REPARAMETRIZING SHAMPOO AND SOAP

Anonymous authors
Paper under double-blind review

ABSTRACT

Shampoo-based methods, such as KL-Shampoo and SOAP, have demonstrated strong performance in neural network training, but their QR-based updates become costly for large preconditioning matrices. To address this, we propose a preconditioner reparametrization that accelerates full-basis QR computation and enables faster partial basis updates. By maintaining preconditioning factors in the current orthogonal basis, our approach exposes useful input structure to facilitate QR decomposition and allows dynamically selected basis vectors to be cheaply updated via subspace QR while preserving the full-basis orthogonality. To further accelerate QR computation, we provide a customized implementation supporting mixed-precision computation and batched execution. The resulting framework unifies full-basis and subspace updates and applies broadly to QR-based updates, including KL-Shampoo, SOAP, and KL-SOAP. Empirically, our approach reduces computational overhead and improves the training performance of SOAP-type methods in mixed precision, enabling KL-SOAP to match or exceed KL-Shampoo. Overall, our approach improves the time efficiency of Shampoo-based methods.

1 INTRODUCTION

Shampoo-based methods, including Shampoo (Gupta et al., 2018; Shi et al., 2023), SOAP (Vyas et al., 2025), and KL-Shampoo/KL-SOAP (Lin et al., 2026), have demonstrated strong performance in neural network training (Dahl et al., 2023; Kasimbeg et al., 2025; Eschenhagen et al., 2026). These methods use non-diagonal preconditioning and exploit Kronecker structure for matrix-valued weights. Pure Shampoo methods typically compute inverse matrix roots by eigendecomposing each Kronecker factor, $\boldsymbol{S}_i = \boldsymbol{Q}_i \mathrm{Diag}(\boldsymbol{\lambda}_i)\boldsymbol{Q}_i^\top$ (Shi et al., 2023; Eschenhagen et al., 2025), whereas SOAP-type methods run Adam in the eigenbasis. Computing these bases is costly for large factors (see fig. 1).

QR-based updates offer a cheaper way to compute an approximate eigenbasis (Vyas et al., 2025; Eschenhagen et al., 2025; Lin et al., 2026), but QR remains the main computational bottleneck (see fig. 1, right). Basis-free approaches based on Newton–Schulz-type iterations leverage mixed-precision and batched computation to compute preconditioned directions without forming $\boldsymbol{Q}_i$ (Amsel et al., 2026; Zhang et al., 2026). However, these approaches often require a sequential inner loop of matrix multiplications at every optimization step and do not directly apply to SOAP-type methods, which require an explicit basis.

An alternative is to retain the explicit basis while exposing useful input structure through a different factor representation. This can accelerate basis updates even with existing decomposition routines, while customized QR implementations can provide further gains by exploiting this structure with mixed-precision and batched computation. Updating only selected basis vectors can further reduce overhead, but requires preserving orthogonality with the unchanged vectors. The challenge is to expose the structure needed for efficient full-basis and dynamically selected subspace updates within the maintained optimizer state.

We address this challenge through a preconditioner reparametrization. For each Kronecker factor, we maintain $\boldsymbol{P}_i := \boldsymbol{Q}_i^\top \boldsymbol{S}_i \boldsymbol{Q}_i$, replacing $\{\boldsymbol{Q}_i, \boldsymbol{S}_i\}$ with $\{\boldsymbol{Q}_i, \boldsymbol{P}_i\}$, and update $\boldsymbol{P}_i$ directly without materializing $\boldsymbol{S}_i$. Importantly, $\boldsymbol{P}_i$ inherits symmetric positive definiteness from $\boldsymbol{S}_i$ and is nearly diagonal when $\boldsymbol{Q}_i$ closely approximates an eigenbasis. This representation reduces basis-update time even with standard PyTorch QR routine and enables orthogonality-preserving updates through selected submatrices of $\boldsymbol{P}_i$. Our customized QR implementation further leverages the structure of $\boldsymbol{P}_i$ to accelerate these updates.

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.

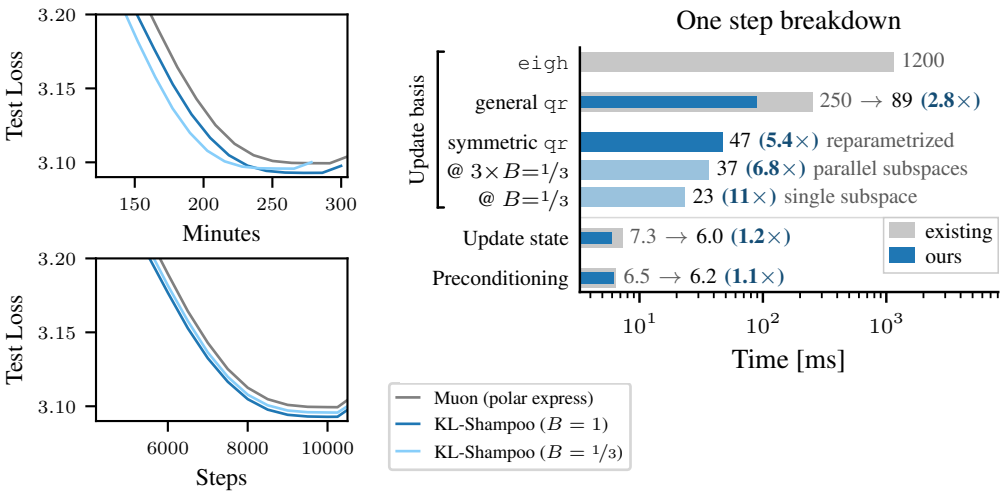

Figure 1: **Left:** Performance of reparametrized full-basis and subspace KL-Shampoo (with the BFP16 mixed-precision training) on Llama3 (598M) trained on FineWeb 4.1B, evaluated with the hyperparameters tuned for full-basis KL-Shampoo over 120 runs. Muon is tuned independently over 120 runs. **Right:** Per-step cost of KL-Shampoo for an 8000×8000 weight matrix on one H100 GPU, with subspaces of relative size B. Our reparametrization makes every subroutine of a step cheaper and enables subspace updates that further reduce the cost. Grey bars are existing methods, blue bars our contributions. Eigendecomposition is far slower than QR, hence recent methods use QR. The existing parametrization applies QR to the nonsymmetric product $\boldsymbol{S}_i\boldsymbol{Q}_i$. Our customized QR accelerates this case by $2.8\times$, and gains $5.4\times$ on the symmetric, nearly diagonal $\boldsymbol{P}_i$ of our parametrization, which all remaining blue bars use. Subspace updates shrink the decomposed matrix and cut the cost further, both for a single subspace and for parallel updates of three disjoint subspaces. Our state update and preconditioning additionally requires fewer matrix multiplications. Speed-ups in the basis block are relative to PyTorch's `qr` on the existing parametrization. Bottom rows compare old to new.

Our main contributions are:

- **Unified full-basis and subspace updates.** Our reparametrization alone accelerates full-basis updates even with standard PyTorch QR. We enable cheaper partial updates while preserving orthogonality of the complete basis without full-basis reorthogonalization (fig. 1, right).
- **Further acceleration through customized QR.** Building on recent QR advances (GPU MODE, 2026) on GPUs, we provide a customized PyTorch implementation of QR that exploits the structure of $\boldsymbol{P}_i$ and supports mixed-precision computation and batched execution. These improvements complement the speedups from reparametrization alone.
- **Broad applicability and flexible subspace selection.** Our framework applies to Shampoo-based methods, including KL-Shampoo, SOAP, and KL-SOAP, with a shared storage convention for preconditioning factors and momentum. Both can be updated through subspaces rather than full-size matrix products. It supports dynamic subspace selection, including Jacobi-style strategies (Van Loan, 1985; Yamamoto et al., 2014), and parallel updates of disjoint subspaces.
- **Improved training performance and runtime.** Even with full-basis updates, our approach improves the training performance of Shampoo- and SOAP-type methods in mixed precision, enabling KL-SOAP to match or exceed KL-Shampoo and closing the gap reported by Lin et al. (2026) (see table 2 and fig. 5). Subspace updates further reduce basis-update time, complementing the gains from reparametrization and customized QR (see fig. 1, right).

2 Background

Notation Shampoo-based methods precondition each weight matrix independently, so we focus on a single matrix $\boldsymbol{\Theta} \in \mathbb{R}^{d_1 \times d_2}$ with gradient $\boldsymbol{G}$. We write $\boldsymbol{\theta} := \mathrm{vec}(\boldsymbol{\Theta})$ and $\boldsymbol{g} := \mathrm{vec}(\boldsymbol{G})$, where $\mathrm{vec}(\cdot)$ flattens its input, and denote the step size by $\gamma > 0$. We omit momentum and weight decay for

simplicity. We use $\mathrm{diag}(\cdot)$ to extract a matrix's diagonal, $\mathrm{Diag}(\cdot)$ to construct a diagonal matrix from a vector, $\odot$ to denote elementwise operations, and $\otimes$ to denote Kronecker products.

2.1 Preconditioning and basis updates

Pure Shampoo methods Pure Shampoo methods, including modern Shampoo variants (Shi et al., 2023; Morwani et al., 2024; Eschenhagen et al., 2025) and KL-Shampoo (Lin et al., 2026), use a Kronecker-factored preconditioning matrix $\boldsymbol{S}_{\text{shampoo}} := \boldsymbol{S}_1 \otimes \boldsymbol{S}_2$, where $\boldsymbol{S}_i \in \mathbb{R}^{d_i \times d_i}$ is symmetric positive definite (SPD). We focus on the inverse-square-root formulation,

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma \boldsymbol{S}_{\text{shampoo}}^{-1/2} \boldsymbol{g} \quad \Longleftrightarrow \quad \boldsymbol{\Theta} \leftarrow \boldsymbol{\Theta} - \gamma \boldsymbol{S}_1^{-1/2} \boldsymbol{G} \boldsymbol{S}_2^{-1/2},$$

where the equivalence follows from the Kronecker-product identity under vectorization.

Explicit basis construction A common approach (Shi et al., 2023; Eschenhagen et al., 2025) computes inverse matrix roots through eigendecomposition of each factor, $\boldsymbol{S}_i = \boldsymbol{Q}_i \mathrm{Diag}(\boldsymbol{\lambda}_i) \boldsymbol{Q}_i^\top$:

$$\boldsymbol{S}_{\text{shampoo}}^{-1/2} = \boldsymbol{Q}\, \mathrm{Diag}\left(\boldsymbol{\lambda}_1^{\odot -1/2} \otimes \boldsymbol{\lambda}_2^{\odot -1/2}\right) \boldsymbol{Q}^\top, \qquad \boldsymbol{Q} := \boldsymbol{Q}_1 \otimes \boldsymbol{Q}_2.$$

Here, $\boldsymbol{Q}_i$ contains the eigenvectors of $\boldsymbol{S}_i$. Preconditioning can be applied through $\boldsymbol{Q}_1$ and $\boldsymbol{Q}_2$, without materializing the full Kronecker matrix $\boldsymbol{Q}$.

SOAP-type methods require an explicit basis SOAP (Vyas et al., 2025) and KL-SOAP (Lin et al., 2026) use the same type of Kronecker-factored basis but estimate Adam's second-moment vector $\tilde{\boldsymbol{d}} \in \mathbb{R}^{d_1 d_2}$ from the rotated gradient $\boldsymbol{Q}^\top \boldsymbol{g}$. Their augmented preconditioning matrix and corresponding update are

$$\boldsymbol{S}_{\text{soap}} := \boldsymbol{Q}\, \mathrm{Diag}(\tilde{\boldsymbol{d}}) \boldsymbol{Q}^\top, \qquad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma \boldsymbol{Q}\, \mathrm{Diag}\left(\tilde{\boldsymbol{d}}^{\odot -1/2}\right) \boldsymbol{Q}^\top \boldsymbol{g}.$$

SOAP-type methods require explicit basis factors $\boldsymbol{Q}_i$ to compute the rotated gradients and estimate $\tilde{\boldsymbol{d}}$.

Basis-free inverse-root computation For pure Shampoo methods, Newton–Schulz-type inverse-root solvers provide an alternative to eigendecomposition (Zhang et al., 2026) while other iterations directly compute preconditioned directions (Amsel et al., 2026). These methods support mixed-precision and batched computation and avoid constructing $\boldsymbol{Q}_i$ but require sequential matrix products whenever invoked, often at every optimization step. They do not directly replace SOAP's basis updates because they do not provide the explicit basis needed for its second-moment estimation $\tilde{\boldsymbol{d}}$.

Basis staleness and update frequency To amortize expensive decomposition costs, explicit-basis methods typically refresh $\boldsymbol{Q}_i$ every T gradient steps and reuse it between refreshes. Increasing T reduces overhead but can degrade training performance as the basis becomes stale (Shi et al., 2023; Vyas et al., 2025; Eschenhagen et al., 2025). This trade-off depends on how rapidly the factors evolve and motivates cheaper basis updates rather than simply longer refresh intervals.

QR-based basis updates A cheaper alternative to eigendecomposition is a *single* warm-started orthogonal-iteration step (Vyas et al., 2025; Eschenhagen et al., 2025; Lin et al., 2026):

$$\boldsymbol{Q}_i^{(\text{new})} = \mathrm{qr}\left(\boldsymbol{S}_i \boldsymbol{Q}_i^{(\text{old})}\right), \qquad i \in \{1, 2\}, \tag{1}$$

where $\mathrm{qr}(\cdot)$ returns the orthogonal QR factor. This update refines the previous basis estimate and can permit more frequent refreshes within a given computational budget. QR updates the basis rather than its eigenvalues; diagonal scaling estimates, such as $\boldsymbol{\lambda}_i$ or $\tilde{\boldsymbol{d}}$, are maintained separately and can be updated at every gradient step (Vyas et al., 2025; Eschenhagen et al., 2025; Lin et al., 2026). This separates inexpensive scale estimation from costly basis computation.

2.2 Limitations and opportunities in orthogonal basis updates

Full-basis computation remains expensive Standard QR and eigendecomposition require cubic complexity in the factor dimension, with sequential stages that limit GPU throughput relative to matrix multiplication (Golub and Van Loan, 2013). Although QR reduces the cost of basis updates, it remains the main computational bottleneck in the workloads shown in fig. 1 (right). This motivates both faster full-basis updates and cheaper updates to selected basis vectors.

Factor representation and QR specialization offer complementary gains Conventional basis updates pass $\boldsymbol{S}_i\boldsymbol{Q}_i$ (see eq. (1)) to a general-purpose QR routine. Although $\boldsymbol{S}_i$ is symmetric positive definite, this product is generally nonsymmetric and does not directly expose that structure. Changing the factor representation can reduce basis-update time even with standard PyTorch QR routine (see fig. 1). Customized QR offers further opportunities by exploiting input structure. Recent QR optimization efforts on GPUs (GPU MODE, 2026) provide a foundation for customized mixed-precision implementations. Building on these advances, we seek to exploit mixed precision and input structure jointly in Shampoo and SOAP.

Partial updates must preserve the orthogonality of the full basis $\boldsymbol{Q}_i$ Applying QR only to selected columns of $\boldsymbol{S}_i\boldsymbol{Q}_i$ in eq. (1) orthogonalizes the updated vectors among themselves but generally not against the unchanged columns of the basis $\boldsymbol{Q}_i$. Efficient partial updates must therefore preserve orthogonality across the full basis without expensive full-basis re-orthogonalization.

Subspace selection should be dynamic As each factor $\boldsymbol{S}_i$ evolves, the selected subset of basis vectors should be allowed to change across updates rather than remain tied to a fixed partition. Efficient dynamic selection requires a factor representation that keeps the relevant information readily accessible without repeated full-matrix computations and remains consistent with the evolving basis.

Together, these considerations motivate our reparametrization: accelerating basis updates with standard routines, enabling further gains from customized mixed-precision QR, and unifying full-basis and dynamically selected subspace updates.

3 Reparametrization for Shampoo and SOAP

We address the limitations of conventional basis updates through two complementary components: a preconditioner reparametrization that exposes useful input structure and a customized QR implementation that provides further acceleration. Specifically, we maintain $\boldsymbol{P}_i := \boldsymbol{Q}_i^\top \boldsymbol{S}_i \boldsymbol{Q}_i$ in the current orthogonal basis and update it directly without materializing $\boldsymbol{S}_i$. This reparametrization reduces basis-update time even with standard PyTorch QR and eigendecomposition routines (see fig. 1). Building on recent QR advances (GPU MODE, 2026), our customized implementation further exploits the structured input while supporting mixed-precision computation and batched execution.

The same representation also enables efficient partial updates without sacrificing full-basis orthogonality. Maintaining $\boldsymbol{P}_i$ directly makes the information needed for dynamic subspace selection accessible: selected subspaces can change across updates, and disjoint subspaces can be updated simultaneously.

3.1 Full-basis updates: Exposing structured QR inputs

We first show how the reparametrization exposes symmetric positive-definite QR inputs while preserving the conventional full-basis QR update. Throughout this subsection, we use the QR convention in which the triangular factor $\boldsymbol{R}$ has positive diagonal entries, making the decomposition unique for nonsingular inputs. See section A for applying this convention to an existing QR implementation; our customized implementation uses the same convention.

From a general to structured QR input Existing QR-based Shampoo methods (Vyas et al., 2025; Lin et al., 2026) store $\boldsymbol{S}_i$ and refine its approximate eigenbasis through

$$\boldsymbol{Q}_i^{(\text{new})}\boldsymbol{R}_i^{(\text{new})} = \boldsymbol{S}_i\boldsymbol{Q}_i^{(\text{old})}, \qquad i \in \{1, 2\}, \tag{2}$$

as illustrated in step 4 of fig. 2. Here, $\boldsymbol{S}_i$ is held fixed during the basis refresh. Instead of the generally nonsymmetric $\boldsymbol{S}_i\boldsymbol{Q}_i^{(\text{old})}$, our reparametrization instead exposes

$$\boldsymbol{P}_i^{(\text{old})} := \boldsymbol{Q}_i^{(\text{old})\top}\boldsymbol{S}_i\boldsymbol{Q}_i^{(\text{old})},$$

which is SPD and nearly diagonal when $\boldsymbol{Q}_i^{(\text{old})}$ closely approximates an eigenbasis (and in fact diagonal if the basis is exact). QR implementations can exploit this structure, and the near diagonal $\boldsymbol{P}_i$ also proves better suited to the BFP16 mixed-precision computation in practice; see fig. 5.

KL-Shampoo (Original)	**KL-Shampoo (Reparameterized)**
Store raw momentum $\boldsymbol{M}$	Store rotated momentum $\tilde{\boldsymbol{M}}$, as in KL-SOAP
$\{\boldsymbol{\lambda}_i, \boldsymbol{Q}_i, \boldsymbol{S}_i \mid i = 1, 2\}$	$\{\boldsymbol{\lambda}_i, \boldsymbol{Q}_i, \boldsymbol{P}_i := \boldsymbol{Q}_i^\top \boldsymbol{S}_i \boldsymbol{Q}_i \mid i = 1, 2\}$
1. Compute the gradient	Every iteration
$\boldsymbol{g} := \nabla \ell(\boldsymbol{\theta}), \quad \boldsymbol{G} := \mathrm{Mat}(\boldsymbol{g}) \in \mathbb{R}^{d_1 \times d_2}$	
2. Update preconditioning factors	EMA; every iteration; full basis
$\boldsymbol{G}'_1 := \boldsymbol{G}\boldsymbol{Q}_2 \,\mathrm{Diag}(\boldsymbol{\lambda}_2^{-1/2})/\sqrt{d_2}$ #1 mm $\boldsymbol{G}'_2 := \boldsymbol{G}^\top \boldsymbol{Q}_1 \,\mathrm{Diag}(\boldsymbol{\lambda}_1^{-1/2})/\sqrt{d_1}$ #1 mm $\Delta_i := \boldsymbol{G}'_i \boldsymbol{G}'^\top_i$ #2 mms $\boldsymbol{S}_i \leftarrow (1-\beta_2)\boldsymbol{S}_i + \beta_2 \Delta_i$	$\tilde{\boldsymbol{G}}' := \boldsymbol{Q}_1^\top \boldsymbol{G} \boldsymbol{Q}_2$ #2 mms $\tilde{\boldsymbol{G}}'_1 := \tilde{\boldsymbol{G}}' \,\mathrm{Diag}(\boldsymbol{\lambda}_2^{-1/2})/\sqrt{d_2}$ $\tilde{\boldsymbol{G}}'_2 := \tilde{\boldsymbol{G}}'^\top \,\mathrm{Diag}(\boldsymbol{\lambda}_1^{-1/2})/\sqrt{d_1}$ $\tilde{\Delta}_i := \tilde{\boldsymbol{G}}'_i \tilde{\boldsymbol{G}}'^\top_i$ #2 mms $\boldsymbol{P}_i \leftarrow (1-\beta_2)\boldsymbol{P}_i + \beta_2 \tilde{\Delta}_i$
3. Track eigenvalues	EMA; every iteration; full basis
$\tilde{\boldsymbol{G}}'_i := \boldsymbol{Q}_i^\top \boldsymbol{G}'_i$ #2 mms $\mathrm{diag}(\tilde{\Delta}_i) := (\tilde{\boldsymbol{G}}'_i \odot \tilde{\boldsymbol{G}}'_i)\mathbf{1}$	Reuse $\tilde{\Delta}_i$ from step 2. No additional matrix products.
$\boldsymbol{\lambda}_i \leftarrow (1-\beta_2)\boldsymbol{\lambda}_i + \beta_2 \,\mathrm{diag}(\tilde{\Delta}_i)$	
4. Update the approximate eigenbasis	Full-basis QR; every $T \geq 1$ iterations
$\boldsymbol{Z}_i \leftarrow \boldsymbol{S}_i \boldsymbol{Q}_i$ #$2/T$ mms $\boldsymbol{Q}_i \leftarrow \mathrm{qr}(\boldsymbol{Z}_i)$ #$O(d_i^3/T)$	$\boldsymbol{O}_i \leftarrow \mathrm{qr}(\boldsymbol{P}_i)$ #$O(d_i^3/T)$ $\boldsymbol{Q}_i \leftarrow \boldsymbol{Q}_i \boldsymbol{O}_i$ #$2/T$ mms $\boldsymbol{P}_i \leftarrow \boldsymbol{O}_i^\top \boldsymbol{P}_i \boldsymbol{O}_i$ #$4/T$ mms
5. Update rotated momentum for preconditioning	Every iteration
$\boldsymbol{Q} := \boldsymbol{Q}_1 \otimes \boldsymbol{Q}_2, \quad \boldsymbol{m} := \mathrm{vec}(\boldsymbol{M}), \quad \tilde{\boldsymbol{m}} := \mathrm{vec}(\tilde{\boldsymbol{M}})$	
Store raw momentum $\boldsymbol{M}$. $\boldsymbol{M} \leftarrow (1-\beta_1)\boldsymbol{M} + \beta_1 \boldsymbol{G}$ $\tilde{\boldsymbol{m}} \leftarrow \boldsymbol{Q}^\top \boldsymbol{m}$ #2 mms $\equiv \mathrm{vec}(\boldsymbol{Q}_1^\top \boldsymbol{M} \boldsymbol{Q}_2)$	Reuse pre-refresh $\tilde{\boldsymbol{G}}'$ from step 2. $\tilde{\boldsymbol{M}} \leftarrow (1-\beta_1)\tilde{\boldsymbol{M}} + \beta_1 \tilde{\boldsymbol{G}}'$ If the basis is updated in step 4: $\tilde{\boldsymbol{M}} \leftarrow \boldsymbol{O}_1^\top \tilde{\boldsymbol{M}} \boldsymbol{O}_2$ #$2/T$ mms
6. Precondition and update parameters	Learning rate γ
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma\big(\boldsymbol{Q}\,\mathrm{Diag}(\boldsymbol{\lambda}_1 \otimes \boldsymbol{\lambda}_2)^{-1/2}\boldsymbol{Q}^\top\big)\boldsymbol{m} \equiv \boldsymbol{\theta} - \gamma\big(\boldsymbol{Q}\,\mathrm{Diag}(\boldsymbol{\lambda}_1 \otimes \boldsymbol{\lambda}_2)^{-1/2}\big)\tilde{\boldsymbol{m}}.$	

Figure 2: **KL-Shampoo and its reparameterized variant.** Replacing $\boldsymbol{S}_i$ with $\boldsymbol{P}_i := \boldsymbol{Q}_i^\top \boldsymbol{S}_i \boldsymbol{Q}_i$ and storing rotated momentum $\tilde{\boldsymbol{M}} := \boldsymbol{Q}_1^\top \boldsymbol{M} \boldsymbol{Q}_2$ establishes a shared factor-and-momentum storage convention across reparameterized KL-Shampoo, SOAP, and KL-SOAP (figs. 8 and 9). The two KL-Shampoo variants produce equivalent iterates (see section 3.1). **Color code.** Red marks reparameterized factors; blue marks $\tilde{\boldsymbol{G}}'_i = \boldsymbol{Q}_i^\top \boldsymbol{G}'_i$. The original variant forms $\tilde{\boldsymbol{G}}'_i$ in step 3; the reparameterized variant reuses them from step 2 to update $\boldsymbol{P}_i$, with $\tilde{\Delta}_i = \boldsymbol{Q}_i^\top \Delta_i \boldsymbol{Q}_i$. **Cost.** Gray annotations count matrix products (mm), summed over $i = 1, 2$ and amortized over T where indicated; QR costs are per factor. Steps 2–3 save 2 mm per iteration, step 4 adds $(4/T)$ mm, and step 5 saves $(2 - 2/T)$ mm: a net saving of $(4 - 6/T)$ mm, positive for integer $T \geq 2$.

An equivalent full-basis QR update Let $\boldsymbol{O}_i \bar{\boldsymbol{R}}_i = \boldsymbol{P}_i^{(\text{old})}$ be the QR decomposition of the reparametrized factor. Substituting the conventional update in eq. (2) gives

$$\boldsymbol{P}_i^{(\text{old})} = \boldsymbol{Q}_i^{(\text{old})\top}\big(\boldsymbol{S}_i \boldsymbol{Q}_i^{(\text{old})}\big) = \big(\boldsymbol{Q}_i^{(\text{old})\top} \boldsymbol{Q}_i^{(\text{new})}\big)\boldsymbol{R}_i^{(\text{new})}. \tag{3}$$

The right-hand side is another QR decomposition of $\boldsymbol{P}_i^{(\text{old})}$, $\big(\boldsymbol{Q}_i^{(\text{old})\top} \boldsymbol{Q}_i^{(\text{new})}\big)$ is orthogonal and $\boldsymbol{R}_i^{(\text{new})}$ is upper triangular with positive diagonal entries. Uniqueness therefore implies

$$\boldsymbol{O}_i = \boldsymbol{Q}_i^{(\text{old})\top} \boldsymbol{Q}_i^{(\text{new})}, \qquad \bar{\boldsymbol{R}}_i = \boldsymbol{R}_i^{(\text{new})}.$$

Thus, we recover the same updated basis by computing QR on $\boldsymbol{P}_i^{(\text{old})}$ and applying the rotation:

$$\boldsymbol{Q}_i^{(\text{new})} = \boldsymbol{Q}_i^{(\text{old})} \boldsymbol{O}_i. \tag{4}$$

KL-Shampoo (Original)	**KL-Shampoo (Reparameterized)**
Raw factors $\boldsymbol{S}_i$; raw momentum $\boldsymbol{M}$	Rotated factors $\boldsymbol{P}_i$; rotated momentum $\tilde{\boldsymbol{M}}$
4. Update the approximate eigenbasis	Subspace QR; every $T \geq 1$ iterations
4a. Select the subspace dynamically	
Full-matrix products at each refresh.	**No matrix products.**
$\boldsymbol{P}_i \leftarrow \boldsymbol{Q}_i^\top \boldsymbol{S}_i \boldsymbol{Q}_i$ $\quad \#\, 4/T$ mms	Reuse $\boldsymbol{P}_i$ from step 2.
$\mathbb{I}_i := \text{select}(\boldsymbol{P}_i), \quad \lvert\mathbb{I}_i\rvert = d_{\text{sub},i} = Bd_i, \quad 0 < B \leq 1, \quad i \in \{1, 2\}$	
Reselect $\mathbb{I}_i$ at each basis refresh.	
4b. Compute QR within the selected subspace	
$\boldsymbol{O}_i \leftarrow \text{qr}(\boldsymbol{P}_i[\mathbb{I}_i, \mathbb{I}_i])$ $\quad \#\, O(d_{\text{sub},i}^3/T)$	
Full-basis QR is recovered when $B = 1$.	
4c. Update selected basis vectors and rotate the state	
$\boldsymbol{Q}_i[:, \mathbb{I}_i] \leftarrow \boldsymbol{Q}_i[:, \mathbb{I}_i]\boldsymbol{O}_i$ $\quad \#\, 2/T$ smms	
No state rotation is needed for $\boldsymbol{S}_i$.	Rotate $\boldsymbol{P}_i$ with **subspace products**:
	$\boldsymbol{P}_i[:, \mathbb{I}_i] \leftarrow \boldsymbol{P}_i[:, \mathbb{I}_i]\boldsymbol{O}_i$ $\quad \#\, 2/T$ smms
	$\boldsymbol{P}_i[\mathbb{I}_i, :] \leftarrow \boldsymbol{O}_i^\top \boldsymbol{P}_i[\mathbb{I}_i, :]$ $\quad \#\, 2/T$ smms
5. Update rotated momentum for preconditioning	Every iteration
$\boldsymbol{Q} := \boldsymbol{Q}_1 \otimes \boldsymbol{Q}_2, \quad \boldsymbol{m} := \text{vec}(\boldsymbol{M}), \quad \tilde{\boldsymbol{m}} := \text{vec}(\tilde{\boldsymbol{M}})$	
Full-matrix products at every iteration.	**Subspace products only at basis refreshes.**
$\boldsymbol{M} \leftarrow (1-\beta_1)\boldsymbol{M} + \beta_1 \boldsymbol{G}$	$\tilde{\boldsymbol{M}} \leftarrow (1-\beta_1)\tilde{\boldsymbol{M}} + \beta_1 \tilde{\boldsymbol{G}}'$
$\tilde{\boldsymbol{m}} \leftarrow \boldsymbol{Q}^\top \boldsymbol{m}$ $\quad \#\, 2$ mms	If the basis is updated in step 4:
$\equiv \text{vec}(\boldsymbol{Q}_1^\top \boldsymbol{M} \boldsymbol{Q}_2)$	$\tilde{\boldsymbol{M}}[:, \mathbb{I}_2] \leftarrow \tilde{\boldsymbol{M}}[:, \mathbb{I}_2]\boldsymbol{O}_2$ $\quad \#\, 1/T$ smms
	$\tilde{\boldsymbol{M}}[\mathbb{I}_1, :] \leftarrow \boldsymbol{O}_1^\top \tilde{\boldsymbol{M}}[\mathbb{I}_1, :]$ $\quad \#\, 1/T$ smms
Reuse pre-refresh $\tilde{\boldsymbol{G}}'$ from step 2 in the EMA; apply column/row rotations in steps 4c and 5.	

Figure 3: **Subspace updates for steps 4 and 5 of fig. 2.** Steps 1–3 and 6 are unchanged: factors and eigenvalues are updated in the full basis every iteration. Reselect $\mathbb{I}_i$ at each basis refresh; $B = 1$ recovers the reparametrized full-basis update. **Full-matrix versus subspace products.** The original parameterization uses full-matrix products (mm) to reconstruct $\boldsymbol{P}_i$ in step 4a ($(4/T)$ mm) and project momentum in step 5 (2 mm per iteration). The reparameterized one reuses $\boldsymbol{P}_i$ with *no matrix products* in step 4a and uses only subspace products (smm) for basis and state rotations in step 4c ($(6/T)$ smm) and momentum rotations in step 5 ($(2/T)$ smm), retaining unified KL-Shampoo/KL-SOAP storage. **Cost.** Matrix-product counts are summed over $i = 1, 2$ and amortized over T; QR costs are per factor. For $d_{\text{sub},i} = Bd_i$, leading-order costs scale by B^2 for subspace products and B^3 for QR relative to full-basis updates.

The reparametrization therefore changes the input to QR implementations from a general matrix $\boldsymbol{S}_i\boldsymbol{Q}_i$ to a near-diagonal SPD matrix $\boldsymbol{P}_i$, which specialized QR routines can exploit.

Maintaining the structured factor directly After a basis refresh, the stored factor must be expressed in the new coordinates:

$$\boldsymbol{P}_i^{(\text{new})} := \boldsymbol{Q}_i^{(\text{new})\top} \boldsymbol{S}_i \boldsymbol{Q}_i^{(\text{new})} = \boldsymbol{O}_i^\top \big(\boldsymbol{Q}_i^{(\text{old})\top} \boldsymbol{S}_i \boldsymbol{Q}_i^{(\text{old})}\big) \boldsymbol{O}_i = \boldsymbol{O}_i^\top \boldsymbol{P}_i^{(\text{old})} \boldsymbol{O}_i. \tag{5}$$

This updates $\boldsymbol{P}_i$ consistently with the updated basis without reconstructing $\boldsymbol{S}_i$.

Typically during factor estimation, $\boldsymbol{Q}_i$ is held fixed while $\boldsymbol{S}_i$ is updated by an exponential moving average (EMA), as in step 2 of fig. 2. The corresponding update of $\boldsymbol{P}_i$ is

$$\boldsymbol{P}_i^{(\text{new})} = \boldsymbol{Q}_i^\top \big((1-\beta_2)\boldsymbol{S}_i^{(\text{old})} + \beta_2 \Delta_i\big) \boldsymbol{Q}_i = (1-\beta_2)\boldsymbol{P}_i^{(\text{old})} + \beta_2 \tilde{\Delta}_i, \qquad \tilde{\Delta}_i := \boldsymbol{Q}_i^\top \Delta_i \boldsymbol{Q}_i.$$

The rotated increment $\tilde{\Delta}_i$ can be computed directly from rotated gradient factors (step 2 of fig. 2). Together, these maintain $\boldsymbol{P}_i = \boldsymbol{Q}_i^\top \boldsymbol{S}_i \boldsymbol{Q}_i$ via factor and basis updates, without materializing $\boldsymbol{S}_i$.

Applicability to Shampoo and SOAP with shared storage The QR equivalence is independent of how Δ_i is estimated, so the same construction applies to KL-Shampoo, SOAP, and KL-SOAP (see figs. 2, 8 and 9). Although their factor-estimation rules differ, each can maintain $\boldsymbol{P}_i$ through the corresponding rotated increment $\tilde{\Delta}_i$. Each method's estimation and preconditioning rules remain compatible with the reparametrization. Storing momentum as $\tilde{\boldsymbol{M}} := \boldsymbol{Q}_1^\top \boldsymbol{M} \boldsymbol{Q}_2$ in the same basis coordinates further establishes a shared factor-and-momentum storage convention across the three methods and enables common subspace updates for factors and momentum. At a basis refresh, the rotations update momentum via $\tilde{\boldsymbol{M}} \leftarrow \boldsymbol{O}_1^\top \tilde{\boldsymbol{M}} \boldsymbol{O}_2$. The formulation in eqs. (4) and (5) provides a common foundation for the customized QR implementation and the subspace updates.

3.2 Customized QR: Acceleration through structure and mixed precision

Building on recent QR advances targeting NVIDIA B200 GPUs (GPU MODE, 2026), we adapt techniques from the top three leaderboard submissions to Shampoo-based methods, providing further acceleration beyond reparametrization alone. Our PyTorch implementation supports mixed-precision computation and batched execution while extending hardware support to NVIDIA A100 and H100 GPUs. Unlike `torch.linalg.qr`, it enforces the positive-diagonal QR convention used above for nonsingular inputs, uniquely specifying the orthogonal factor $\boldsymbol{O}_i$ for $\boldsymbol{P}_i$. Because Shampoo methods require only the orthogonal factor, we avoid computing the triangular factor to reduce overhead.

Although our implementation supports general inputs, it can exploit the SPD structure and near-diagonality of $\boldsymbol{P}_i$ to enable faster, structure-aware computation. For suitable inputs, specialized orthogonalization replaces the more expensive computations used in the general case. Structural information also helps identify and skip unnecessary computation. See section B for details.

3.3 Subspace updates: Reducing computation while preserving orthogonality

To reduce basis-update cost further, our reparametrization supports QR on selected blocks of $\boldsymbol{P}_i$, rotating only the corresponding basis vectors in $\boldsymbol{Q}_i$ while preserving full-basis orthogonality. These local rotations resemble block Jacobi-type methods (Van Loan, 1985; Yamamoto et al., 2014) that use eigendecomposition within selected blocks. We focus on subspace QR because it is computationally cheaper (see the right of fig. 1) while empirically achieving training performance similar to that of subspace eigendecomposition for Shampoo-based methods (see table 4).

We describe a single subspace update. For illustration, group selected indices $\mathbb{I}_i$ and partition $\boldsymbol{P}_i$ as

$$\boldsymbol{P}_i = \begin{bmatrix} \boldsymbol{P}_i^{(XX)} & \boldsymbol{P}_i^{(XY)} \\ \boldsymbol{P}_i^{(YX)} & \boldsymbol{P}_i^{(YY)} \end{bmatrix} \quad \text{with } \boldsymbol{P}_i[:, \mathbb{I}_i] = \begin{bmatrix} \boldsymbol{P}_i^{(XX)} \\ \boldsymbol{P}_i^{(YX)} \end{bmatrix},$$

where the block $\boldsymbol{P}_i^{(XX)} = \boldsymbol{P}_i[\mathbb{I}_i, \mathbb{I}_i]$, highlighted in red, is selected for the update.

Conceptually, we perform QR decomposition only on $\boldsymbol{P}_i^{(XX)}$ to obtain a local orthogonal factor $\boldsymbol{O}_i^{(XX)}$, and then define $\boldsymbol{O}_i = \boldsymbol{O}_i^{(XX)} \oplus \boldsymbol{I}^{(YY)}$, where $\oplus$ denotes the direct sum and $\boldsymbol{I}^{(YY)}$ is an identity matrix. We apply this block-diagonal rotation through eqs. (4) and (5), without explicitly forming the rotation matrix. The basis update $\boldsymbol{Q}_i^{(\text{new})} = \boldsymbol{Q}_i^{(\text{old})} \boldsymbol{O}_i$ changes only the selected columns:

Partially Updating Orthogonal Basis Vectors via Subspace QR

$$\begin{bmatrix} (\boldsymbol{Q}_i^{(\text{old})})^{(XX)} & \underbrace{(\boldsymbol{Q}_i^{(\text{old})})^{(XY)}}_{\text{old basis}} \\ (\boldsymbol{Q}_i^{(\text{old})})^{(YX)} & (\boldsymbol{Q}_i^{(\text{old})})^{(YY)} \end{bmatrix} \begin{bmatrix} \boldsymbol{O}_i^{(XX)} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{I}^{(YY)} \end{bmatrix} = \begin{bmatrix} \underbrace{(\boldsymbol{Q}_i^{(\text{old})})^{(XX)} \boldsymbol{O}_i^{(XX)}}_{\text{new subspace basis}} & \underbrace{(\boldsymbol{Q}_i^{(\text{old})})^{(XY)}}_{\text{unchanged basis}} \\ (\boldsymbol{Q}_i^{(\text{old})})^{(YX)} \boldsymbol{O}_i^{(XX)} & (\boldsymbol{Q}_i^{(\text{old})})^{(YY)} \end{bmatrix}. \tag{6}$$

Similarly, we express $\boldsymbol{P}_i$ in the updated basis using eq. (5):

$$\boldsymbol{P}_i^{(\text{new})} = \boldsymbol{O}_i^\top \boldsymbol{P}_i^{(\text{old})} \boldsymbol{O}_i = \begin{bmatrix} (\boldsymbol{O}_i^{(XX)})^\top \boldsymbol{P}_i^{(XX)} \boldsymbol{O}_i^{(XX)} & (\boldsymbol{O}_i^{(XX)})^\top \boldsymbol{P}_i^{(XY)} \\ \boldsymbol{P}_i^{(YX)} \boldsymbol{O}_i^{(XX)} & \underbrace{\boldsymbol{P}_i^{(YY)}}_{\text{unchanged block}} \end{bmatrix}. \tag{7}$$

$\boldsymbol{Q}_i^{(\text{new})}$ remains orthogonal because both $\boldsymbol{Q}_i^{(\text{old})}$ and $\boldsymbol{O}_i$ are orthogonal. Thus, the updated vectors remain orthogonal to the unchanged vectors *without* full-basis re-orthogonalization. Moreover, only

the selected columns of $\boldsymbol{Q}_i$ and the corresponding rows and columns of $\boldsymbol{P}_i$ need updating. The same local rotations update $\tilde{\boldsymbol{M}}$ through subspace matrix products (step 5 of fig. 3). Selecting all indices recovers the full-basis scheme. The update in eqs. (6) and (7) recovers the block-Jacobi-type update (Yamamoto et al., 2014) when using eigendecomposition on $\boldsymbol{P}_i^{(XX)}$ rather than QR decomposition.

A single subspace QR step for an evolving factor Our goal is to improve preconditioned directions rather than fully diagonalize each evolving factor. As in full-basis QR updates, we perform a *single* subspace QR step per selected block at each basis update, rather than iterating on a fixed factor as in standard Jacobi methods. This reduces decomposition cost and allows more frequent basis updates as $\boldsymbol{S}_i$ evolves. We compare these single-step and multi-step approaches and find that a single step is often sufficient for Shampoo methods when the decomposition interval is small (see tables 5 and 6).

Storing $\boldsymbol{P}_i$ facilitates dynamic subspace selection Like Jacobi-type methods, we select one block of $\boldsymbol{P}_i$ for each local update. When $\boldsymbol{Q}_i$ is an exact eigenbasis of $\boldsymbol{S}_i$, $\boldsymbol{P}_i = \boldsymbol{Q}_i^\top \boldsymbol{S}_i \boldsymbol{Q}_i$ is diagonal. Motivated by this observation, Jacobi-type greedy methods commonly seek to reduce the Frobenius norm of the off-diagonal part of $\boldsymbol{P}_i$ when selecting a block. Greedy block-selection strategies offer practical alternatives to searching for an optimal block. Because we store $\boldsymbol{P}_i$ explicitly, the entries needed to evaluate such criteria are directly available, without reconstructing them from $\boldsymbol{S}_i$. Thus, our approach supports existing and new greedy strategies, with blocks free to change across updates rather than remain tied to a fixed partition. See section D for a GPU-friendly selection method.

Parallel updates of disjoint subspaces For disjoint index sets, we compute local QR factors independently from the same $\boldsymbol{P}_i$ and apply the basis rotations in parallel. To the authors' knowledge, no direct analog to single block greedy selection exists in the parallel setting, so we use randomly partitioned blocks as a baseline when selection more than two parallel subspaces. For a point of comparison we tested an LLM generated parallel block selection heuristic, finding similar results to random selection. See figures fig. 4 and fig. 6 for results and section C for details.

4 Experiments

We conduct experiments to evaluate mixed-precision training performance, subspace updates, and runtime. In Experiment 1, we consider KL-Shampoo, SOAP, and KL-SOAP. The remaining experiments focus on KL-Shampoo as a representative Shampoo method due to limited computational resources and tranferability to other methods (section 3.3). See section F for additional experiments.

Experimental setup We consider the following baseline training methods: SOAP (Vyas et al., 2025), KL-Shampoo/KL-SOAP (Lin et al., 2026), and Muon (Liu et al., 2025). We train language models—nanoGPT (Jordan, 2024) (123M) and Llama 3 (Raschka, 2024) (119M, 313M, and 598M)—on the FineWeb dataset. We use the official implementations of SOAP (Vyas et al., 2025) and KL-Shampoo/KL-SOAP (Lin et al., 2026). For Muon, we use the Polar Express implementation (Amsel et al., 2026). For each baseline, we tune all available hyperparameters, including the learning rate, weight decay, damping, β_1, and β_2, using random search over 120 runs. For Shampoo-based methods, we set the decomposition interval to $T = 10$ for full-basis updates, as suggested by Vyas et al. (2025) and Lin et al. (2026). Both full-basis and subspace reparametrized variants reuse the best optimizer hyperparameters found for their original counterparts, without additional tuning. By default, we use BFP16 mixed-precision training. Decomposition precision depends on the implementation: standard PyTorch QR and eigendecomposition use FP32, while our customized QR uses mixed precision internally. See section E for additional details.

Experiment 1: Our reparametrization preserves training performance under mixed-precision computation. Table 2 compares FP32 and BFP16 mixed-precision computation using full-basis updates. Our reparametrization preserves training performance in these experiments, whereas the original parametrization exhibits degradation (see the old − new gap over training in fig. 5). It also closes the performance gap between KL-Shampoo and KL-SOAP reported by Lin et al. (2026), enabling KL-SOAP to match or exceed KL-Shampoo under mixed-precision computation. These improvements do not require subspace updates or additional hyperparameter tuning.

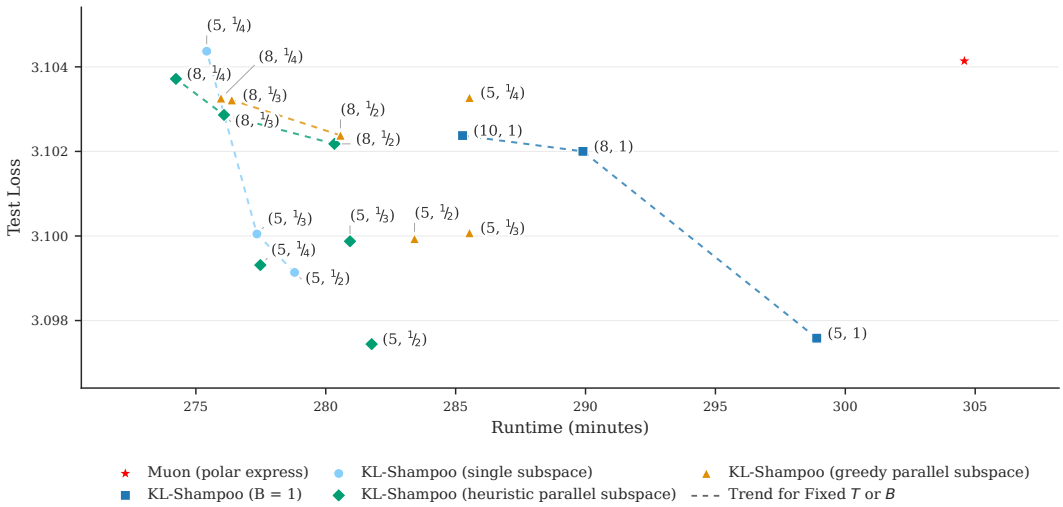

Figure 4: Performance of reparametrized KL-Shampoo varying T, B, and subspace selection strategies (with the BFP16 mixed-precision training) on Llama3 (598M) trained on FineWeb 4.1B, evaluated with the hyperparameters tuned for full-basis KL-Shampoo over 120 runs. We note that in all but one configuration, KL-Shampoo outperforms tuned Muon in both data efficiency and runtime. In most cases, a natural tradeoff between time and loss can be seen as T and B are varied. While this trend is unclear in some parallel subspace configurations, we suspect this is due to noise introduced by no optimal selection heuristic being known.

Experiment 2: Comparing dynamic subspace selection strategies We compare uniform random selection with the greedy strategy described in section 3.3. We also consider a K-step inner loop at each basis refresh (table 3). Greedy selection is generally more effective than random selection. Its advantage is particularly obvious on nanoGPT.

Experiment 3: Subspace QR matches eigendecomposition in training performance at lower cost As discussed in section 3.3, our subspace construction recovers block Jacobi-type updates when local QR is replaced by eigendecomposition. We therefore compare both decompositions within the same subspace framework. Table 4 shows comparable training performance, while the timing results in fig. 1 (right) show that QR is substantially faster. This extends the motivation for QR-based basis updates (Vyas et al., 2025) to dynamically selected subspaces.

Experiment 4: Reparametrization, customized QR, and subspace updates provide complementary runtime gains The timing breakdown in fig. 1 separates three sources of acceleration. We also evaluate the effect on training a larger model. The left panel of fig. 1 shows that subspace methods reduce wall-clock runtime at the cost of slightly worse training performance per step. These results illustrate the trade-off between per-step progress and computational cost. We show a more comprehensive comparison of per-step progress and computational cost, for different preconditioner update frequencies, subspace sizes, and selection strategies in fig. 4.

5 Conclusion

We introduce a reparametrization of the preconditioning factors in Shampoo-based methods. The reparametrization accelerates basis updates even with the standard PyTorch QR routine. A customized, structure-aware QR implementation provides further acceleration through mixed-precision computation and batched execution. Reparameterization further enables efficient updates of dynamically selected subspaces while preserving full-basis orthogonality and supporting factor and momentum updates through subspace matrix products. The approach unifies full-basis and subspace updates and applies to KL-Shampoo, SOAP, and KL-SOAP. Our empirical results demonstrate significantly reduced computational overhead and improved mixed-precision training performance for Shampoo-based methods, and offers a new approach to accelerating these algorithms.

AI USE DISCLOSURE

We used generative AI tools to improve the manuscript's clarity, grammar, and readability. We also used these tools to improve our code writing and algorithm implementations. However, they were not used to develop the theoretical ideas, formulate mathematical claims, design experiments, interpret results, or produce the scientific conclusions of this work. All AI-assisted edits were reviewed and revised by the authors, who take full responsibility for the final content of the paper.

REFERENCES

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

Martin Bečka, Gabriel Okša, and Marian Vajteršic. Dynamic ordering for a parallel block-Jacobi SVD algorithm. *Parallel Computing*, 2002.

George E Dahl, Frank Schneider, Zachary Nado, Naman Agarwal, Chandramouli Shama Sastry, Philipp Hennig, Sourabh Medapati, Runa Eschenhagen, Priya Kasimbeg, Daniel Suo, et al. Benchmarking neural network training algorithms, 2023.

Runa Eschenhagen, Aaron Defazio, Tsung-Hsien Lee, Richard E Turner, and Hao-Jun Michael Shi. Purifying Shampoo: Investigating Shampoo's heuristics by decomposing its preconditioner. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Runa Eschenhagen, Anna Cai, Tsung-Hsien Lee, and Hao-Jun Michael Shi. Clarifying Shampoo: Adapting spectral descent to stochasticity and the parameter trajectory. *arXiv preprint arXiv:2602.09314*, 2026.

George Elmer Forsythe and Peter Henrici. The cyclic Jacobi method for computing the principal values of a complex matrix. *Transactions of the American Mathematical Society*, 1960.

Gene H Golub and Charles F Van Loan. *Matrix computations*. JHU press, 2013.

GPU MODE. qr_v2: Batched square compact-householder QR factorization. https://www.gpumode.com/leaderboard/774?tab=rankings, 2026. Leaderboard 774. Accessed September 21, 2026.

Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. In *International Conference on Machine Learning (ICML)*, 2018.

Keller Jordan. NanoGPT (124M) in 3 minutes. https://github.com/KellerJordan/modded-nanogpt, 2024. Accessed: 2025-06.

Priya Kasimbeg, Frank Schneider, Runa Eschenhagen, Juhan Bae, Chandramouli Shama Sastry, Mark Saroufim, Boyuan Fend, Less Wright, Edward Z Yang, Zachary Nado, et al. Accelerating neural network training: An analysis of the AlgoPerf competition. In *The Thirteenth International Conference on Learning Representations*, 2025.

Wu Lin, Scott C. Lowe, Felix Dangel, Runa Eschenhagen, Zikun Xu, and Roger B. Grosse. Understanding and improving Shampoo and SOAP via Kullback-Leibler minimization. In *International Conference on Learning Representations (ICLR)*, 2026.

Jingyuan Liu, Jianlin Su, Xingcheng Yao, Zhejun Jiang, Guokun Lai, Yulun Du, Yidao Qin, Weixin Xu, Enzhe Lu, Junjie Yan, Yanru Chen, Huabin Zheng, Yibo Liu, Shaowei Liu, Bohong Yin, Weiran He, Han Zhu, Yuzhi Wang, Jianzhou Wang, Mengnan Dong, Zheng Zhang, Yongsheng Kang, Hao Zhang, Xinran Xu, Yutao Zhang, Yuxin Wu, Xinyu Zhou, and Zhilin Yang. Muon is scalable for llm training. *arXiv*, 2025.

Depen Morwani, Itai Shapira, Nikhil Vyas, Eran Malach, Sham Kakade, and Lucas Janson. A new perspective on Shampoo's preconditioner. In *International Conference on Learning Representations (ICLR)*, 2024.

Sebastian Raschka. Build a large language model (from scratch). https://github.com/rasbt/LLMs-from-scratch, 2024. Accessed: 2025-10.

Andrei Semenov, Matteo Pagliardini, and Martin Jaggi. Benchmarking optimizers for large language model pretraining. *arXiv preprint arXiv:2509.01440*, 2025.

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

Charles Van Loan. The block Jacobi method for computing the singular value decomposition. Technical report, Cornell University, 1985.

Nikhil Vyas, Depen Morwani, Rosie Zhao, Itai Shapira, David Brandfonbrener, Lucas Janson, and Sham M Kakade. SOAP: Improving and stabilizing Shampoo using Adam for language modeling. In *International Conference on Learning Representations (ICLR)*, 2025.

Yusaku Yamamoto, Zhang Lan, and Shuhei Kudo. Convergence analysis of the parallel classical block Jacobi method for the symmetric eigenvalue problem. *JSIAM Letters*, 2014.

Ashley Zhang, Ben Keigwin, Dhruv Pai, and Alec Dewulf. Online KL shampoo, 2026. URL https://blog.tilderesearch.com/blog/online-kl-shampoo. Public technical report and open-source release.

A Make any QR Implementation Unique

For a non-singular square matrix $\boldsymbol{A}$, we can always make any QR implementation unique by requiring the upper-triangular matrix $\boldsymbol{R}$ to have *positive* diagonal entries. This is possible because $\boldsymbol{R}$ must have non-zero diagonal entries when $\boldsymbol{A}$ is non-singular. We then use the following procedure to make the decomposition unique

$$\mathrm{qr}(\boldsymbol{A}) = \underbrace{\boldsymbol{Q}\boldsymbol{R}}_{\text{non-unique QR}} = \underbrace{\boldsymbol{Q}\boldsymbol{D}}_{=\bar{\boldsymbol{Q}}}\underbrace{\boldsymbol{D}^\top\boldsymbol{R}}_{=\bar{\boldsymbol{R}}}$$

where $\boldsymbol{D}\boldsymbol{D}^\top = \boldsymbol{I}$ and $\boldsymbol{D} := \mathrm{Diag}(\mathrm{sign}(\mathrm{diag}(\boldsymbol{R})))$ is constructed to be a diagonal sign matrix so that $\bar{\boldsymbol{R}} := \boldsymbol{D}\boldsymbol{R}$ has positive diagonal entries. Note that $\bar{\boldsymbol{Q}}$ is orthogonal and $\bar{\boldsymbol{R}}$ is upper-triangular. Thus, we obtain a unique QR decomposition of $\boldsymbol{A}$ via $\boldsymbol{A} = \bar{\boldsymbol{Q}}\bar{\boldsymbol{R}}$.

B More Implementation Details of Our Customized QR

Building on recent advances in QR decomposition using NVIDIA B200 GPUs (GPU MODE, 2026), we adapt techniques from the top three leaderboard submissions to the structured inputs arising in Shampoo-based methods. These include Cholesky-QR on blocks of columns (panels), Newton-corrected panel factors, tensor-core triangular inverses, split-precision matrix products, and input-dependent routing with CUDA-graph capture. Our QR supports a general input with an option to exploit the SPD structure. The code for general inputs uses two block Gram–Schmidt sweeps, with Cholesky factorizations, triangular solves, and Newton corrections for panel orthogonalization. Our implementation is pure PyTorch, supports mixed-precision computation—the large panel-level products (projections, Gram matrices, and the application of triangular factors) run as FP16 tensor-core GEMMs with FP32 accumulation, using split FP16 operands with compensating cross terms where FP32-level accuracy is required, while the small panel-sized Cholesky, inverse, and Newton computations and the numerical checks remain in FP32—and batched execution, and runs on NVIDIA A100, H100, and newer GPUs. Unlike `torch.linalg.qr`, our implementation enforces the positive-diagonal convention, which uniquely specifies the orthogonal factor $\boldsymbol{Q}$ for nonsingular inputs. Our implementation returns only $\boldsymbol{Q}$, without assembling the full triangular factor $\boldsymbol{R}$. We also do lightweight orthogonality and triangularity checks to assess numerical accuracy. Our QR implementation internally uses FP16 mixed-precision computation to reduce cost and preserve numerical accuracy, while our training pipeline and optimizers use BFP16 for mixed-precision computation.

The reparametrized factor $\boldsymbol{P}$ is symmetric positive definite and near-diagonal when the current basis closely approximates an eigenbasis. Inexpensive estimates of near-diagonality and numerically inactive directions guide routing. When handling this structural input, we exploit the fact that $\boldsymbol{P}$ and its Cholesky factor share the same orthogonal QR factor. For suitable near-diagonal inputs, our implementation uses a single block Gram–Schmidt sweep instead of two sweeps used for a general input, while batched Newton corrections replace panel Cholesky factorizations and triangular solves. Numerically inactive directions are identified from the diagonal and regularized by zeroing their off-diagonal entries and flooring their diagonal entries.

C Parallel update for disjoint subspaces via subspace QR

We now describe a parallel update of two disjoint subspaces. For illustration, partition the basis indices into two disjoint sets $\mathbb{I}_i^{(X)}$ and $\mathbb{I}_i^{(Y)}$ and group them to write the factor $\boldsymbol{P}_i$ as

$$\boldsymbol{P}_i = \begin{bmatrix} \boldsymbol{P}_i^{(XX)} & \boldsymbol{P}_i^{(XY)} \\ \boldsymbol{P}_i^{(YX)} & \boldsymbol{P}_i^{(YY)} \end{bmatrix},$$

where the principal blocks $\boldsymbol{P}_i^{(XX)} = \boldsymbol{P}_i[\mathbb{I}_i^{(X)}, \mathbb{I}_i^{(X)}]$ and $\boldsymbol{P}_i^{(YY)} = \boldsymbol{P}_i[\mathbb{I}_i^{(Y)}, \mathbb{I}_i^{(Y)}]$, highlighted in red and blue, are both selected for the update.

We perform QR decomposition on these two blocks independently and in parallel, using the same $\boldsymbol{P}_i$, to obtain the local orthogonal factors

$$\boldsymbol{O}_i^{(XX)} := \mathrm{qr}\Big(\boldsymbol{P}_i^{(XX)}\Big), \qquad \boldsymbol{O}_i^{(YY)} := \mathrm{qr}\Big(\boldsymbol{P}_i^{(YY)}\Big).$$

We then define $\boldsymbol{O}_i := \boldsymbol{O}_i^{(XX)} \oplus \boldsymbol{O}_i^{(YY)}$, where $\oplus$ denotes the direct sum, and apply this block-diagonal rotation through eqs. (4) and (5) without explicitly forming the full rotation matrix. The basis update $\boldsymbol{Q}_i^{(\mathrm{new})} = \boldsymbol{Q}_i^{(\mathrm{old})}\boldsymbol{O}_i$ rotates the two disjoint column groups in parallel:

Parallel Updates of Two Disjoint Subspaces via QR

$$\begin{bmatrix} (\boldsymbol{Q}_i^{(\mathrm{old})})^{(XX)} & (\boldsymbol{Q}_i^{(\mathrm{old})})^{(XY)} \\ (\boldsymbol{Q}_i^{(\mathrm{old})})^{(YX)} & (\boldsymbol{Q}_i^{(\mathrm{old})})^{(YY)} \end{bmatrix} \begin{bmatrix} \boldsymbol{O}_i^{(XX)} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{O}_i^{(YY)} \end{bmatrix} = \begin{bmatrix} \underbrace{(\boldsymbol{Q}_i^{(\mathrm{old})})^{(XX)}\boldsymbol{O}_i^{(XX)}}_{\text{updated } X \text{ basis}} & \underbrace{(\boldsymbol{Q}_i^{(\mathrm{old})})^{(XY)}\boldsymbol{O}_i^{(YY)}}_{\text{updated } Y \text{ basis}} \\ (\boldsymbol{Q}_i^{(\mathrm{old})})^{(YX)}\boldsymbol{O}_i^{(XX)} & (\boldsymbol{Q}_i^{(\mathrm{old})})^{(YY)}\boldsymbol{O}_i^{(YY)} \end{bmatrix}. \tag{8}$$

Because both local factors are orthogonal, their direct sum $\boldsymbol{O}_i$ is orthogonal. The two updated column groups therefore remain mutually orthogonal, preserving full-basis orthogonality without full-basis reorthogonalization.

D GPU-FRIENDLY GREEDY SELECTION OF A SINGLE SUBSPACE

Existing block Jacobi-type selection methods (Bečka et al., 2002; Yamamoto et al., 2014) require an additional loop to select a block. We instead consider a loop-free, GPU-friendly alternative: **a greedy two-phase method** inspired by the Jacobi method. In phase 1, we select the classical greedy Jacobi pair (Forsythe and Henrici, 1960),

$$\text{phase 1}: \quad (k^*, j^*) = \arg\max_{k \neq j} P_{k,j}^2,$$

and in phase 2, we expand this pair into a block of size b by choosing the $b-2$ indices with the largest total coupling to (k^*, j^*),

$$\text{phase 2}: \quad S^* = \arg\max_{\substack{S \subseteq \|\boldsymbol{P}\| \setminus \{k^*, j^*\} \\ |S| = b-2}} \sum_{x \in S} \Big(P_{x,k^*}^2 + P_{x,j^*}^2 \Big).$$

Here we drop the subscript of $\boldsymbol{P}_i$, $\|\boldsymbol{P}\|$ denotes the index set of $\boldsymbol{P}_i$, b is the block size, $\mathbb{I}_i := S^* \cup \{k^*, j^*\}$ is the selected index set, and $P_{k,j}$ denotes the (k,j)-th entry of $\boldsymbol{P}_i$. This can be implemented efficiently with `torch.topk`.

E ADDITIONAL EXPERIMENTAL DETAILS

We conduct five sets of experiments on four language models—nanoGPT (123M) and Llama 3 (119M, 313M, and 598M)—using the FineWeb dataset from Hugging Face. We use the default train/test split in all experiments. As strong baselines for training matrix-valued weights in neural networks, we consider Muon (Liu et al., 2025) with the Polar Express backend (Amsel et al., 2026), SOAP (Vyas et al., 2025), and KL-Shampoo/KL-SOAP (Lin et al., 2026). For vector-valued weights, such as those in normalization layers, we use AdamW.

In the nanoGPT experiments (Jordan, 2024), we use the warmup-stable-decay learning-rate schedule, a batch size of 512, and a sequence length of 1024. In all Llama 3 experiments (Semenov et al., 2025), we use cosine learning-rate scheduling, a batch size of 768, and a sequence length of 512. For each baseline, we tune all available hyperparameters using random search with 120 runs. Our search follows a two-stage policy. In Stage 1, we explore a wider search range with 60 runs and narrow the range based on test loss. In Stage 2, we refine the search range and perform an additional 60 runs. Reparametrized variants reuse the best optimizer hyperparameters found for their original counterparts, without additional tuning. For all models, namely nanoGPT (123M), Llama3-small (119M), Llama3-mid (313M), and Llama3-big (598M), we train on two H100 GPUs. Because computational resources are limited, we train each model for 10,000 iterations and report the performance of each method. See table 1 for the weight matrices trained by these methods.

	`llama3-big`	`llama3-mid`	`llama3-small`	`nanogpt`
Attention	$24 \times (1920, 1920)$ $+ 24 \times (320, 1920)$	$24 \times (1344, 1344)$ $+ 24 \times (224, 1344)$	$24 \times (768, 768)$ $+ 24 \times (128, 768)$	$48 \times (768, 768)$
MLP	$24 \times (5760, 1920)$ $+ 12 \times (1920, 5760)$	$24 \times (4032, 1344)$ $+ 12 \times (1344, 4032)$	$24 \times (2304, 768)$ $+ 12 \times (768, 2304)$	$12 \times (3072, 768)$ $+ 12 \times (768, 3072)$
# matrices	$48 + 36 = 84$	$48 + 36 = 84$	$48 + 36 = 84$	$48 + 24 = 72$

Table 1: Number and shape of the 2-D weight matrices used for Shampoo/SOAP/Muon in each model. Shapes are `nn.Linear` weights, (out, in). All models have 12 layers and no biases; the tied embedding/LM head is excluded.

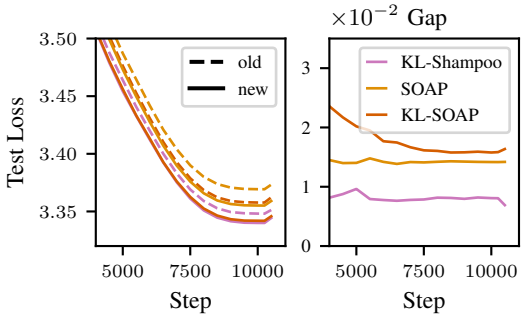

Figure 5: Test loss and per-method gap (old − new) on Llama3 (119M) trained on FineWeb 4.1B for KL-Shampoo, SOAP, and KL-SOAP, showing that the reparametrized variants outperform the original baselines with BFP16 mixed-precision computation and full-basis updates. This gap highlights the performance degradation caused by the BFP16 mixed-precision computation under the original parametrization.

F Additional Experiments

Experiment 5: Balancing subspace accuracy and update frequency Block Jacobi methods often employ an inner loop with K local updates per basis refresh. Given the similarity between these methods and ours, we investigate whether multiple inner steps are necessary as the preconditioning factors evolve. **(I) Fixed refresh interval, varying inner-loop length:** We begin by fixing the decomposition interval at $T = 10$ for both full-basis and subspace updates and varying $K \in \{1, 3, 5\}$. As shown in table 5, subspace updates can achieve performance comparable to that of full-basis updates with multiple inner steps, provided that the subspace is not too small. Using multi-step subspace updates can improve training performance at the cost of increased runtime. By contrast, using a single step is inexpensive but gives worse training performance at this refresh interval. **(II) Single-step updates, varying refresh interval:** Unlike the classical setting, $\boldsymbol{P}_i$ evolves during training. This motivates us to vary the decomposition interval T while using one local QR step per selected block at each refresh (i.e., $K = 1$). Rather than accurately diagonalizing a fixed factor, these updates track its evolving basis. Our experiments show (see table 6) that a single subspace update can achieve competitive training performance relative to the full-basis method while reducing the total runtime. Together, these results support the single-step scheme in section 3.3: shorter refresh intervals can compensate for less accurate individual subspace updates.

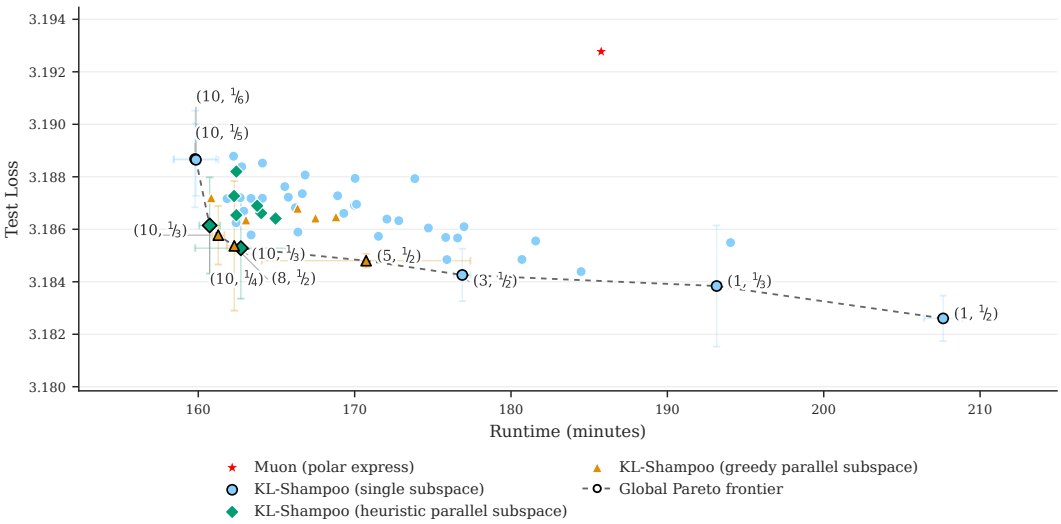

Figure 6: Performance of reparametrized KL-Shampoo varying T, B, and subspace selection strategies (with the BFP16 mixed-precision training) on Llama3 (313M) trained on FineWeb 4.1B, evaluated with the hyperparameters tuned for full-basis KL-Shampoo over 120 runs. Muon is tuned independently over 120 runs. We run three different random seeds per configuration and display one standard deviation error bars for both test loss and runtime along the Pareto frontier. Unlabeled points include multistep subspace updates (as in table 5, table 6) but the Pareto frontier indicates one step is sufficient. We see more compute intensive configurations tend to decrease loss further at the cost of runtime.

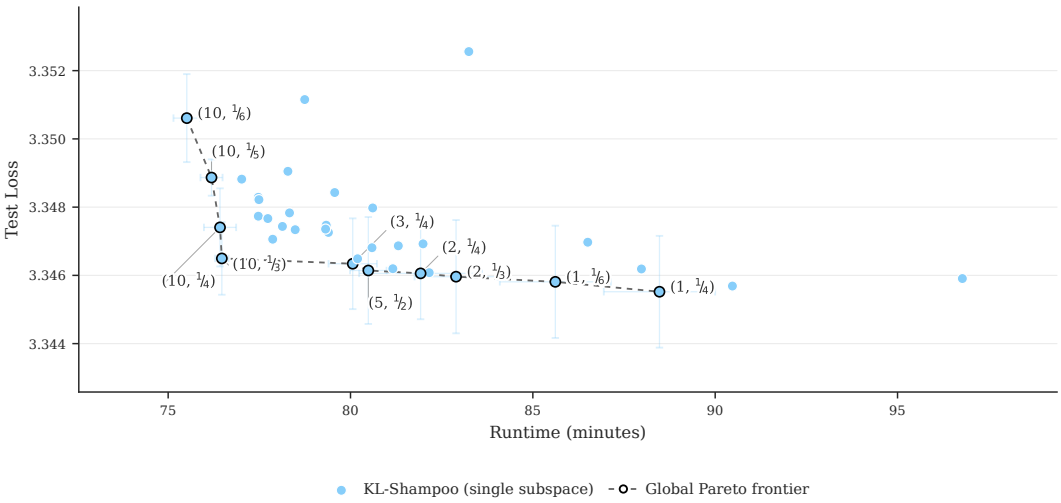

Figure 7: Performance of reparametrized KL-Shampoo varying T, B, and subspace selection strategies (with the BFP16 mixed-precision training) on Llama3 (119M) trained on FineWeb 4.1B, evaluated with the hyperparameters tuned for full-basis KL-Shampoo over 120 runs. We run three different random seeds per configuration and display one standard deviation error bars for both test loss and runtime along the Pareto frontier. Unlabeled points include multistep subspace updates (as in table 5, table 6) but the Pareto frontier indicates one step is sufficient. We see more compute intensive configurations tend to decrease loss further at the cost of runtime.

Table 2: Performance of each method in terms of test loss for full-basis updates. Each cell shows the result with full FP32 computation followed by mixed-precision computation (BFP16); the number above the arrow is the (BFP16−FP32) delta. Small deltas indicate the parameterization is robust to half-precision computation. Best result per column (FP32 and BFP16 separately) is shown in bold. Our reparameterization is consistently best when switching from FP32 to BFP16.

Method	**Parameterization**	**nanoGPT (123M)** FP32 → BFP16	**llama3 (119M)** FP32 → BFP16
KL-Shampoo	old	$3.239 \xrightarrow{+0.010} 3.249$	$\mathbf{3.345} \xrightarrow{+0.000} 3.345$
	new	$3.239 \xrightarrow{+0.005} 3.244$	$\mathbf{3.345} \xrightarrow{-0.001} \mathbf{3.344}$
KL-SOAP	old	$3.239 \xrightarrow{+0.013} 3.252$	$3.351 \xrightarrow{+0.011} 3.362$
	new	$\mathbf{3.235} \xrightarrow{+0.006} \mathbf{3.241}$	$3.346 \xrightarrow{+0.000} 3.346$
SOAP	old	$3.248 \xrightarrow{+0.013} 3.261$	$3.358 \xrightarrow{+0.014} 3.372$
	new	$3.247 \xrightarrow{+0.005} 3.252$	$3.353 \xrightarrow{+0.006} 3.359$

Table 3: This demonstrates how subspace selection methods supported by our reparametrization can affect the performance of KL-Shampoo (ours, BFP16, QR) on test loss. As shown below, the greedy method is preferred over the random method due to its robustness on different models.

T	B	K	**Select**	**nanoGPT (123M)**	**llama3 (119M)**
10	1/2	1	random	3.249	3.345
			greedy	3.247	3.344
		3	random	3.249	3.344
			greedy	3.245	3.344
		5	random	3.249	3.345
			greedy	3.244	3.345

Table 4: This demonstrates that subspace orthogonal bases of KL-Shampoo (ours, BFP16, greedy) can be updated via an eigen- or QR-decomposition, as supported by our representation. As shown below, both decomposition methods perform similarly on test loss. This motivates the use of QR, as it is lower-cost.

T	B	K	**Basis**	**nanoGPT (123M)**	**llama3 (119M)**
10	1/2	1	Eig	3.247	3.345
			QR	3.247	3.344
		3	Eig	3.245	3.344
			QR	3.245	3.344
		5	Eig	3.245	3.344
			QR	3.244	3.345

Table 5: This shows that performing multi-subspace updates in KL-Shampoo (ours, BFP16, QR, greedy) can improve accuracy at the cost of increased runtime. We update the basis via multi-step procedures ($K = 1, 3, 5$) in subspaces. We consider different subspace sizes ($d_{\text{sub},i} = Bd_i$).

T	B	K	**nanoGPT (123M)**	**llama3 (119M)**	**llama3 (313M)**
10	1	1	3.244	3.345	3.183
10	1/2	1	3.247	3.347	3.187
		3	3.245	3.346	3.186
		5	3.244	3.395	3.185
10	1/4	1	3.251	3.347	3.188
		3	3.251	3.351	3.188
		5	3.248	3.347	3.188
10	1/6	1	3.256	3.351	3.189
		3	3.251	3.348	3.186
		5	3.252	3.348	3.186

Table 6: This shows that updating the KL-Shampoo basis (ours, BFP16, QR, greedy) more frequently ($T = 2, 3, 4, 5$) can be beneficial even when updating only a subset of the basis ($d_{\text{sub},i} = Bd_i$). A cheap subspace update can be useful when the basis is updated more frequently.

T	B	K	**nanoGPT (123M)**	**llama3 (119M)**	**llama3 (313M)**
2	1/3	1	3.247	3.346	3.186
	1/4		3.250	3.346	3.185
	1/5		3.252	3.347	3.186
3	1/3	1	3.246	3.346	3.186
	1/4		3.251	3.346	3.187
	1/5		3.253	3.347	3.187
4	1/3	1	3.247	3.357	3.187
	1/4		3.252	3.347	3.187
	1/5		3.250	3.349	3.187
5	1/3	1	3.249	3.347	3.187
	1/4		3.252	3.348	3.188
	1/5		3.254	3.348	3.187

SOAP (Original)	**SOAP (Reparameterized)**
$\{\boldsymbol{Q}_i, \boldsymbol{S}_i \mid i = 1, 2\}$	$\{\boldsymbol{Q}_i, \boldsymbol{P}_i := \boldsymbol{Q}_i^\top \boldsymbol{S}_i \boldsymbol{Q}_i \mid i = 1, 2\}$
Both store rotated momentum $\tilde{\boldsymbol{M}}$ and coordinatewise second moments $\tilde{\boldsymbol{d}}$.	
1. Compute the gradient	Every iteration
$\boldsymbol{g} := \nabla \ell(\boldsymbol{\theta}), \qquad \boldsymbol{G} := \mathrm{Mat}(\boldsymbol{g}) \in \mathbb{R}^{d_1 \times d_2}$	
2. Update preconditioning factors	EMA; every iteration; full basis
$\boldsymbol{G}_1 := \boldsymbol{G}$ $\boldsymbol{G}_2 := \boldsymbol{G}^\top$ $\Delta_i := \boldsymbol{G}_i \boldsymbol{G}_i^\top$ $\boldsymbol{S}_i \leftarrow (1-\beta_2)\boldsymbol{S}_i + \beta_2 \Delta_i$	$\tilde{\boldsymbol{G}}_1 := \boldsymbol{Q}_1^\top \boldsymbol{G}$ $\tilde{\boldsymbol{G}}_2 := \boldsymbol{Q}_2^\top \boldsymbol{G}^\top$ $\tilde{\Delta}_i := \tilde{\boldsymbol{G}}_i \tilde{\boldsymbol{G}}_i^\top$ $\boldsymbol{P}_i \leftarrow (1-\beta_2)\boldsymbol{P}_i + \beta_2 \tilde{\Delta}_i$
3. Update the eigenbasis	Full-basis QR; every $T \geq 1$ iterations
$\boldsymbol{Z}_i \leftarrow \boldsymbol{S}_i \boldsymbol{Q}_i$ $\boldsymbol{M} \leftarrow \boldsymbol{Q}_1 \tilde{\boldsymbol{M}} \boldsymbol{Q}_2^\top$ (old basis) $\boldsymbol{Q}_i \leftarrow \mathrm{qr}(\boldsymbol{Z}_i)$ $\tilde{\boldsymbol{M}} \leftarrow \boldsymbol{Q}_1^\top \boldsymbol{M} \boldsymbol{Q}_2$ (new basis)	$\boldsymbol{O}_i \leftarrow \mathrm{qr}(\boldsymbol{P}_i)$ $\boldsymbol{Q}_i \leftarrow \boldsymbol{Q}_i \boldsymbol{O}_i$ $\boldsymbol{P}_i \leftarrow \boldsymbol{O}_i^\top \boldsymbol{P}_i \boldsymbol{O}_i$ $\tilde{\boldsymbol{M}} \leftarrow \boldsymbol{O}_1^\top \tilde{\boldsymbol{M}} \boldsymbol{O}_2$
4. Update rotated first and second moments	EMA; every iteration
$\tilde{\boldsymbol{G}}' := \boldsymbol{Q}_1^\top \boldsymbol{G} \boldsymbol{Q}_2, \qquad \tilde{\boldsymbol{g}}' := \mathrm{vec}(\tilde{\boldsymbol{G}}')$ (current basis) $\tilde{\boldsymbol{M}} \leftarrow (1-\beta_1)\tilde{\boldsymbol{M}} + \beta_1 \tilde{\boldsymbol{G}}',$ $\tilde{\boldsymbol{d}} \leftarrow (1-\beta_2)\tilde{\boldsymbol{d}} + \beta_2 (\tilde{\boldsymbol{g}}')^{\odot 2}.$	
5. Precondition and update parameters	Learning rate γ
$\boldsymbol{Q} := \boldsymbol{Q}_1 \otimes \boldsymbol{Q}_2, \qquad \tilde{\boldsymbol{m}} := \mathrm{vec}(\tilde{\boldsymbol{M}})$ $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma \boldsymbol{Q}\, \mathrm{Diag}(\tilde{\boldsymbol{d}})^{-1/2} \tilde{\boldsymbol{m}} \equiv \boldsymbol{\theta} - \gamma \boldsymbol{Q} \big(\tilde{\boldsymbol{m}} \odot \tilde{\boldsymbol{d}}^{\odot -1/2}\big).$	

Figure 8: **SOAP and its reparameterized variant.** Replacing $\boldsymbol{S}_i$ with $\boldsymbol{P}_i := \boldsymbol{Q}_i^\top \boldsymbol{S}_i \boldsymbol{Q}_i$ while retaining rotated momentum $\tilde{\boldsymbol{M}}$ establishes the same factor-and-momentum storage convention as reparameterized KL-Shampoo and KL-SOAP (figs. 2 and 9). Like KL-SOAP, SOAP maintains coordinatewise second moments $\tilde{\boldsymbol{d}}$ for preconditioning. At each basis refresh, $\boldsymbol{O}_i$ rotates both $\boldsymbol{P}_i$ and $\tilde{\boldsymbol{M}}$ (step 3), after which the moment updates use the current basis (step 4). Both variants produce equivalent iterates. **Color code.** Red marks reparameterized factors; blue marks $\tilde{\boldsymbol{G}}_i = \boldsymbol{Q}_i^\top \boldsymbol{G}_i$, with $\tilde{\Delta}_i = \boldsymbol{Q}_i^\top \Delta_i \boldsymbol{Q}_i$.

KL-SOAP (Original)	**KL-SOAP (Reparameterized)**
$\{\boldsymbol{\lambda}_i, \boldsymbol{Q}_i, \boldsymbol{S}_i \mid i = 1, 2\}$	$\{\boldsymbol{\lambda}_i, \boldsymbol{Q}_i, \boldsymbol{P}_i := \boldsymbol{Q}_i^\top \boldsymbol{S}_i \boldsymbol{Q}_i \mid i = 1, 2\}$
Both store rotated momentum $\tilde{\boldsymbol{M}}$ and coordinatewise second moments $\tilde{\boldsymbol{d}}$.	
1. Compute the gradient	Every iteration
$\boldsymbol{g} := \nabla \ell(\boldsymbol{\theta}), \qquad \boldsymbol{G} := \mathrm{Mat}(\boldsymbol{g}) \in \mathbb{R}^{d_1 \times d_2}$	
2. Update preconditioning factors	EMA; every iteration; full basis
$\boldsymbol{G}'_1 := \boldsymbol{G}\boldsymbol{Q}_2 \,\mathrm{Diag}(\boldsymbol{\lambda}_2^{-1/2})/\sqrt{d_2}$ $\boldsymbol{G}'_2 := \boldsymbol{G}^\top \boldsymbol{Q}_1 \,\mathrm{Diag}(\boldsymbol{\lambda}_1^{-1/2})/\sqrt{d_1}$ $\Delta_i := \boldsymbol{G}'_i \boldsymbol{G}_i'^\top$ $\boldsymbol{S}_i \leftarrow (1-\beta_2)\boldsymbol{S}_i + \beta_2 \Delta_i$	$\tilde{\boldsymbol{G}}' := \boldsymbol{Q}_1^\top \boldsymbol{G} \boldsymbol{Q}_2$ $\tilde{\boldsymbol{G}}'_1 := \tilde{\boldsymbol{G}}' \,\mathrm{Diag}(\boldsymbol{\lambda}_2^{-1/2})/\sqrt{d_2}$ $\tilde{\boldsymbol{G}}'_2 := \tilde{\boldsymbol{G}}'^\top \,\mathrm{Diag}(\boldsymbol{\lambda}_1^{-1/2})/\sqrt{d_1}$ $\tilde{\Delta}_i := \tilde{\boldsymbol{G}}'_i \tilde{\boldsymbol{G}}_i'^\top$ $\boldsymbol{P}_i \leftarrow (1-\beta_2)\boldsymbol{P}_i + \beta_2 \tilde{\Delta}_i$
3. Track eigenvalues	EMA; every iteration; full basis
$\tilde{\boldsymbol{G}}'_i := \boldsymbol{Q}_i^\top \boldsymbol{G}'_i$ $\mathrm{diag}(\tilde{\Delta}_i) := (\tilde{\boldsymbol{G}}'_i \odot \tilde{\boldsymbol{G}}'_i)\mathbf{1}$	Reuse $\tilde{\Delta}_i$ from step 2. No additional matrix products.
$\boldsymbol{\lambda}_i \leftarrow (1-\beta_2)\boldsymbol{\lambda}_i + \beta_2 \,\mathrm{diag}(\tilde{\Delta}_i)$	
4. Update the eigenbasis	Full-basis QR; every $T \geq 1$ iterations
$\boldsymbol{Z}_i \leftarrow \boldsymbol{S}_i \boldsymbol{Q}_i$ $\boldsymbol{M} \leftarrow \boldsymbol{Q}_1 \tilde{\boldsymbol{M}} \boldsymbol{Q}_2^\top$ (old basis) $\boldsymbol{Q}_i \leftarrow \mathrm{qr}(\boldsymbol{Z}_i)$ $\tilde{\boldsymbol{M}} \leftarrow \boldsymbol{Q}_1^\top \boldsymbol{M} \boldsymbol{Q}_2$ (new basis)	$\boldsymbol{O}_i \leftarrow \mathrm{qr}(\boldsymbol{P}_i)$ $\boldsymbol{Q}_i \leftarrow \boldsymbol{Q}_i \boldsymbol{O}_i$ $\boldsymbol{P}_i \leftarrow \boldsymbol{O}_i^\top \boldsymbol{P}_i \boldsymbol{O}_i$ $\tilde{\boldsymbol{M}} \leftarrow \boldsymbol{O}_1^\top \tilde{\boldsymbol{M}} \boldsymbol{O}_2$
5. Update rotated first and second moments	EMA; every iteration
Compute using the current basis. $\tilde{\boldsymbol{G}}' := \boldsymbol{Q}_1^\top \boldsymbol{G} \boldsymbol{Q}_2$	Reuse step 2; rotate only after a basis refresh: $\tilde{\boldsymbol{G}}' \leftarrow \boldsymbol{O}_1^\top \tilde{\boldsymbol{G}}' \boldsymbol{O}_2$
$\tilde{\boldsymbol{g}}' := \mathrm{vec}(\tilde{\boldsymbol{G}}')$ (current basis) $\tilde{\boldsymbol{M}} \leftarrow (1-\beta_1)\tilde{\boldsymbol{M}} + \beta_1 \tilde{\boldsymbol{G}}',$ $\tilde{\boldsymbol{d}} \leftarrow (1-\beta_2)\tilde{\boldsymbol{d}} + \beta_2 (\tilde{\boldsymbol{g}}')^{\odot 2}.$	
6. Precondition and update parameters	Learning rate γ
$\boldsymbol{Q} := \boldsymbol{Q}_1 \otimes \boldsymbol{Q}_2, \qquad \tilde{\boldsymbol{m}} := \mathrm{vec}(\tilde{\boldsymbol{M}})$ $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma \boldsymbol{Q} \,\mathrm{Diag}(\tilde{\boldsymbol{d}})^{-1/2} \tilde{\boldsymbol{m}} \equiv \boldsymbol{\theta} - \gamma \boldsymbol{Q} \big(\tilde{\boldsymbol{m}} \odot \tilde{\boldsymbol{d}}^{\odot -1/2}\big).$	

Figure 9: **KL-SOAP and its reparameterized variant.** KL-SOAP combines KL-Shampoo's factor and eigenvalue updates (fig. 2, steps 2–3) with SOAP's momentum update, coordinatewise moment updates, and preconditioning (fig. 8, steps 3–5). Storing $\boldsymbol{P}_i := \boldsymbol{Q}_i^\top \boldsymbol{S}_i \boldsymbol{Q}_i$ and rotated momentum $\tilde{\boldsymbol{M}} := \boldsymbol{Q}_1^\top \boldsymbol{M} \boldsymbol{Q}_2$ unifies factor-and-momentum storage across all three reparameterized methods. KL-SOAP retains $\boldsymbol{\lambda}_i$ for factor estimation and $\tilde{\boldsymbol{d}}$ for coordinatewise preconditioning. At each basis refresh, step 4 updates $\tilde{\boldsymbol{M}}$ and step 5 rotates the cached $\tilde{\boldsymbol{G}}'$ before updating both moments. The two KL-SOAP variants produce equivalent iterates. **Color code.** Red marks reparameterized factors; blue marks $\tilde{\boldsymbol{G}}'_i = \boldsymbol{Q}_i^\top \boldsymbol{G}'_i$, with $\tilde{\Delta}_i = \boldsymbol{Q}_i^\top \Delta_i \boldsymbol{Q}_i$.