}[\mathcal{S}, :])\|_{\mathsf{F}} = \eta'\sqrt{fn}$. For these to match, we must set $\eta' = \eta/\sqrt{f}$. We do not bother retuning the momentum coefficient; we simply copy the optimal setting from NorMuon ($\mu = 0.9$).

While this learning rate transfer rule works beautifully for the "larger-scale runs " with 1B-paramter models, 100B tokens of trainin data, it does not hold in the smaller-scale experiments. Results of those runs are also more sensitive to sub-optimal learning rates.

G.3 Dion3 versus NorMuon Across Model Sizes

Table 4 compares NorMuon and Dion3 across different model sizes, scaling up from the 1B-parameter experiments reported in Section 7.1. Figure 22 shows that, for the 14B parameter model, validation loss is lower throughout training when using Dion3. This matches the results of the 1B-parameter, 100B token experiment of Figure 4.

G.4 Finer-Grained Timing Metrics

The main text reports GPU (device) time, the metric of interest for the orthogonalization compute. Here we report three per-step metrics side by side:

- **GPU (device) time**, read from CUDA events between two stream markers around `optimizer.step()`;

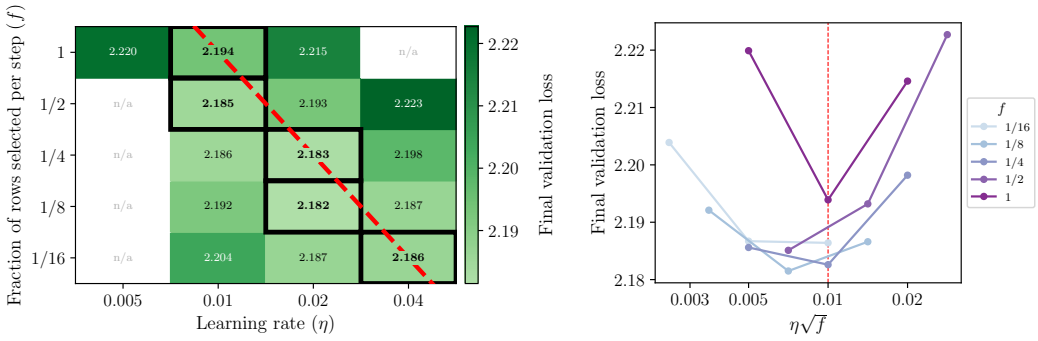


Figure 21: **Left:** Final validation loss for 1B-parameter models trained on 100B tokens of ClimbMix with NorMuon or Dion3, as a function of row-selection fraction f and learning rate η. NorMuon is $f = 1$. Bolded cell in each row shows best η. Optimal ηs track the line $\eta\sqrt{f} = 0.01$, shown in red on a log-log scale. Not shown: $f = 1, \eta = 0.03$ (loss = 2.237) and $f = 1/32, \eta = 0.04$ (loss = 2.191). **Right:** The same data reduced to two dimensions: validation loss as a function of $\eta\sqrt{f}$. Final loss is minimized when $\eta\sqrt{f} \approx 0.01$.

Table 4: Dion3 ($f = 1/4$, $\eta = 0.02$) versus NorMuon ($\eta = 0.01$) across model sizes trained on 10B tokens of ClimbMix: final validation loss and downstream accuracy (macro-average over 12 standard benchmarks: ARC easy/challenge, BoolQ, COPA, HellaSwag, LAMBADA, MMLU, OpenBookQA, PIQA, RTE, TruthfulQA, WinoGrande; higher is better). The winner of each comparison is bolded.

Model Size	Final Validation Loss			Downstream Accuracy (%)		
	NorMuon	Dion3 ($f = 1/4$)	Δ	NorMuon	Dion3 ($f = 1/4$)	Δ
3B	2.269	**2.257**	-0.012	53.9	**54.9**	$+1.0$
4B	2.243	**2.232**	-0.011	**55.2**	54.9	-0.3
7B	2.220	**2.206**	-0.014	56.0	**56.1**	$+0.1$
14B	2.189	**2.162**	-0.027	57.4	**58.1**	$+0.7$

- **CPU (host) time**, from `time.perf_counter()` stopped immediately after `step()` returns and before any device sync — cost of kernel launch, Python, and any host-side syncs the optimizer performs;
- **Communication volume**, the number of bytes moved by the megabatch collectives (`all_to_all`, `all_gather`, `reduce_scatter`) at each step, measured by tallying the receive-side payload per-GPU.

As always, Newton-Schulz runs in half precision arithmetic. Results for a 14B model sharded over eight B200s and a 7B model on four GH200s appear in Table 5 and Table 6, respectively. Unlike in Section 7.2, we show all four combinations of kernel and Newton-Schulz variant. We see that Gram Newton-Schulz *without* symmetric kernels achieves up to a $1.2\times$ speedup. Otherwise, GPU timings are consistent with the results presented in Section 7.2. The NorMuon family has higher costs overall due to the overhead of its extra normalization steps. Our contributions still yield similar speedups: up to $1.5\times$ for symmetric kernels and Gram Newton-Schulz, $3.3\times$ for $f = 1/2$, and $5.6\times$ for $f = 1/4$. Table 5 shows that the CPU cost is negligible—just $\sim$0.1 ms across configurations. This near-constant host cost is a direct consequence of CUDA-graph capture. Row selection, error feedback, and Gram Newton-Schulz issue many small operations and kernel launches. Without capture, the host would dispatch each one individually, and this per-launch overhead would dominate, especially for the filtered configurations, whose device work is small. Capture and replay instead collapse the entire step into a single graph launch, so the host issues one call regardless of how many kernels the step contains or how much they compute. The reported host time is therefore just the fixed cost of that one launch ($\sim$0.1 ms), essentially independent of the configuration. Finally, communication volume is directly proportional to the fraction f, as expected.

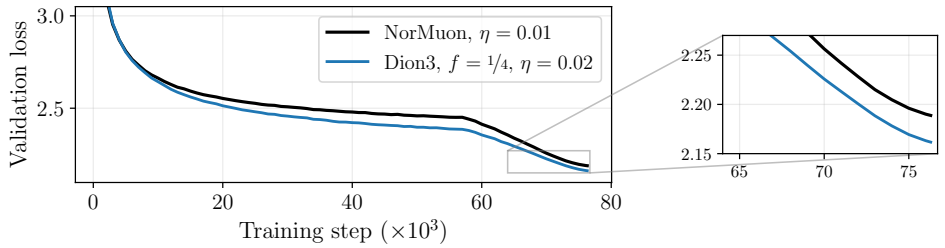


Figure 22: Validation loss over training at 14B for Dion3 ($f = 1/4$, $\eta = 0.02$) versus NorMuon ($\eta = 0.01$), trained on 10B tokens of ClimbMix. Dion3 tracks below NorMuon throughout.

Table 5: Fine-grained per-step metrics for the distributed 14B / 8-GPU benchmark: GPU (device) time from CUDA events, CPU (host) time from `perf_counter` stopped before the device sync, and per-GPU communication volume of one step. We distinguish the GPU time between the Muon and NorMuon families of optimizers; CPU time and communication volume are reported once because neither depends on the family: host time is the fixed CUDA-graph launch cost, and communication volume is set by the parameter shapes and the fraction f.

				GPU (ms)			
Configuration	Kernels	Algorithm	NS frac.	Muon	NorMuon	CPU (ms)	Comm (MB)
Standard Newton-Schulz	PyTorch	Standard	1	153.8	174.1	0.1	5479
+ symmetric kernels	CuteDSL	Standard	1	132.7	153.4	0.1	5479
+ Gram NS	PyTorch	Gram	1	137.6	158.5	0.1	5479
+ kernels + Gram NS	CuteDSL	Gram	1	106.8	127.0	0.1	5479
+ filtering ($f = 1/2$)	CuteDSL	Gram	$1/2$	66.4	73.8	0.1	2740
+ filtering ($f = 1/4$)	CuteDSL	Gram	$1/4$	34.8	38.1	0.1	1370

G.5 Ablations

Tuning the baseline The experiments in Section 7.1 compare Dion3 to a NorMuon baseline. Table 7, below, demonstrates that this baseline is properly tuned for a fair comparison.

Dion3 with no compression As a control, we run Dion3 with $f = 1$, which should reduce to plain NorMuon. Figure 23 confirms this at 1B; the two curves coincide throughout training, and their final validation losses differ by 0.0005.

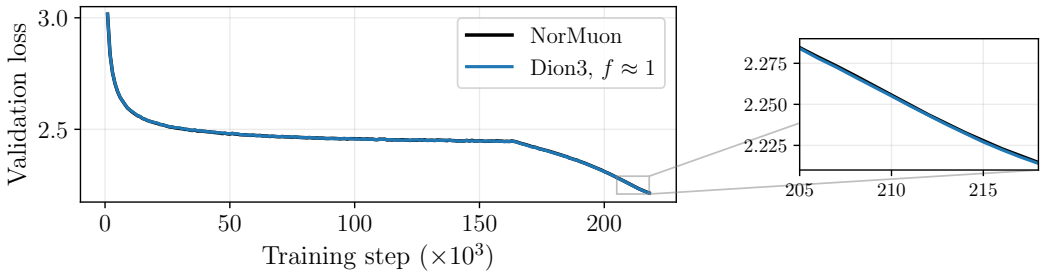


Figure 23: Validation loss at 1B for NorMuon and for Dion3 at $f \approx 1$, on 100B tokens of ClimbMix. As expected, Dion3 tracks NorMuon throughout, ending at 2.2141 versus 2.2146.

Timings with NorMuon family Section 7.2 compares the runtime of Muon to the Muon version of Dion3. Here, in Figure 24, we repeat this benchmark with the NorMuon family. Results are quite alike, but due to the extra overhead of NorMuon's normalization steps, it benefits slightly less from our contributions, which target the orthogonalization step.

Table 6: Replication of Table 5 for a 7B-parameter model sharded over a node of four GH200s with NVLink.

Configuration	Kernels	Algorithm	NS frac.	GPU (ms)		Comm (MB)
				Muon	NorMuon	
Standard Newton-Schulz	PyTorch	Standard	1	360.6	385.4	6442
+ symmetric kernels	CuteDSL	Standard	1	274.4	315.0	6442
+ Gram NS	PyTorch	Gram	1	296.6	321.1	6442
+ kernels + Gram NS	CuteDSL	Gram	1	217.7	252.1	6442
+ filtering ($f = 1/2$)	CuteDSL	Gram	$1/2$	101.5	116.7	3221
+ filtering ($f = 1/4$)	CuteDSL	Gram	$1/4$	59.6	68.6	1610

Table 7: Tuning hyperparameters (learning rate η, momentum μ) of NorMuon at 1B. Final validation loss on 100B tokens of ClimbMix. Winner: $\eta = 0.01$, $\mu = 0.9$ (bold).

Sweep η (at $\mu = 0.9$)	
η	CE
0.005	2.220
0.01	**2.194**
0.02	2.215
0.03	2.237

Sweep μ (at $\eta = 0.01$)	
μ	CE
0.9	**2.194**
0.949	2.201
0.974	2.217
0.987	2.238

H Case Studies of End-to-End Training Time

The share of end-to-end training time taken up by Newton-Schulz can vary widely depending on the training setup. To explain this variability, we analyze two idealized scenarios. In one, standard Newton-Schulz takes 2% of training time; in the other it takes 17%.

Case Study 1: Standard Newton-Schulz takes 2% of Kimi K2 training time

The following analysis gives a very optimistic estimate of the optimizer's wall clock time. We assume an efficient training infrastructure with highly optimized pipeline parallelism. Moreover, we assume that the optimizer step of each pipeline stage is completely hidden behind the backward pass of the next pipeline stage.

Kimi K2 Thinking is a 1.1 trillion parameter model with 32 billion active parameters. It has 1 dense layer followed by 60 MoE layers (Kimi Team et al., 2025). It is pretrained with 256-GPU model parallel groups, 16-way pipeline parallelism, 16-way expert parallelism within each pipeline stage, and a huge batch size of 67 million tokens.

We use a single H100 to approximate the share of each training step's runtime occupied by Newton-Schulz in this setting under the following assumptions:

1. The training cluster consists of 256 nodes of eight H100s each (2048 GPUs in total), connected with NDR 400 Gb/s InfiniBand inter-node (8 NICs per node, 1:1 NIC-to-GPU ratio) and NVLink 4.0 intra-node. This is the size of the cluster used to train DeepSeek-V3, with upgraded hardware (DeepSeek-AI et al., 2025).
2. Training in `bfloat16` attains 40% model flop utilization (MFU), which is typical for MoEs at this scale on H100s.
3. The only non-overlapped optimizer time is that of the last pipeline stage to complete its backward pass (i.e., pipeline stage 1 of 16). The optimizer steps of pipeline stages 2 to 16 are fully hidden behind the backwards of stages 1 to 15.
4. Pipeline stage 1 contains the dense layer and 3 MoE layers.

Under these assumptions, the optimal way to partition the Newton-Schulz work of pipeline stage 1 is as follows. Each of the 16 GPUs in pipeline stage 1's expert parallel group gets

$$\frac{384 \text{ experts/layer} \times 3 \text{ MoE layers}}{16 \text{ GPUs}} = 72 \text{ experts/GPU} = 216 \text{ expert up-gate-down/GPU}.$$

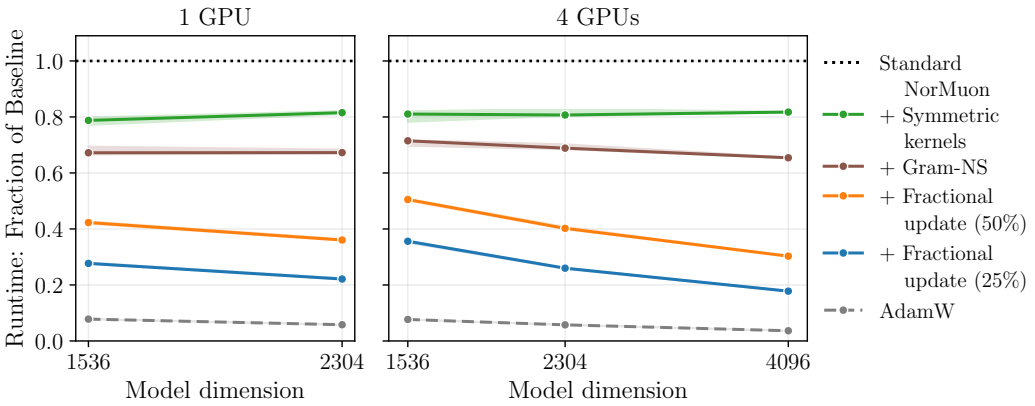


Figure 24: Optimizer step time (excluding forward/backward pass) relative to standard Muon across model scales when training on 1 GH200 (left) and FSDP over 4 GH200s (right). Each colored line adds one of our contributions on top of the previous one; AdamW is shown for reference. Lines show median over 25 steps and bands show interquartile range. Compare to Figure 5.

Each of the 16 GPUs has its own unique expert weights, so no communication is needed. Pipeline stage 1 also contains the dense MLP's three 7168×18432 weights (up/gate/down) and three shared experts. Orthogonalizing the dense MLP weights is the dominant cost, so they are sent to three different GPUs; the shared experts are split amongst the remaining 13 GPUs. Thus, the orthogonalization time for pipeline stage 1 is the time it takes for a single GPU to orthogonalize 216 expert up/gate/down weights and 1 dense up/gate/down weight. As benchmarked in Figure 20, standard Newton-Schulz implemented in PyTorch takes 315 ms to do this. Per our assumption, stage 1's orthogonalization step is the only one that is not overlapped.

Having estimated the Newton-Schulz time, we now estimate the end-to-end wall clock time of an entire Kimi K2 global training step. We use the standard estimate of $6NB$ FLOPs for the forward and backward pass, where N is the number of active parameters and B is the global batch size. Given

- Active parameters: $N = 32 \cdot 10^9$
- H100 peak: $P = 989 \cdot 10^{12}$ FLOP/s
- Model flop utilization: MFU $= 40\%$
- Cluster size: $G = 2048$ GPUs
- Global batch size: $B = 67 \cdot 10^6$ tokens

we have

$$\text{sec/batch} = \frac{6N \times B}{P \times \text{MFU} \times G} = 15.9$$

Thus, Newton-Schulz takes approximately $\frac{315 \text{ ms}}{15900 \text{ ms} + 315 \text{ ms}} = 1.9\%$ of total pretraining wall clock time in this setting.

Case Study 2: Standard Newton-Schulz takes 17% of Llama3-70B SFT time

Llama3-70B is an 80-layer dense model with hidden size 8192, intermediate size 28672, and grouped query attention with $\boldsymbol{W}_K, \boldsymbol{W}_V$ of size 1024×8192 and $\boldsymbol{W}_Q, \boldsymbol{W}_O$ of size 8192×8192 (Grattafiori et al., 2024). Supervised finetuning (SFT) typically uses small batch sizes, ranging from 32 to 256 sequences (DeepSeek-AI et al., 2025).

We analyze the following SFT case:

1. Training uses 32 H100s across 4 nodes (8 GPUs per node).

2. Training in `bfloat16` hits 40% MFU.
3. Weights are sharded evenly across GPUs using FSDP, and the exposed Newton-Schulz time is that of 80 layers$/32$ GPUs ≈ 3 layers. Each layer has 3 MLP weights (up, gate, down), and the attention weights $\boldsymbol{W}_Q$, $\boldsymbol{W}_K$, $\boldsymbol{W}_V$, and $\boldsymbol{W}_O$.

According to our benchmarking, standard Newton-Schulz of

- nine 8192×28672 weights takes 739 ms,
- six 8192×8192 weights takes 156 ms,
- six 1024×8192 weights takes 2.32 ms,

totaling 897 ms. Given

- Parameters: $N = 70 \cdot 10^9$
- H100 peak: $P = 989 \cdot 10^{12}$ FLOP/s
- Model flop utilization: MFU $= 40\%$
- Cluster size: $G = 32$ GPUs
- Global batch size: $B = 64$ sequences $\times$ 2048 tokens/sequence $= 131{,}072$ tokens

we have

$$\text{sec/batch} = \frac{6N \times B}{P \times \text{MFU} \times G} = 4.35 \text{ s}$$

Thus, Newton-Schulz takes approximately $\frac{897 \text{ ms}}{4350 \text{ ms} + 897 \text{ ms}} = 17\%$ of total SFT wall clock time in this setting.