

Dion: Distributed Orthonormalized Updates

Kwangjun Ahn¹ Byron Xu¹ Natalie Abreu^{1,2} John Langford¹

¹Microsoft Research, AI Frontiers

²Harvard University

Correspondence: {kwangjunahn, byronxu}@microsoft.com

Abstract

Recent work has shown that orthonormal matrix updates speed up neural network optimization, improve training stability, and offer better hyperparameter transfer across model sizes. Applying these updates efficiently when model weights and optimizer states are sharded across a large-scale distributed LLM training system remains a major challenge. We introduce Dion (DIstributed OrthoNormalization), a scalable and communication-efficient orthonormalizing optimizer. Dion leverages low-rank approximation and decoupled momentum buffers, eliminating the need for full gradient synchronization while producing numerically equivalent results. It is compatible with simultaneous DDP, FSDP, and TP parallelism, and it computes an orthonormalized update without unsharding a full parameter matrix on any single device. We evaluate Dion on language models from 120M to 3B parameters and find that its benefits improve with increasing model size and batch size.

1 Introduction

Training state-of-the-art AI models consumes millions of GPU-hours, so improvements to the optimizer’s *update rule* directly reduce cost and increase iteration speed. In this work, we propose a new update rule for *matrix-valued* parameters—such as the attention and MLP matrices in Transformer models—which constitute the bulk of modern neural network weights.

The Muon optimizer [Jordan et al., 2024b] has recently shown that orthonormalizing the weight update matrix can give large speedups for neural network training. By conditioning weight updates to induce consistent changes in hidden states [Bernstein, 2025], orthonormalized updates yield faster convergence, greater training stability, and more robust hyperparameter transfer across model scales [Bernstein and Newhouse, 2024a, Large et al., 2024, Pethick et al., 2025]. A large-scale study by Moonshot AI [Liu et al., 2025a] has shown that Muon achieves nearly twice the computational efficiency of AdamW [Loshchilov and Hutter, 2019] when training 16B parameter models. Similarly, Essential AI [Shah et al., 2025] has found improvements with larger batch sizes.

Yet a fundamental obstacle remains. Muon’s orthonormalization procedure relies on Newton-Schulz iteration [Bernstein and Newhouse, 2024a,b, Jordan et al., 2024b], involving dense matrix-matrix multiplications that do not respect the ways in which modern LLM training shards parameters across devices. The Moonshot AI reference implementation of Muon [Liu et al., 2025a] therefore executes

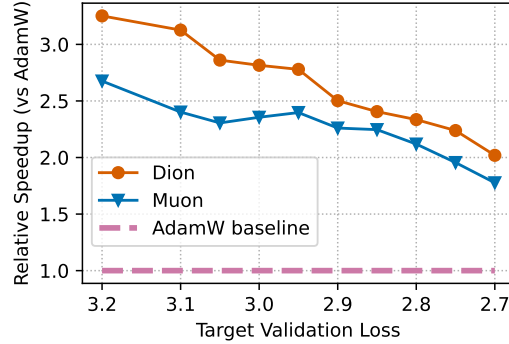


Figure 1: **Speed-up to reach target loss.** We report wall-clock time (relative to AdamW) for training a 3B parameter model to reach each given validation loss. Dion consistently outperforms Muon and demonstrates 2–3× training efficiency over AdamW. See Section 5.4 for experiment details.

the full matrix computation *redundantly* on every shard. As estimated in [Appendix A](#), this design would add over **278 days** (!) of extra optimizer compute to a Llama 3 405B scale training run—a clearly unsustainable cost. See also [\[Essential-AI, 2025\]](#) for a discussion of the overhead introduced by alternative sharding strategies. This bottleneck motivates a central question:

Can we devise a communication- and compute-efficient orthonormalized update rule?

We introduce **Dion** (Distributed OrthoNormalization), which uses low-rank approximation to create an efficient orthonormalizing update rule. Dion is compatible with DDP, FSDP, and TP parallelism, computing orthonormalized updates without unsharding a full parameter matrix on any single device. While Muon orthogonalizes the entire weight update matrix using Newton-Schulz iteration, Dion relies on orthogonalizing smaller matrices using QR and Cholesky decompositions. Combined with decoupled momentum buffers, Dion further compresses the standard data-parallel gradient synchronization while maintaining mathematically equivalent results.

Low-rank approximation allows Dion to exchange between compute and communication efficiency versus per-step update efficiency, through adjusting a *rank fraction* hyperparameter. We observe that larger models are more robust to update sparsification, experiencing smaller declines relative to full-rank learning performance with a given level of rank reduction. We further note that full-rank Dion gives improved per-step update efficiency over Muon, which we attribute to QR decomposition producing more accurate orthonormalization than Newton-Schulz iteration.

We evaluate Dion on language models ranging from 120M to 3B parameters and analyze its scaling behavior with respect to both model size and batch size. Across both axes, we observe better scaling relative to either AdamW or Muon for both per-FLOP and per-step improvements, suggesting that Dion may be the preferred optimizer for training the largest scale models.

2 Parallelism Techniques for Large-Scale Training

Modern neural networks have far outgrown single-GPU training. To fit trillion-parameter models and keep GPU utilization high, practitioners simultaneously apply many dimensions of parallelism. Model parameters and intermediate states are *sharded* over a network of devices, and *collective communication operations* (e.g., all-reduce, reduce-scatter, all-gather) are used to move data at every step. To better understand the advantages of Dion, let us first recall three common parallelism techniques used for large-scale model training:

- **Data Parallelism (DP)**. All model weights and optimizer states are *replicated* across the DP dimension. Each rank processes a different batch of data, so gradients must be averaged (typically by an all-reduce [\[Agarwal et al., 2014\]](#)) before the optimizer step. Canonical implementations include PyTorch DDP [\[Li et al., 2020\]](#) and Horovod [\[Sergeev and Del Balso, 2018\]](#).
- **Fully Sharded Data Parallelism (FSDP)**. FSDP shards model parameters, gradients, and optimizer states to reduce memory usage [\[Rajbhandari et al., 2020, Zhao et al., 2023\]](#). Each layer’s parameters are gathered just-in-time and re-sharded immediately after use. Only one full model layer needs to be materialized at any time, and the optimizer only needs to update its local shard.
- **Tensor Parallelism (TP)**. TP scales compute by slicing individual weight tensors so that every device computes only a fraction of a layer’s forward and backward work [\[Shoeybi et al., 2019, Narayanan et al., 2021\]](#). Parameters, gradients, and optimizer states are permanently sharded in the TP dimension, while intermediate activations are synchronized across devices.

FSDP can be combined with DP, with sharding across an FSDP axis and replication across a different DP axis, in a configuration known as *hybrid-sharded data parallel* [\[Zhao et al., 2023\]](#). Further scaling is achievable using TP along a third axis. The combination of $DP \times FSDP \times TP$ covers all common parallelization and sharding strategies relevant to Dion. For simplicity, we refer to this setup as **3D parallelism** throughout. Other parallelism techniques (e.g. pipeline or expert parallelism) leave the parameters of a single layer intact, and thus require no changes to the optimizer algorithm. Any parallelism axis with different data and gradients can be merged into the DP all-reduce.¹

¹See, e.g., <https://main-horse.github.io/posts/visualizing-6d/> for an illustration.

Algorithm 1 Centralized Dion (see Algorithm 3 for distributed implementation)

Require: Learning rate η , momentum decay μ , rank r , momentum $M_0 = 0 \in \mathbb{R}^{m \times n}$, right factor $Q_0 \in \mathbb{R}^{n \times r}$ (randomly initialized).

```
1: for  $t = 1$  to  $T$  do
2:   Compute gradient  $G_t \in \mathbb{R}^{m \times n}$ 
3:    $B_t \leftarrow M_{t-1} + G_t \in \mathbb{R}^{m \times n}$ 
4:    $P_t, R_t \leftarrow \text{POWERITER1}(B_t; Q_{t-1}) \in \mathbb{R}^{m \times r}, \mathbb{R}^{n \times r}$   $\triangleright$  approximate  $B_t \approx P_t R_t^\top$ 
5:    $\Delta_t = B_t - P_t R_t^\top \in \mathbb{R}^{m \times n}$   $\triangleright$  approximation error
6:    $M_t \leftarrow \mu B_t + (1 - \mu) \Delta_t \in \mathbb{R}^{m \times n}$   $\triangleright$  error feedback
7:    $(\Leftrightarrow M_t \leftarrow B_t - (1 - \mu) P_t R_t^\top)$ 
8:    $Q_t \leftarrow \text{COLUMNNORMALIZE}(R_t) \in \mathbb{R}^{n \times r}$ 
9:    $X_t \leftarrow X_{t-1} - \eta \sqrt{m/n} P_t Q_t^\top \in \mathbb{R}^{m \times n}$   $\triangleright$  scaled orthonormal update
10: end for
11: function  $\text{POWERITER1}(B; Q) \in (\mathbb{R}^{m \times n}, \mathbb{R}^{n \times r})$   $\triangleright$  single power iteration (from  $Q$ )
12:    $P \leftarrow BQ \in \mathbb{R}^{m \times r}$ 
13:    $P \leftarrow \text{ORTHOGONALIZE}(P) \in \mathbb{R}^{m \times r}$   $\triangleright$  using QR decomposition
14:    $R \leftarrow B^\top P \in \mathbb{R}^{n \times r}$ 
15:   return  $P, R$   $\triangleright P$  is orthonormal
16: end function
```

3 Warm-up: Centralized Dion

To build intuition behind Dion, we begin by considering a simplified scenario without weight sharding or distributed computation. For clarity, we consider optimizing a single $m \times n$ parameter matrix, denoted by X . At each step t , let X_t represent the current parameter, G_t the gradient, and M_t the momentum matrix.

3.1 Design Intuition Behind Dion

The Muon optimizer [Jordan et al., 2024b] introduces an effective update rule for matrix parameters based on approximate zeroth-power orthonormalization. Muon maintains a momentum matrix that accumulates gradients over time:

$$M_t \leftarrow \mu M_{t-1} + G_t, \quad \text{for } \mu \in (0, 1),$$

and then applies an update based on the zeroth power of this matrix. Specifically, it approximates

$$U_t V_t^\top, \quad \text{where } M_t = U_t \Sigma_t V_t^\top$$

is the singular value decomposition (SVD) of the momentum matrix. To approximate this zeroth-power orthonormalization efficiently, Jordan et al. [2024b] employ Newton-Schulz iterations. However, Newton-Schulz iteration requires computing full matrix-matrix products, which faces challenges in distributed training when matrices are sharded over many GPUs. Element-wise optimizer algorithms such as Adam, on the other hand, are unaffected by sharding (see Table 2).

To address the limitations of Muon, we propose using lower-rank updates, with rank $r < m, n$. We efficiently compute a rank- r approximation of the momentum matrix using power iteration, followed by column normalization to ensure the update is orthonormal. Using a technique from PowerSGD [Vogels et al., 2019], we warm-start power iteration with the result from the previous optimizer step, so that a single step of power iteration suffices to produce an accurate low-rank approximation. We empirically validate the effectiveness of this approach through an ablation study comparing it with a low-rank approximation using truncated SVD, presented in Section B.1.

Because a low-rank update cannot capture all the information in its input, we apply an **error feedback** mechanism to compensate. Unlike PowerSGD’s error feedback, we reuse the momentum buffer itself and require no additional optimizer state memory. In Section B.2, we present an ablation study demonstrating the importance of error feedback.

Specifically, at each step, we first form the buffer $B_t = M_{t-1} + G_t$ and use power iteration to compute a rank- r approximation

$$P_t R_t^\top \approx B_t, \quad \text{where } P_t \in \mathbb{R}^{m \times r} \text{ and } R_t \in \mathbb{R}^{n \times r}.$$

(Here P_t and R_t have orthogonal columns.) We then incorporate the error of approximation into the next momentum update:

$$M_t \leftarrow \mu B_t + (1 - \mu) \Delta_t \quad \text{where} \quad \Delta_t := B_t - P_t R_t^\top.$$

This is equivalently written as $M_t \leftarrow B_t - (1 - \mu) P_t R_t^\top$. This way, information not captured by the low-rank approximation may propagate across iterations.

Lastly, we apply a scaling factor of $\sqrt{m/n}$ to the orthonormalized update. This scaling has been found to improve learning rate transferability across different model sizes [Bernstein and Newhouse, 2024b, Pethick et al., 2025]. We empirically validate Dion’s learning rate transferability in Section 5.5.

3.2 Non-Matrix Parameters and Learning Rate Compatibility

Parameter	Shape	Norm	Scaling factor
Weight	$d_{\text{out}} \times d_{\text{in}}$	Spectral	$\sqrt{d_{\text{out}}/d_{\text{in}}}$
Bias	$d_{\text{out}} \times 1$	RMS	1
Embedding	$d_{\text{out}} \times 1$	RMS	1
Unembedding	$1 \times d_{\text{in}}$	RMS	$1/\sqrt{d_{\text{in}}}$
Normalization	1×1	RMS	1

Table 1: Normalization and scaling factors for various parameter types. For learning rate transfer, the parameter update should be unit-normed in the specified norm, and the base learning rate should be multiplied by the specified scaling factor. Refer to Appendix D for a more detailed explanation.

Dion’s orthonormal update is designed specifically for *matrix*-shaped weights. For other parameters—such as biases, embedding and unembedding vectors, and learned normalization factors—we instead apply a different *element-wise* (scalar) optimizer algorithm. The same practice is used for Muon, as illustrated by the works of Jordan et al. [2024b] and Liu et al. [2025a]. In experiments here, we show the compatibility of Dion with AdamW [Loshchilov and Hutter, 2019] and Lion [Chen et al., 2023].

The simultaneous use of two optimizer algorithms raises an important question: *How do we find the best learning rates for each optimizer?* We demonstrate a simple yet effective approach that scales a **shared base learning rate** for each parameter type and shape. This technique preserves Dion’s hyperparameter transfer properties across model sizes, allowing a single learning rate to be used for two optimizers across any model scale. See Section 5.5 for experimental results.

We begin by observing that the parameter updates produced by the optimizer algorithms here (before applying the learning rate) are *unit-normed*. Dion and Muon compute orthonormalized matrices, giving a spectral norm of one. Lion computes a sign function, giving an RMS norm of one. Adam has been empirically found to produce an approximately constant RMS norm [Liu et al., 2025a]. Following Yang et al. [2023], we then scale the base learning rate by a per-parameter factor, chosen to let each layer’s parameter updates maintain a roughly constant input-to-output mapping. The learning rate scaling factors are given in Table 1, and a detailed derivation can be found in Appendix D.

For many experiments in this paper, we use AdamW as the scalar optimizer to enable fair comparisons with existing baselines. However, we empirically find that Dion is more compatible with Lion due to Lion’s unit-RMS update property (see Figure 7). As a result, we recommend using the Dion+Lion combination with the scaling factors in Table 1, and we adopt this setting for the speed run experiments reported in Section 5.4.

4 Distributed Dion

4.1 Notational Conventions

The key advantage of Dion lies in its efficient support for distributed training, when model weights are sharded across multiple parallelism dimensions. We use the following notation:

- **DP**, **FS**, and **TP** denote data parallel, fully sharded data parallel, and tensor parallel dimensions.
- FS and TP shard indices are i and j , respectively; DP does not shard weights.
- Sharded tensors are indexed as $X^{(i,j)}$, $G^{(i,j)}$, $M^{(i,j)} \in \mathbb{R}^{m_j \times n_i}$ and $Q^{(i,j)} \in \mathbb{R}^{n_i \times r_j}$.
- Hat notation (e.g., \hat{M} , \hat{G}) refers to per-device local states, which may differ across the DP axis.
- All-reduce by mean and by sum are denoted with \mathbb{E} and Σ , respectively.

4.2 Distributed Orthogonalization

Algorithm 2 DISTRIBUTED-ORTHOGONALIZE($P^{(j)} \in \mathbb{R}^{m_j \times r}$)

Require: Oversampling factor $k \geq r$ (default value $k = \lceil 1.25r \rceil$).

```

1:  $S^{(j)} \leftarrow \mathbb{R}^{k \times m_j} \sim \mathcal{N}(0, 1/\sqrt{k})$  ▷ generate random sketching matrix
2:  $G^{(j)} \leftarrow S^{(j)} P^{(j)} \in \mathbb{R}^{k \times r}$  ▷ begin 1st iteration using randomized QR
3:  $G \leftarrow \Sigma_{\text{TP}}(G^{(j)})$  ▷ TP sharded matrix multiply
4:  $\_, R_1 \leftarrow \text{QR}(G) \in \mathbb{R}^{r \times r}$  ▷ only  $R$  component needed
5:  $B^{(j)} \leftarrow P^{(j)} R_1^{-1} \in \mathbb{R}^{m_j \times r}$ 
6:  $H^{(j)} \leftarrow (B^{(j)})^\top B^{(j)} \in \mathbb{R}^{r \times r}$  ▷ begin 2nd iteration using Cholesky QR
7:  $H \leftarrow \Sigma_{\text{TP}}(H^{(j)})$  ▷ TP sharded matrix multiply
8:  $R_2 \leftarrow \text{CHOLSKY}(H) \in \mathbb{R}^{r \times r}$  ▷ only upper triangular component needed
9:  $\bar{P}^{(j)} \leftarrow B^{(j)} R_2^{-1} \in \mathbb{R}^{m_j \times r}$ 
10: return  $\bar{P}^{(j)} \in \mathbb{R}^{m_j \times r}$ 

```

An important component of Dion is **distributed orthogonalization** of sharded matrices. Given a matrix $P \in \mathbb{R}^{m \times r}$ partitioned row-wise across s shards as $P^{(j)} \in \mathbb{R}^{m_j \times r}$ (i.e. $m = s \cdot m_j$), the objective is to orthonormalize P without reconstructing the full matrix.

The method given in [Algorithm 2](#) is a distributed adaptation of the randomized Cholesky QR algorithm [[Fan et al., 2021](#), [Epperly, 2024](#)]. For the random sketching matrix, [Melnichenko et al. \[2025\]](#) recommend an oversampling factor $k = 1.25r$, which we experimentally find to be effective. [Lemma G.1](#) establishes equivalence between the distributed procedure and a centralized version, provided for reference in [Appendix G](#).

4.3 3D-Parallel Dion

The main result is a 3D-parallel algorithm for computing the Dion update. This algorithm admits four possible variants, depending on whether the power iteration is transposed and whether the Fully Sharded (FS) and Tensor Parallel (TP) axes are swapped. We present one representative variant in [Algorithm 3](#), and describe the remaining alternatives in [Appendix E](#).

The following theorem establishes that the distributed algorithm is mathematically equivalent to centralized Dion.

Theorem 4.1. *The distributed implementation of Dion ([Algorithm 3](#)) is equivalent to the centralized version ([Algorithm 1](#)).*

Proof. See [Appendix F](#) for the proof. □

4.4 Decoupled Momentum

A subtle yet important feature of [Algorithm 3](#) is **decoupled momentum**. Although each data-parallel device maintains a separate local momentum buffer, the final optimizer updates remain fully synchronized and are provably equivalent to those of centralized Dion. This stands in contrast to prior methods such as DeMo [[Peng et al., 2024](#)], whose decoupled momentum only approximates synchronous momentum and does not result in such equivalence.

In [Algorithm 3](#), each DP worker keeps its own local momentum shard $\hat{M}_{t-1}^{(i,j)}$, which drifts apart as workers process different data. However, $\hat{M}_{t-1}^{(i,j)}$ is only used to update the buffer $\hat{B}_t^{(i,j)}$, which subsequently is only used through two linear projections:

$$\hat{P}_t^{(j)} = \hat{B}_t^{(i,j)} Q_{t-1}^{(i)} \quad \text{and} \quad \hat{R}_t^{(i)} = (\hat{B}_t^{(i,j)})^\top P_t^{(j)}.$$

Both projections are immediately followed by a *mean* all-reduce over the DP axis. Because the momentum update, buffer update, and all-reduce are all linear operations, we have for instance

$$\mathbb{E}_{\text{DP}}[\hat{P}_t^{(j)}] = \mathbb{E}_{\text{DP}}[\hat{B}_t^{(i,j)} Q_{t-1}^{(i)}] = \mathbb{E}_{\text{DP}}[\hat{B}_t^{(i,j)}] Q_{t-1}^{(i)} = B_t^{(i,j)} Q_{t-1}^{(i)},$$

Algorithm 3 Dion (3D-parallel implementation)

Require: Learning rate η , momentum decay μ , rank r

```
1: for  $t = 1$  to  $T$  do
2:   Compute local gradient  $\hat{G}_t^{(i,j)} \in \mathbb{R}^{m_j \times n_i}$ 
3:    $Q_{t-1}^{(i)} \leftarrow \text{ALLGATHER}_{\text{TP}}(Q_{t-1}^{(i,j)}) \in \mathbb{R}^{n_i \times r}$   $\triangleright$  unshard  $Q$  along TP dimension
4:    $\hat{B}_t^{(i,j)} \leftarrow \hat{M}_{t-1}^{(i,j)} + \hat{G}_t^{(i,j)} \in \mathbb{R}^{m_j \times n_i}$   $\triangleright$  accumulate new gradient
5:    $P_t^{(j)}, R_t^{(i)} \leftarrow \text{DISTRIBUTED-POWERITER1}(\hat{B}_t^{(i,j)}; Q_{t-1}^{(i)})$ 
6:    $\hat{M}_t^{(i,j)} \leftarrow \hat{B}_t^{(i,j)} - (1 - \mu) P_t^{(j)} (R_t^{(i)})^\top \in \mathbb{R}^{m_j \times n_i}$   $\triangleright$  error feedback
7:    $Q_t^{(i)} \leftarrow \text{DISTRIBUTED-COLUMNNORMALIZE}(R_t^{(i)})$ 
8:    $X_t^{(i,j)} \leftarrow X_{t-1}^{(i,j)} - \eta \sqrt{m/n} P_t^{(j)} (Q_t^{(i)})^\top \in \mathbb{R}^{m_j \times n_i}$   $\triangleright$  scaled orthonormal update
9:    $Q_t^{(i,j)} \leftarrow \text{SHARD}_{\text{TP}}(Q_t^{(i)}) \in \mathbb{R}^{n_i \times r_j}$   $\triangleright$  reshard  $Q$  along TP dimension
10: end for

1: function  $\text{DISTRIBUTED-POWERITER1}(\hat{B}^{(i,j)}, Q^{(i)}) \in (\mathbb{R}^{m_j \times n_i}, \mathbb{R}^{n_i \times r})$ 
2:    $\hat{P}^{(j)} \leftarrow \hat{B}^{(i,j)} Q^{(i)} \in \mathbb{R}^{m_j \times r}$ 
3:    $P^{(j)} \leftarrow \mathbb{E}_{\text{DP}\Sigma_{\text{FS}}}(\hat{P}^{(j)}) \in \mathbb{R}^{m_j \times r}$   $\triangleright$  FS sharded matrix multiply and DP sync
4:    $P^{(j)} \leftarrow \text{DISTRIBUTED-ORTHOGONALIZE}(P^{(j)})$ 
5:    $\hat{R}^{(i)} \leftarrow (\hat{B}^{(i,j)})^\top P^{(j)} \in \mathbb{R}^{n_i \times r}$ 
6:    $R^{(i)} \leftarrow \mathbb{E}_{\text{DP}\Sigma_{\text{TP}}}(\hat{R}^{(i)}) \in \mathbb{R}^{n_i \times r}$   $\triangleright$  TP sharded matrix multiply and DP sync
7:   return  $P^{(j)}, R^{(i)} \in (\mathbb{R}^{m_j \times r}, \mathbb{R}^{n_i \times r})$ 
8: end function

1: function  $\text{DISTRIBUTED-COLUMNNORMALIZE}(Q^{(i)}) \in \mathbb{R}^{n_i \times r}$ 
2:    $c^{(i)} \leftarrow \text{COLUMN-NORM-SQUARED}(Q^{(i)}) \in \mathbb{R}^r$ 
3:    $c \leftarrow \sqrt{\Sigma_{\text{FS}} c^{(i)}} \in \mathbb{R}^r$   $\triangleright$  all-reduce sum over FS
4:   return  $Q^{(i)} / c \in \mathbb{R}^{n_i \times r}$   $\triangleright$  column-wise divide
5: end function
```

where $B_t^{(i,j)}$ is the corresponding shard of the synchronous buffer in the centralized [Algorithm 1](#). A similar equivalence applies to $\mathbb{E}_{\text{DP}}[\hat{R}_t^{(i)}]$. This yields $P_t^{(j)}$ and $R_t^{(i)}$ that are *identical* to a globally-synchronous momentum up to numerical precision. Consequently, the orthonormal parameter update and error-feedback subtraction match those of centralized Dion.

4.5 Computational Complexity

For an $m \times n$ weight matrix, rank r Dion on an FS \times TP grid incurs

$$\frac{8mnr}{\text{FS} \cdot \text{TP}} + \frac{13mr^2}{2\text{TP}} + \frac{13}{6}r^3 + O(mn) \text{ FLOPs}$$

per device per optimizer step. Details are provided in [Appendix H](#). All $O(mnr)$ computations are sharded over both parallelism axes. The $O(mr^2)$ term is sharded over TP only. The replicated term is merely $O(r^3)$ and becomes negligible when r is much smaller than m, n . The $O(mn)$ term captures element-wise operations, which do not dominate the asymptotic runtime.

For comparison, the five Newton-Schulz iterations used by Muon require

$$20mn^2 + 10n^3 + O(mn) \text{ FLOPs}$$

independently of sharding [[Jordan et al., 2024b](#), [Liu et al., 2025a](#)]. Even in Dion’s *worst-case* scenario with no sparsity and no sharding ($r = n$ and FS = TP = 1), Dion needs approximately

$$14.5mn^2 + 2.17n^3 < 20mn^2 + 10n^3,$$

cutting the $O(n^3)$ term by over $3\times$ and the $O(mn^2)$ term by $\approx 25\%$. With practical ranks ($r < n$) and parallelism, Dion’s advantage widens with both decreased r and with increased FS \times TP sharding.

4.6 Communication and Memory Footprint

Optimizer	DP I/O	FSDP I/O	TP I/O	Memory
Adam	mn	0	0	$2mn$
Muon	mn	mn	mn	mn
Dion	$(m+n)r$	$(m+1)r$	$2nr + 2.25r^2$	$mn + nr$

Table 2: Extra optimizer-step communication and total optimizer state memory required for an $m \times n$ matrix. Dion’s I/O traffic depends on the low-rank factor $r \leq m, n$.

Table 2 compares the *additional* optimizer-step communication for a single $m \times n$ weight matrix. Details are provided in Appendix I. Notably, Dion’s communication volume scales with the low-rank factor rather than with both matrix dimensions. Dion can have significantly lower I/O overhead than Adam or Muon when $r < m, n$.

Dion stores a momentum buffer $M \in \mathbb{R}^{m \times n}$ and a warm-start basis $Q \in \mathbb{R}^{n \times r}$, for a total size of $mn + nr$ —close to Muon’s mn when r is small, and never greater than Adam’s $2mn$. Both optimizer states can be doubly sharded across FS and TP, further shrinking the per-GPU memory footprint.

5 Experimental Results

We present a range of experimental results for Dion, comparing to baselines of AdamW [Loshchilov and Hutter, 2019], Muon [Jordan et al., 2024b], and DeMo [Peng et al., 2024]. All experiments are conducted using GPT-style decoder-only Transformer models at various scales, trained on the FineWeb dataset [Penedo et al., 2024] or the FineWeb-Edu dataset [Lozhkov et al., 2024], following the setup of the modded-nanoGPT codebase [Jordan et al., 2024a]. For additional details on training configurations and hyperparameters, please refer to Appendix J.

5.1 Performance Across Model Size

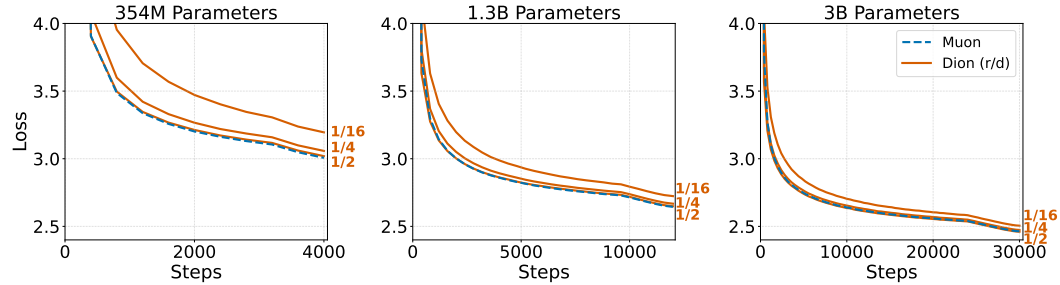


Figure 2: Validation loss for Dion at three low-rank settings ($r = d/2, d/4, d/16$) versus (full-rank) Muon. As the model size increases, Dion with $r = d/2$ or $d/4$ gradually matches Muon, while even the highly sparse $d/16$ variant converges within a small margin.

Figure 2 shows validation loss curves across three model sizes. We compare Dion with rank fraction $r/d = 1/2, 1/4$, and $1/16$, where r is Dion’s low-rank factor and d is the hidden dimension of each Transformer model. Larger models appear more robust towards sparsification. At a scale of 3B parameters, Dion with $r = d/2$ and $d/4$ roughly match Muon, even though Muon uses full-rank orthonormalization. Furthermore, the gap between $r = d/16$ and $r = d/2$ diminishes as the model size increases. Considering that modern LLMs are typically far larger than 3B parameters, these results suggest that $1/16$ and potentially lower rank fractions are realistic for practical use.

In Figure 3 (left), we compare the peak convergence rates of Dion and Muon across multiple model sizes. We use $r = d$ for Dion, since Muon computes full-rank updates. For each size, we define a target validation loss, selected to match the loss achieved by AdamW after consuming approximately 80% of the Chinchilla-optimal token count. We show the *steps ratio*—the number of steps required by AdamW to reach the target loss, divided by the steps required by Dion or Muon. Dion consistently achieves a faster reduction in loss per training step than both Muon and AdamW across all model scales. See Section J.1 for full experimental details.

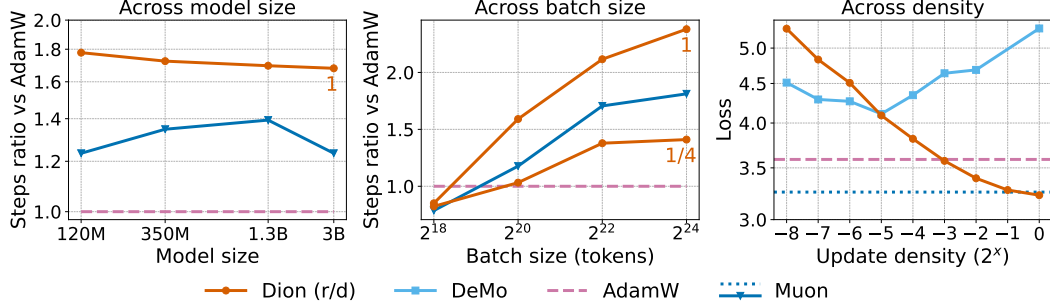


Figure 3: Scaling experiments along three axes: model size, batch size, and update density.

Left: model size. Convergence rate across model sizes for Dion (with $r = d$) and Muon. To measure relative speedup compared to AdamW, we show the steps ratio (AdamW steps / Dion or Muon steps). Higher is better.

Middle: batch size. Scaling behavior with increasing batch size. We show the steps ratio relative to AdamW for Dion (with $r = d$ and $r = d/4$) and Muon. Higher is better.

Right: update density. Dion compared to DeMo, Muon, and AdamW as the rank fraction r/d increases. Dion improves with higher update density, matching DeMo at $r/d = 1/32$ and AdamW at $r/d = 1/8$.

5.2 Performance Across Batch Size

In Figure 3 (middle), we evaluate optimizer performance across a range of batch sizes by measuring the number of steps required to reach a target validation loss of 3.4. We show the steps ratio for Dion (with $r = d$ and $r = d/4$) and Muon. As batch size increases, all configurations exhibit greater advantages relative to AdamW. All experiments use 120M-parameter models with a fixed sequence length of 1024 tokens. Additional hyperparameter details are specified in Section J.2. A further analysis of critical batch size in the style of Zhang et al. [2025] is provided in Appendix C.

5.3 Performance Across Rank Fraction

Dion can decrease communication requirements by lowering the rank fraction r/d . We compare Dion to DeMo, an optimizer that reduces communication using Discrete Cosine Transform (DCT) based compression. DeMo uses two hyperparameters to adjust compression: top- k frequency selection and chunk size $s \times s$. To draw a fair comparison with DeMo, we define the *update density* for Dion as r/d and for DeMo as k/s^2 , the number of selected elements out of the total per chunk. AdamW and Muon do not use compression, and can be regarded as having a fixed update density of 1.

In Figure 3 (right), we see that Dion remains highly competitive with our baselines across moderate levels of compression. For instance, Dion consistently surpasses DeMo beyond update density $1/32$, outperforms AdamW starting at $1/8$, and slightly outperforms Muon at full rank ($r = d$). All experiments here use 120M parameter models trained with a batch size of 4M tokens and a sequence length of 1024. Additional hyperparameters are specified in Section J.3.

Perhaps unsurprisingly, Dion performs best at full rank. It undergoes a smooth and predictable decline in performance as the rank fraction decreases. DeMo, on the other hand, achieves its best performance at an update density of $1/32$, and begins to show poorer performance with both increasing and decreasing compression. We note that for a sufficiently low update density, DeMo will surpass Dion. In Appendix K, we explore a modification to Dion that improves low-rank performance.

5.4 Speed Runs for 3B Models

Motivated by the observation that full-rank Dion (i.e., $r = d$) yields higher-quality updates than Muon, we conduct a speed run experiment on 3B-parameter models trained on the FineWeb dataset. We measure the wall-clock time each optimizer takes to reach a series of target validation losses, and normalize these times relative to AdamW. Timings are measured with 16 NVIDIA H100 GPUs.

The results are shown in Figure 1. Consistent with previous findings [Jordan et al., 2024b, Liu et al., 2025a, Shah et al., 2025], Muon achieves a noticeable speedup over AdamW. Remarkably, full-rank Dion outperforms Muon even in wall-clock time, highlighting the advantage of Dion’s update quality. Note that the smaller state of Muon and Dion optimizers allows larger batch sizes, which is part but not all of the improvement observed here.

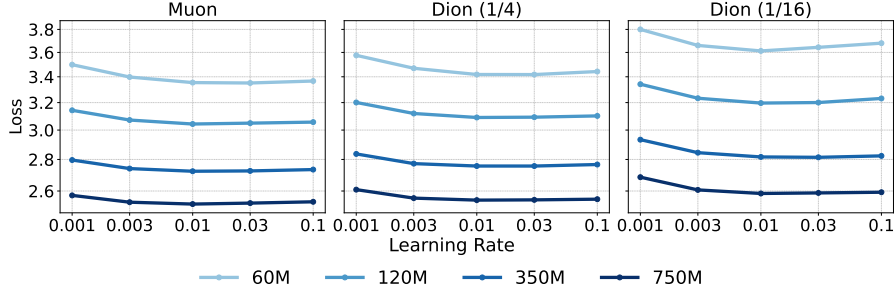


Figure 4: Learning rate versus loss for four model scales. Dion maintains hyperparameter transferability trends of Muon across both model size and different rank fractions.

5.5 Hyperparameter Transfer

Previous work [Bernstein, 2025, Bernstein and Newhouse, 2024a, Pethick et al., 2025] has shown that Muon can exhibit learning rate transfer across model size via shape-dependent scale factors (Section 3.2). To empirically verify that Dion inherits this property, we sweep learning rates for Muon and Dion with rank fractions 1/4 and 1/16 across four model sizes, each trained for a Chinchilla-optimal number of tokens [Hoffmann et al., 2022]. As shown in Figure 4, optimal learning rates are approximately identical for all model sizes. Detailed hyperparameters are given in Section J.5.

6 Related Work and Conclusion

Prior work on preconditioned optimizer update rules includes Shampoo [Gupta et al., 2018] and SOAP [Vyas et al., 2025]. These second-order optimization algorithms have substantial compute and memory overheads, motivating the development of distributed [Anil et al., 2020, Shi et al., 2023] and quantized [Wang et al., 2024] implementations of Shampoo. However, distributed Shampoo and SOAP are outperformed by Muon both on a per-step and wall-clock basis [Jordan et al., 2024b], suggesting that Muon’s orthonormalizing update rule is algorithmically and computationally superior. In contrast, full-rank Dion can outperform Muon, particularly at larger batch sizes. Recent work such as COSMOS [Liu et al., 2025b] has shown promising results from combining SOAP- and Muon-style updates across different eigenspaces.

Prior approaches to reducing communication overhead include gradient sparsification [Wang et al., 2023] and federated averaging [McMahan et al., 2017, Douillard et al., 2023]. We consider these techniques as complementary to Dion. Dion aims to be an efficient optimizer in itself, and it can be combined with these techniques to further lower communication.

Dion takes inspiration from DeMo [Peng et al., 2024], which uses decoupled momentum and lowers communication requirements through lossy compression based on the discrete cosine transform. DeMo is quite effective when very aggressive compression is necessary, but it underperformed Dion at more modest levels of compression. Unlike DeMo, Dion’s decoupled momentum also guarantees exact equivalence to synchronous momentum.

Dion also connects to recent studies on low-rank updates [Cosson et al., 2023, Jadbabaie et al., 2023] and memory-efficient optimizers like GaLore [Zhao et al., 2024]. In contrast to observations from Song et al. [2025], the results here suggest that low-rank training can succeed when paired with an effective error feedback rule.

Lastly, we highlight a few directions for future work:

- **Quantization:** Dion’s optimizer states may be quantizable to lower precision formats to reduce memory use. The column-normalized Q matrix may be particularly quantization-friendly. In addition, expensive steps like QR decomposition may also be faster in reduced-precision arithmetic.
- **Error feedback:** Refining the error feedback rule may improve convergence at lower ranks. A variant explored in Appendix K shows promising results in this regime.
- **Beyond LLMs:** While experiments here focus on GPT-style models, any architecture with matrix-shaped parameters and dense activation vectors may benefit from orthonormalized updates.

Acknowledgments

We thank Shital Shah and Zheng Zhan for their helpful feedback. We are also grateful to Laker Newhouse and Jeremy Bernstein for their many constructive suggestions on the manuscript.

References

- Alekh Agarwal, Olivier Chapelle, Miroslav Dudík, and John Langford. A reliable effective terascale linear learning system. *The Journal of Machine Learning Research*, 15(1):1111–1133, 2014.
- Rohan Anil, Vineet Gupta, Tomer Koren, Kevin Regan, and Yoram Singer. Scalable second order optimization for deep learning. *arXiv preprint arXiv:2002.09018*, 2020.
- Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016. URL <https://arxiv.org/abs/1607.06450>.
- Jeremy Bernstein. Deriving Muon, 2025. URL <https://jeremybernste.in/writing/deriving-muon>.
- Jeremy Bernstein and Laker Newhouse. Modular duality in deep learning. *arXiv preprint arXiv:2410.21265*, 2024a.
- Jeremy Bernstein and Laker Newhouse. Old optimizer, new norm: An anthology. *arXiv preprint arXiv:2409.20325*, 2024b.
- Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiusi Du, Zhe Fu, et al. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954*, 2024.
- Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, Yifeng Lu, et al. Symbolic discovery of optimization algorithms. *Advances in neural information processing systems*, 36:49205–49233, 2023.
- Romain Cosson, Ali Jadbabaie, Anuran Makur, Amirhossein Reisizadeh, and Devavrat Shah. Low-rank gradient descent. *IEEE Open Journal of Control Systems*, 2:380–395, 2023.
- Arthur Douillard, Qixuan Feng, Andrei A Rusu, Rachita Chhaparia, Yani Donchev, Adhiguna Kuncoro, Marc’Aurelio Ranzato, Arthur Szlam, and Jiajun Shen. Diloco: Distributed low-communication training of language models. *arXiv preprint arXiv:2311.08105*, 2023.
- Ethan N. Epperly. Neat randomized algorithms: Randomized Cholesky QR. *Blog post*, 2024. URL <https://www.ethanepperly.com/index.php/2024/06/25/neat-randomized-algorithms-randomized-cholesky-qr/>.
- Essential-AI. Layer Sharding for Large-Scale Training with Muon, 2025. URL <https://www.essential.ai/blog/infra>.
- Yuwei Fan, Yixiao Guo, and Ting Lin. A novel randomized xr-based preconditioned choleskyqr algorithm. *arXiv preprint arXiv:2111.11148*, 2021.
- 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. *arXiv preprint arXiv:2407.21783*, 2024.
- Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. In *International Conference on Machine Learning*, pages 1842–1850. PMLR, 2018.
- Nick Higham. The big six matrix factorizations, 2022. URL <https://nhigham.com/2022/05/18/the-big-six-matrix-factorizations/>.
- Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

- Ali Jadbabaie, Anuran Makur, and Amirhossein Reisizadeh. Adaptive low-rank gradient descent. In *2023 62nd IEEE Conference on Decision and Control (CDC)*, pages 3315–3320. IEEE, 2023.
- Keller Jordan, Jeremy Bernstein, Brendan Rappazzo, @fernbear.bsky.social, Boza Vlado, You Jiacheng, Franz Cesista, Braden Koszarsky, and @Grad62304977. modded-nanogpt: Speedrunning the nanogpt baseline, 2024a. URL <https://github.com/KellerJordan/modded-nanogpt>.
- Keller Jordan, Yuchen Jin, Vlado Boza, You Jiacheng, Franz Cecista, Laker Newhouse, and Jeremy Bernstein. Muon: An optimizer for hidden layers in neural networks, 2024b. URL <https://kellerjordan.github.io/posts/muon/>.
- Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- Tim Large, Yang Liu, Minyoung Huh, Hyojin Bahng, Phillip Isola, and Jeremy Bernstein. Scalable optimization in the modular norm. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL <https://openreview.net/forum?id=SFxAjB7UXx>.
- Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. PyTorch distributed: experiences on accelerating data parallel training. *Proceedings of the VLDB Endowment*, 13(12):3005–3018, 2020.
- Jingyuan Liu, Jianlin Su, Xingcheng Yao, Zhejun Jiang, Guokun Lai, Yulun Du, Yidao Qin, Weixin Xu, Enzhe Lu, Junjie Yan, et al. Muon is Scalable for LLM Training. *arXiv preprint arXiv:2502.16982*, 2025a.
- Liming Liu, Zhenghao Xu, Zixuan Zhang, Hao Kang, Zichong Li, Chen Liang, Weizhu Chen, and Tuo Zhao. COSMOS: a hybrid adaptive optimizer for memory-efficient training of LLMs. *arXiv preprint arXiv:2502.17410*, 2025b.
- Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019. URL <https://openreview.net/forum?id=Bkg6RiCqY7>.
- 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>.
- Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, pages 1273–1282. PMLR, 2017.
- Maksim Melnichenko, Oleg Balabanov, Riley Murray, James Demmel, Michael W. Mahoney, and Piotr Łuszczek. CholeskyQR with randomization and pivoting for tall matrices (CQRPT), 2025. URL <https://arxiv.org/abs/2311.08316>.
- Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using Megatron-LM. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 1–15, 2021.
- Laker Newhouse, Dakota Goldberg, and Ricardo Ruiz. Faster symmetric matrix multiplication with ThunderKittens, 2024. URL <https://www.lakernewhouse.com/assets/writing/faster-symmml-with-thunderkittens.pdf>.
- Guilherme Penedo, Hynek Kydlíček, Loubna Ben allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. The FineWeb Datasets: decanting the web for the finest text data at scale. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024. URL <https://openreview.net/forum?id=n6SCkn2QaG>.
- Bowen Peng, Jeffrey Quesnelle, and Diederik P Kingma. Decoupled momentum optimization. *arXiv preprint arXiv:2411.19870*, 2024.

- Thomas Pethick, Wanyun Xie, Kimon Antonakopoulos, Zhenyu Zhu, Antonio Silveti-Falls, and Volkan Cevher. Training deep learning models with norm-constrained lmos. *arXiv preprint arXiv:2502.07529*, 2025.
- Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.
- Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.
- Ishaan Shah, Anthony M Polloreno, Karl Stratos, Philip Monk, Adarsh Chaluvvaraju, Andrew Hojel, Andrew Ma, Anil Thomas, Ashish Tanwer, Darsh J Shah, et al. Practical efficiency of Muon for pretraining. *arXiv preprint arXiv:2505.02222*, 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. *arXiv preprint arXiv:2309.06497*, 2023.
- Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.
- David R. So, Wojciech Mańke, Hanxiao Liu, Zihang Dai, Noam Shazeer, and Quoc V. Le. Primer: Searching for efficient transformers for language modeling, 2022. URL <https://arxiv.org/abs/2109.08668>.
- Minhak Song, Kwangjun Ahn, and Chulhee Yun. Does SGD really happen in tiny subspaces? In *The Fourteenth International Conference on Learning Representations*, 2025. URL <https://openreview.net/forum?id=v6iLQBoIJw>.
- Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023. URL <https://arxiv.org/abs/2104.09864>.
- Thijs Vogels, Sai Praneeth Karimireddy, and Martin Jaggi. PowerSGD: Practical low-rank gradient compression for distributed optimization. *Advances in Neural Information Processing Systems*, 32, 2019.
- 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 *The Thirteenth International Conference on Learning Representations*, 2025. URL <https://openreview.net/forum?id=IDxZhXrpNf>.
- Jue Wang, Yucheng Lu, Binhang Yuan, Beidi Chen, Percy Liang, Christopher De Sa, Christopher Re, and Ce Zhang. Cocktailsd: Fine-tuning foundation models over 500mbps networks. In *International Conference on Machine Learning*, pages 36058–36076. PMLR, 2023.
- Sike Wang, Pan Zhou, Jia Li, and Hua Huang. 4-bit shampoo for memory-efficient network training. *Advances in Neural Information Processing Systems*, 37:126997–127029, 2024.
- Greg Yang, James B Simon, and Jeremy Bernstein. A spectral condition for feature learning. *arXiv preprint arXiv:2310.17813*, 2023.
- Biao Zhang and Rico Sennrich. Root mean square layer normalization. *Advances in Neural Information Processing Systems*, 32, 2019.
- Hanlin Zhang, Depen Morwani, Nikhil Vyas, Jingfeng Wu, Difan Zou, Udaya Ghai, Dean Foster, and Sham M. Kakade. How does critical batch size scale in pre-training? In *The Thirteenth International Conference on Learning Representations*, 2025. URL <https://openreview.net/forum?id=JCiF03qnmI>.

- Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. Galore: Memory-efficient llm training by gradient low-rank projection. In *International Conference on Machine Learning*, pages 61121–61143. PMLR, 2024.
- 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.

Appendix

A	Is Muon Scalable for LLM Training?	15
B	Ablation Studies	15
B.1	Single Power Iteration vs. Full SVD	15
B.2	Importance of Error Feedback	16
C	Critical Batch Size	16
D	Update Rules for Non-Matrix Parameters	17
D.1	Normalization and Scaling for Learning Rate Transfer	17
D.2	Scaling Factor for Unembedding Parameters	18
D.3	Scalar Optimizers: Adam and Lion	19
D.4	Against Standard Practice	19
E	Distributed Dion Variants	20
F	Equivalence of Distributed and Centralized Dion	20
G	Equivalence of Distributed and Centralized Orthogonalization	22
H	Details on Computational Complexity	23
H.1	Dion	23
H.2	Muon	24
I	Details on Communication Requirements	24
J	Hyperparameter Tuning	25
J.1	Results in Section 5.1	25
J.2	Results in Section 5.2	25
J.3	Results in Section 5.3	26
J.4	Results in Section 5.4	26
J.5	Results in Section 5.5	26
K	Modifications for Extreme Sparsity	27

A Is Muon Scalable for LLM Training?

Moonshot AI [Liu et al., 2025a] successfully applied Muon for training a 16B parameter language model. In this section, we estimate if Muon continues to scale for larger models.

Let us revisit the flagship **Llama 3 405B** training run [Grattafiori et al., 2024], which processes 15.6T tokens in batches of 16M tokens—about 10^6 optimization steps in total. The public report lists the following parallelism recipe:

Tensor parallel (TP)	8
Context parallel (CP)	1
Pipeline parallel (PP)	16
Fully-sharded data parallel (FS)	128
Total GPUs	16384

Suppose we use the implementation of Liu et al. [2025a] that runs Newton-Schulz iterations *locally*. Consequently, all 8×128 GPUs in a $TP \times FS$ grid perform identical work, and only the 16 pipeline stages process disjoint parameters.

- (a) **One MLP weight matrix.** Each MLP matrix weighs in at 53248×16384 , for 8.7×10^8 parameters. For this matrix size, a single H100 GPU is benchmarked as taking roughly 1 second to run five Newton-Schulz iterations in bf16 precision.
- (b) **Per pipeline stage.** Every transformer block contains three such matrices. With 126 blocks total, each PP rank processes

$$126 \times 3/16 \approx 24$$

MLP matrices, costing around 24 seconds per optimizer step.

- (c) **End-to-end cost.** $24 \text{ s} \times 10^6 \text{ steps} = 2.4 \times 10^7 \text{ s} \approx \mathbf{278 \text{ days}}$ (!) of *extra compute* to orthogonalize MLP matrices alone.

Even in the absence of any communication bottlenecks, Muon is therefore *compute-bound* at 405B scale. In practice the situation is worse: attention matrices still need to be orthogonalized, and any time spent on Newton-Schulz is purely in addition to the forward/backward pass.

This result challenges the claim in Jordan et al. [2024b] of a 0.5% overhead for using Muon on a Llama 405B scale training run. The theoretical 0.5% figure assumes no replicated work, yet the current real-world implementation of distributed Muon [Liu et al., 2025a] cannot avoid redundant computation across the $TP \times FS$ dimensions. Consequently, we see a $0.5\% \times 8 \times 128 \approx 500\%$ overhead in wall-clock time.

We note that it is theoretically possible to distribute the compute of Muon more efficiently and reduce the amount of redundant computation. More advanced parallelism strategies have been proposed,² but to the best of our knowledge such ideas have yet to be validated in any existing implementation of Muon. Further distributing Muon’s compute would lead to a substantially increased communication burden, which may limit the potential room for improvement.

B Ablation Studies

We conduct ablation studies to evaluate two core components of Dion: the error feedback mechanism and the use of a single power iteration for low-rank approximation. For these ablation studies, we train 120M parameter models and set the batch size to be $2048 \cdot 1024 \approx 2.1 \text{ M}$ tokens.

B.1 Single Power Iteration vs. Full SVD

We evaluate the effectiveness of Dion’s single-step power iteration for computing a rank- r approximation. In this experiment, we compare Dion to an alternative that computes the singular value decomposition (SVD) at every step and truncates to the top- r singular values—an ideal but computationally expensive alternative.

²For example, see <https://main-horse.github.io/posts/parallelizing-muon/>

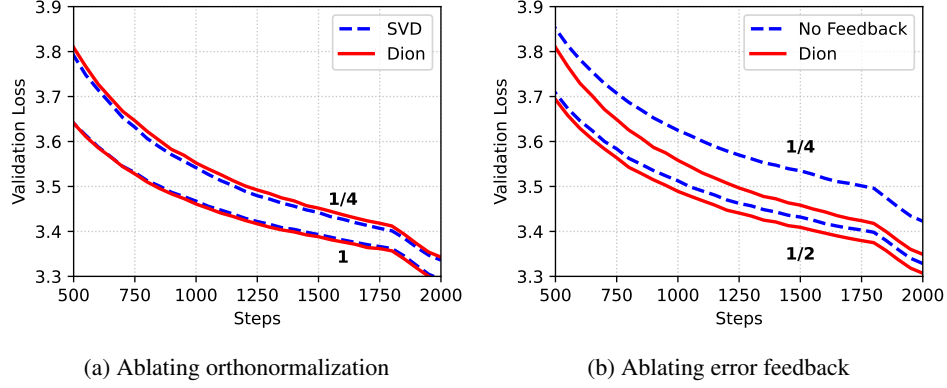


Figure 5: **Testing algorithmic components.** We evaluate two key design choices in Dion: the error feedback mechanism and the use of a single step of power iteration. Error feedback proves crucial at lower ranks. A single power iteration performs on par with SVD, offering a more efficient alternative without sacrificing performance.

The results, shown in Figure 5a, reveal negligible differences in convergence behavior between the two approaches. This suggests that using power iteration—initialized from the previous iteration’s right orthonormal basis [Vogels et al., 2019]—can give a sufficiently accurate approximation at a fraction of the computational cost.

B.2 Importance of Error Feedback

To evaluate the necessity of the error feedback mechanism, we compare Dion against a simplified variant that omits error feedback. In this baseline, the rank- r approximation is applied directly to the momentum buffer:

$$M_t \leftarrow \mu M_{t-1} + G_t, \quad \text{for } \mu \in (0, 1),$$

whereas Dion first compresses an auxiliary buffer B_t and then updates the momentum using error feedback:

$$M_t \leftarrow B_t - (1 - \mu)P_t R_t^\top.$$

As shown in Figure 5b, the version without error feedback suffers from steep performance degradation as the rank decreases from $r = d/2$ to $r = d/4$, while Dion maintains stable performance. This highlights the importance of incorporating error feedback to preserve optimization quality when using low-rank approximation.

C Critical Batch Size

We follow the critical batch size evaluation protocol of Zhang et al. [2025] with one key modification: instead of applying an exponential moving average, we use a constant learning rate for each optimizer and relax the target loss threshold to ensure a fair comparison across optimizers. All runs here use 120M parameter models trained on the FineWeb dataset until reaching a validation loss of 3.4.

For AdamW, we perform a hyperparameter sweep over learning rates $\{0.001, 0.003, 0.01\}$, weight decay values $\{0, 0.01\}$, and $\beta_2 \in \{0.95, 0.99\}$ for each batch size. We use a trapezoidal learning rate schedule with 10% warmup and 10% cooldown, and fix $\beta_1 = 0.9$. For Muon and Dion, we fix the momentum parameter $\mu = 0.95$ and sweep learning rates over $\{0.003, 0.01, 0.03\}$.

We evaluate Dion’s critical batch size across a range of rank fractions r/d . Figure 6 shows the results. As expected, Dion’s performance degrades as the rank r decreases. Nevertheless, Dion remains competitive even at low ranks—for example, Dion with $r = d/8$ is generally on par with AdamW.

Interestingly, Dion seems to exhibit a larger critical batch size than Muon. The gap between full-rank Dion and Muon starts at 2^{11} and becomes especially evident at 2^{13} . At batch size 2^{14} , even Dion with $r = d/2$ begins to outperform Muon’s full-rank orthonormalization. We hypothesize that this discrepancy arises from Muon’s use of Newton-Schulz iteration to approximately orthonormalize the weight update, which may yield less accurate results than Dion’s orthonormalization procedure.

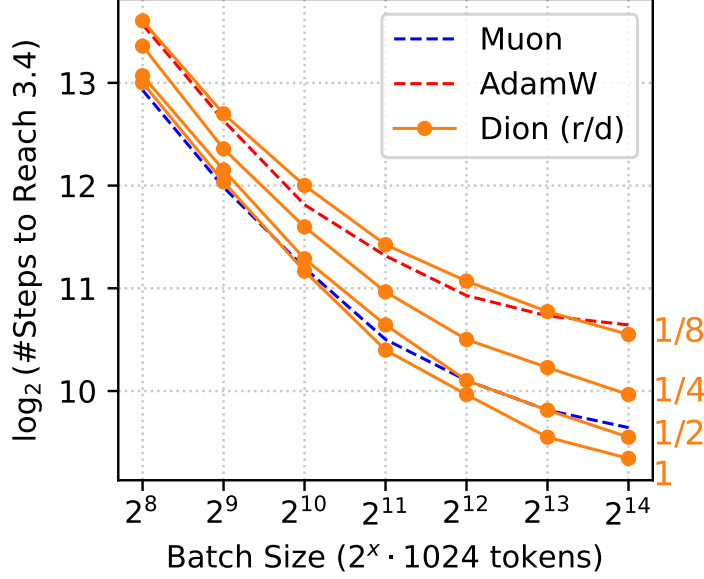


Figure 6: Dion vs. Muon and AdamW across batch sizes 2^8 – 2^{14} tokens. Dion at low rank is competitive with AdamW, while full rank Dion outperforms Muon across larger batch sizes. For all rank fractions r/d evaluated here, Dion’s critical batch size appears to be no less than that of Muon and greater than that of AdamW.

D Update Rules for Non-Matrix Parameters

We provide further details here on the use of Adam and Lion for scalar parameter updates, along with a more thorough explanation of the normalization and scaling factors given in Table 1. The objective here is to maintain consistent learning rate transfer across model sizes for both matrix and scalar parameter updates—any model size, two optimizers, one learning rate.

D.1 Normalization and Scaling for Learning Rate Transfer

For effective learning rate transfer, Yang et al. [2023] advocate that both initial parameters and their updates should maintain a $\Theta(1)$ *natural norm*. This norm is defined as the RMS norm for dense vectors (e.g., activations), the ℓ_2 norm for sparse vectors (e.g., one-hot encodings), and the respective operator norm for matrices. For reference, the RMS norm of a vector $\mathbf{v} \in \mathbb{R}^d$ is defined as

$$\|\mathbf{v}\|_{\text{RMS}} := \frac{1}{\sqrt{d}} \|\mathbf{v}\|_2 = \sqrt{\frac{v_1^2 + v_2^2 + \dots + v_d^2}{d}}.$$

The approach of maintaining a $\Theta(1)$ natural norm with Dion relies on two components: parameter update normalization and learning rate scaling. Normalization is achieved by the optimizer update itself. Dion (and Muon) compute orthonormal matrix updates, Adam produces an approximately constant scalar RMS norm, and Lion produces a true constant scalar RMS norm. Starting from a single base learning rate, we then apply scaling factors depending on parameter types and dimensions. The same base learning rate should then transfer across model sizes.

We analyze the normalization and scaling factors required for the various parameter types in Table 1 in order to produce a unit natural norm.

- **Weight matrix:** A typical $d_{\text{out}} \times d_{\text{in}}$ weight matrix maps dense activation vectors $\mathbb{R}^{d_{\text{in}}}$ to dense vectors $\mathbb{R}^{d_{\text{out}}}$. If the input vector has RMS norm 1, we desire that the output vector also have RMS norm at most 1—that is, the weight update should have a unit $\text{RMS} \rightarrow \text{RMS}$ operator norm. An orthonormal matrix has unit spectral norm ($\ell_2 \rightarrow \ell_2$ operator norm). Applying a scale factor of $\sqrt{d_{\text{out}}/d_{\text{in}}}$ achieves the desired RMS operator norm, as shown by Bernstein [2025].

- **Bias vector:** A bias vector can be treated as a $d_{\text{out}} \times 1$ matrix that maps a constant $[1]$ to the vector itself. It suffices to normalize the parameter update to unit RMS norm.
- **Embedding:** An embedding layer maps a one-hot vector to a dense vector. The embedding matrix can be viewed as a set of independent $d_{\text{out}} \times 1$ vectors, each of which can be treated identically as a bias vector, and we normalize the embedding parameter update to unit RMS norm.
- **Unembedding:** We view the unembedding matrix as a set of independent $1 \times d_{\text{in}}$ vectors, each of which maps an input activation vector to a scalar logit. The expected change in the output logit is proportional to the ℓ_2 norm of the unembedding vector (see [Section D.2](#)). Given a parameter update with unit RMS norm, we then scale by $1/\sqrt{d_{\text{in}}}$.
- **Normalization:** Multiplicative and additive factors may be used in normalization layers, such as LayerNorm [[Ba et al., 2016](#)] and RMSNorm [[Zhang and Sennrich, 2019](#)]. As each factor is an independent learned constant, we may view them as 1×1 weight matrices and 1×1 bias vectors. The spectral, RMS, and ℓ_2 norms are identical in this trivial case, and all scaling factors are 1.

D.2 Scaling Factor for Unembedding Parameters

Suppose we have an unembedding vector \mathbf{v} and an activation vector \mathbf{h} , both with dimension d . The output is the scalar logit $s = \mathbf{v} \cdot \mathbf{h} = v_1 h_1 + v_2 h_2 + \dots + v_d h_d$. If we assume that weight updates $\Delta \mathbf{v}$ are independently distributed and centered around zero, which is very likely true for sign-based optimizers such as Lion, we then have

$$\begin{aligned}
\mathbb{E}[(\Delta s)^2] &= \mathbb{E}[(\Delta \mathbf{v} \cdot \mathbf{h})^2] \\
&= \mathbb{E}[(\Delta v_0 h_0 + \Delta v_1 h_1 + \dots + \Delta v_d h_d)^2] \\
&= \sum_{i \in [1..d]} \mathbb{E}[(\Delta v_i)^2] \mathbb{E}[h_i^2] + \sum_{\substack{i, j \in [1..d] \\ i \neq j}} \mathbb{E}[\Delta v_i] \mathbb{E}[\Delta v_j] \mathbb{E}[h_i] \mathbb{E}[h_j] \\
&= d \cdot \|\Delta \mathbf{v}\|_{\text{RMS}}^2 \|\mathbf{h}\|_{\text{RMS}}^2 + 0.
\end{aligned}$$

Given an input activation vector with $\|\mathbf{h}\|_{\text{RMS}} = 1$,

$$\mathbb{E}[\|\Delta s\|_{\text{RMS}}] = \sqrt{d} \cdot \|\Delta \mathbf{v}\|_{\text{RMS}}.$$

For an expected unit change in output, we can normalize the parameter update to $\|\Delta \mathbf{v}\|_{\text{RMS}} = 1$ and scale by $1/\sqrt{d}$. This is equivalent to normalizing to $\|\Delta \mathbf{v}\|_2 = 1$.

The RMS scaling factor of $1/\sqrt{d}$ differs from the $1/d$ given by [Yang et al. \[2023\]](#). The smaller scaling factor of $1/d$ produces a worst-case $\|\Delta s\|_{\text{RMS}} = O(1)$, which holds even when $\mathbb{E}[\Delta \mathbf{v}] \neq 0$. Scaling by $1/\sqrt{d}$ produces $\|\Delta s\|_{\text{RMS}} = \Theta(1)$ in expectation.

In [Figure 7](#), we empirically test three different scale factors for the unembedding layer— $1/\sqrt{d}$, $1/d$, and 1—along with a manually tuned Adam learning rate. For the three candidate scale factors, we use Lion as the optimizer because of its constant RMS norm. For layers besides the unembedding layer, we use the same scaling factors given in [Table 1](#). For Adam, we use the same learning rate for all non-matrix parameters, following the conventional practice of using a single Adam learning rate.

Using Lion with a $1/\sqrt{d}$ unembedding scale factor consistently results in the lowest validation loss. It outperforms the manually tuned Adam learning rate without needing any additional hyperparameter search. The alternative scale factors all result in poorer performance relative to Adam. Furthermore, using the unscaled base learning rate (scale factor 1) for the unembedding layer led to training instability. We observed much higher gradient norms and large loss spikes at the beginning of training, suggesting that the learning rate was far too high.

Additional hyperparameter details: All experiments here use 120M parameter models trained on the FineWeb dataset. The batch size is 1024 and the sequence length is 1024. The total training steps is 3000, and a linear learning rate decay to zero is applied over the last 20% of steps. The base learning rate for Dion, Lion, and Muon is set to 0.01. The best Adam learning rate was found to be 0.002. For Dion and Muon, we use $\mu = 0.95$. Adam uses $(\beta_1, \beta_2) = (0.9, 0.95)$ and Lion uses $(\beta_1, \beta_2) = (0.95, 0.98)$. All runs use a weight decay of 0.01 for matrix parameters and 0 for non-matrix parameters.

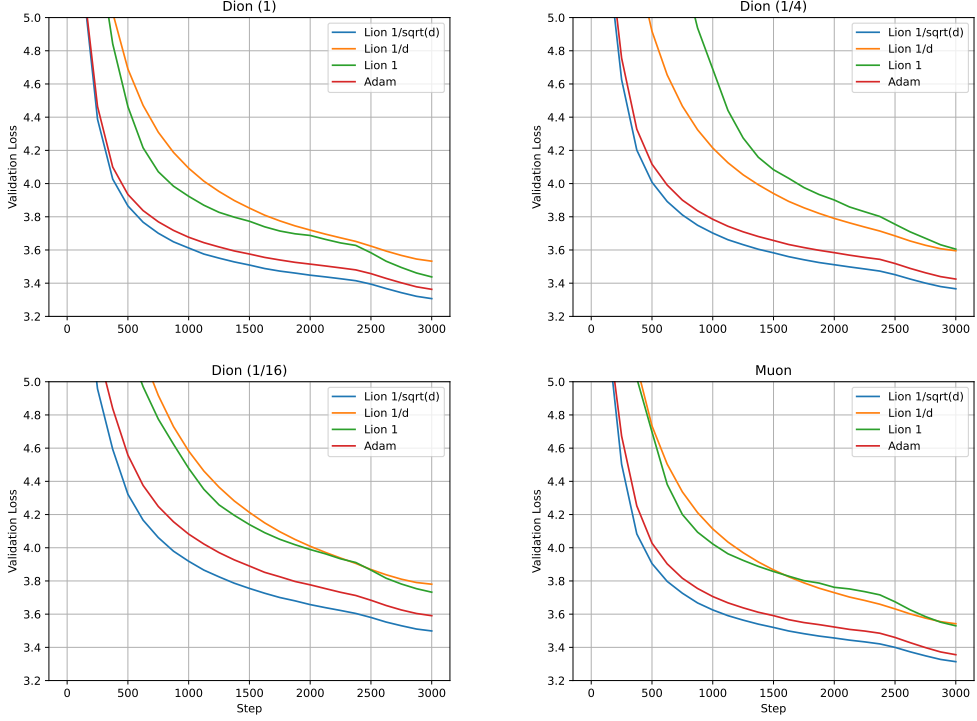


Figure 7: Comparison of unembedding layer optimizer and learning rate scale factors. When using Lion as the scalar optimizer, we experiment with various unembedding layer learning rates, determined by scaling the base learning rate by $1/\sqrt{d}$, $1/d$, and 1 (unscaled). The Adam baseline uses a fixed learning rate for all non-matrix parameters. Using Lion with a $1/\sqrt{d}$ unembedding scale factor consistently gives the best results, which holds across varying Dion rank fractions 1, 1/4, and 1/16, and further generalizes to Muon.

D.3 Scalar Optimizers: Adam and Lion

Adam [Kingma and Ba, 2014] and AdamW [Loshchilov and Hutter, 2019] are highly popular element-wise optimizer algorithms. In previous works on Muon, Jordan et al. [2024a] and Liu et al. [2025a] respectively applied Adam and AdamW for scalar parameter updates. Experimental results show that Adam can perform well when used in conjunction with Dion, but its drawback is that the optimal Adam learning rate does not innately match that of Dion or Muon. Obtaining good performance with Adam may require manually tuning a separate learning rate.

Consequently, we recommend the use of Lion [Chen et al., 2023] instead, along with the per-parameter scale factors given in Table 1. Lion computes its update using the sign function (+1 or -1), thus guaranteeing that the parameter update has a constant RMS norm of one. As shown in Figure 7, the Dion+Lion combination can outperform Dion+Adam while sharing a single base learning rate for both optimizers. We find it likely that Muon is also more compatible with Lion than Adam.

D.4 Against Standard Practice

These normalization and scaling guidelines suggest that the common practice of using a single Adam learning rate for all parameters is highly suboptimal. Looking at merely the embedding and unembedding layers of a typical LLM, their ideal RMS update magnitudes differ by a factor of $\sqrt{d_{\text{model}}}$. This factor can exceed 100 for a large model! The learning rate must then be made smaller in order to tolerate this mismatch, leading to slower convergence. We speculate that the practice of weight-tying the embedding and unembedding layers is likely also detrimental, although it could help compensate for the difference in ideal learning rates.

Indeed, we observe that the NanoGPT speedrun by Jordan et al. [2024a] uses untied weights and a much higher learning rate for embedding parameters than unembedding, with learning rates manually

tuned for each model size. The learning rate ratio does not exactly match $\sqrt{d_{\text{model}}}$, but falls roughly within an order of magnitude. As a further caveat, the NanoGPT speedrun uses Adam to optimize these parameters, which does not guarantee a constant RMS norm.

d_{model}	Embedding LR	Unembedding LR	LR Ratio	$\sqrt{d_{\text{model}}}$
768	0.6	0.22	2.7	27.7
1024	0.3	1/320	96	32

Table 3: NanoGPT speedrun hyperparameters for the *small* and *medium* GPT-2 model configurations.

E Distributed Dion Variants

There exists four variants of 3D parallel Dion algorithm (Algorithm 3), produced by transposing the matrix in power iteration and by swapping the FS and TP parallelism dimensions. The two variants with transposed power iteration are given in Algorithm 3 and Algorithm 4. The remaining two variants can be created simply by replacing FS with TP and vice versa.

As for the best variant to use for any given parameter matrix, we offer the following advice:

- Generally, the TP axis is only applied over a fast intra-node communication network (e.g. NVLink), while the FS axis may use a slower scale-out network (e.g. Infiniband or Ethernet). The sharding dimension used for DISTRIBUTED-ORTHOGONALIZE is more communication-heavy, which we suggest placing on the TP axis. This is the case for both Algorithm 3 and Algorithm 4.
- For any given parameter matrix, the TP sharding dimension typically cannot be freely chosen. It is determined by a layer’s position relative to the activation function, in order to permit the activation to be applied element-wise without additional communication. In language models, attention QKV and MLP up-projection matrices are sharded column-wise, while attention output and MLP down-projection matrices are sharded row-wise. In contrast to TP, the FS sharding dimension can be arbitrarily chosen, as parameters are always unsharded before any computation is performed. We recommend ensuring that the TP and FS sharding dimensions are different.
- In PyTorch, a `nn.Linear` layer with `ColwiseParallel` tensor parallelism has its weight matrix sharded with `Shard(0)`, and a `RowwiseParallel` layer has its weight matrix sharded with `Shard(1)`. On the other hand, PyTorch’s `FSDP2` `fully_shard` API by default always uses `Shard(0)`, but allows specifying a custom `shard_placement_fn` to return arbitrary per-parameter shardings. Therefore, when combining tensor parallelism with FSDP2, one should use a custom placement function that returns `Shard(1)` for parameters with column-wise tensor parallelism and `Shard(0)` for parameters with row-wise tensor parallelism.
- The “standard” Dion (Algorithm 3) uses the case of a $m \times n$ matrix with TP-sharded m and FS-sharded n . The “transposed” Dion (Algorithm 4) uses FS-sharded m and TP-sharded n . Therefore, use Algorithm 3 for `ColwiseParallel` parameters, and use Algorithm 4 for `RowwiseParallel` parameters.

F Equivalence of Distributed and Centralized Dion

Theorem F.1. *Let X_t be the $m \times n$ iterate produced by centralized Dion (Algorithm 1), and let $\hat{X}_t^{(i,j)}$ be the per-device shards in 3D-parallel Dion (Algorithm 3) whose weights are FS sharded over index i and TP sharded over index j . Then, for all t ,*

$$X_t = [\hat{X}_t^{(i,j)}]_{i,j} \quad (\text{i.e. concatenating the FS and TP shards recovers the full matrix}).$$

Moreover, the local momentum shards $\hat{M}_t^{(i,j)}$ average to the centralized M_t , and the low-rank factors produced by the distributed power iteration and column-normalization steps are exactly the corresponding slices of the centralized P_t, Q_t .

Algorithm 4 Transposed Dion

Require: Learning rate η , momentum decay μ , rank r

```

1: for  $t = 1$  to  $T$  do
2:   Compute local gradient  $\hat{G}_t^{(i,j)} \in \mathbb{R}^{m_i \times n_j}$ 
3:    $Q_{t-1}^{(i)} \leftarrow \text{ALLGATHER}_{\text{TP}}(Q_{t-1}^{(i,j)}) \in \mathbb{R}^{m_i \times r}$  ▷ unshard  $Q$  along TP dimension
4:    $\hat{B}_t^{(i,j)} \leftarrow \hat{M}_{t-1}^{(i,j)} + \hat{G}_t^{(i,j)} \in \mathbb{R}^{m_i \times n_j}$  ▷ accumulate new gradient
5:    $P_t^{(j)}, R_t^{(i)} \leftarrow \text{DISTRIBUTED-POWERITER1-T}(\hat{B}_t^{(i,j)}; Q_{t-1}^{(i)})$ 
6:    $\hat{M}_t^{(i,j)} \leftarrow \hat{B}_t^{(i,j)} - (1 - \mu) R_t^{(i)} (P_t^{(j)})^\top \in \mathbb{R}^{m_i \times n_j}$  ▷ error feedback
7:    $Q_t^{(i)} \leftarrow \text{DISTRIBUTED-COLUMNNORMALIZE}(R_t^{(i)})$ 
8:    $X_t^{(i,j)} \leftarrow X_{t-1}^{(i,j)} - \eta \sqrt{m/n} Q_t^{(i)} (P_t^{(j)})^\top \in \mathbb{R}^{m_i \times n_j}$  ▷ scaled orthonormal update
9:    $Q_t^{(i,j)} \leftarrow \text{SHARD}_{\text{TP}}(Q_t^{(i)}) \in \mathbb{R}^{m_i \times r_j}$  ▷ reshard  $Q$  along TP dimension
10: end for

1: function  $\text{DISTRIBUTED-POWERITER1-T}(\hat{B}^{(i,j)}, Q^{(i)}) \in (\mathbb{R}^{m_i \times n_j}, \mathbb{R}^{m_i \times r})$ 
2:    $\hat{P}^{(j)} \leftarrow (\hat{B}^{(i,j)})^\top Q^{(i)} \in \mathbb{R}^{n_j \times r}$ 
3:    $P^{(j)} \leftarrow \mathbb{E}_{\text{DP}} \sum_{\text{FS}} (\hat{P}^{(j)}) \in \mathbb{R}^{n_j \times r}$  ▷ FS sharded matrix multiply and DP sync
4:    $P^{(j)} \leftarrow \text{DISTRIBUTED-ORTHOGONALIZE}(P^{(j)})$ 
5:    $\hat{R}^{(i)} \leftarrow \hat{B}^{(i,j)} P^{(j)} \in \mathbb{R}^{m_i \times r}$ 
6:    $R^{(i)} \leftarrow \mathbb{E}_{\text{DP}} \sum_{\text{TP}} (\hat{R}^{(i)}) \in \mathbb{R}^{m_i \times r}$  ▷ TP sharded matrix multiply and DP sync
7:   return  $P^{(j)}, R^{(i)} \in (\mathbb{R}^{n_j \times r}, \mathbb{R}^{m_i \times r})$ 
8: end function

```

Proof. We proceed by induction on t . We use the hat notation to denote the states of the distributed algorithm. At $t = 0$, all states are identically initialized and trivially match. Suppose that at step $t - 1$ we have

$$\forall i, j, \quad X_{t-1}^{(i,j)} = \hat{X}_{t-1}^{(i,j)}, \quad M_{t-1}^{(i,j)} = \mathbb{E}_{\text{DP}}[\hat{M}_{t-1}^{(i,j)}], \quad Q_{t-1}^{(i,j)} = \hat{Q}_{t-1}^{(i,j)}.$$

We show that the same holds at step t .

1. Forming the buffer. Each device computes $\hat{B}_t^{(i,j)} = \hat{M}_{t-1}^{(i,j)} + \hat{G}_t^{(i,j)}$. Since the true gradient and momentum shards sum/average to the centralized G_t and M_{t-1} , it follows that concatenating and averaging these \hat{B} 's recovers the centralized B_t sharded in the same (i, j) pattern.

2. Distributed power iteration + orthonormalization. Before the TP-sharded power iteration, the algorithm all-gathers each FS shard of Q_{t-1} across TP:

$$\hat{Q}_{t-1}^{(i)} \leftarrow \text{ALLGATHER}_{\text{TP}}(\hat{Q}_{t-1}^{(i,j)}),$$

which by the induction hypothesis equals the centralized $Q_{t-1}^{(i)}$.

Then each device forms $\hat{P}^{(j)} = \hat{B}^{(i,j)} Q_{t-1}^{(i)}$ and carries out the FS-sum and DP-mean to reconstruct the full $B_t Q_{t-1}$ in FS shard j . A call to `DISTRIBUTED-ORTHOGONALIZE` is precisely the distributed orthogonalization algorithm (Algorithm 2) and so by Lemma G.1, it produces exactly the centralized P_t sharded over j .

Likewise, the subsequent $\hat{R}^{(i)} = (\hat{B}^{(i,j)})^\top P_t^{(j)}$ followed by TP-sum and DP-mean reconstructs each slice of the centralized R_t , and the distributed column-normalize yields exactly the centralized Q_t on each (i, j) .

3. Parameter and momentum updates. Finally, each shard updates

$$\hat{X}_t^{(i,j)} = \hat{X}_{t-1}^{(i,j)} - \eta P_t^{(j)} (Q_t^{(i)})^\top, \quad \hat{M}_t^{(i,j)} = \hat{B}_t^{(i,j)} - (1 - \mu) P_t^{(j)} (R_t^{(i)})^\top.$$

Concatenating the X -shards recovers

$$X_t = X_{t-1} - \eta P_t Q_t^\top,$$

and averaging the $\hat{M}_t^{(i,j)}$ over DP recovers

$$M_t = B_t - (1 - \mu) P_t R_t^\top,$$

exactly as in the centralized [Algorithm 1](#).

Thus all induction hypotheses hold at step t , and the 3D-parallel iterates remain identical to the centralized ones for every t . \square

G Equivalence of Distributed and Centralized Orthogonalization

In this section, we provide the details of the distributed implementation of randomized Cholesky QR ([Algorithm 2](#)). We begin by reviewing the standard randomized Cholesky QR algorithm ([Algorithm 5](#)), following the presentation of [Epperly \[2024\]](#). A similar method was first introduced by [Fan et al. \[2021\]](#). The default oversampling factor of 1.25 follows the recommendation in [Melnichenko et al. \[2025\]](#), which produced stable results in the experiments with Dion.

Algorithm 5 Randomized-Cholesky-QR(P) $\in \mathbb{R}^{m \times r}$

Require: Oversampling factor $k \geq r$ (default value $\lceil 1.25r \rceil$).

- 1: $S \leftarrow \mathbb{R}^{k \times m} \sim \mathcal{N}(0, 1/\sqrt{k})$ \triangleright generate random sketching matrix
 - 2: $G \leftarrow SP \in \mathbb{R}^{k \times r}$ \triangleright begin 1st iteration using randomized QR
 - 3: $_, R_1 \leftarrow \text{QR}(G) \in \mathbb{R}^{r \times r}$ \triangleright only R component needed
 - 4: $B \leftarrow PR_1^{-1} \in \mathbb{R}^{m \times r}$
 - 5: $H \leftarrow B^\top B \in \mathbb{R}^{r \times r}$ \triangleright begin 2nd iteration using Cholesky QR
 - 6: $R_2 \leftarrow \text{Cholesky}(H) \in \mathbb{R}^{r \times r}$ \triangleright only upper triangular component needed
 - 7: $\bar{P} \leftarrow BR_2^{-1} \in \mathbb{R}^{m \times r}$
 - 8: **return** $\bar{P} \in \mathbb{R}^{m \times r}$
-

The advantage of [Algorithm 2](#) over the above centralized version is that orthogonalization is performed locally on each shard, requiring only synchronization of small $k \times r$ and $r \times r$ matrices during all-reduce operations. As long as $k, r < m, n$, the communication overhead introduced by this procedure is small relative to the size of the model parameters.

Based on the centralized algorithm procedure, we provide a proof of equivalence.

Lemma G.1. *Let $P \in \mathbb{R}^{m \times r}$ be sharded row-wise across s devices as $P^{(j)} \in \mathbb{R}^{m_j \times r}$ with $j \in \{1, 2, \dots, s\}$. Suppose each device holds a corresponding shard $S^{(j)}$ of a shared sketch matrix $S \in \mathbb{R}^{k \times m}$ with $S = [S^{(1)}, \dots, S^{(s)}]$. If the Σ_{TP} all-reduce computes exact sums, then the final output $\{\bar{P}^{(j)}\}$ of the distributed orthogonalization ([Algorithm 2](#)) matches the output of the centralized randomized Cholesky QR ([Algorithm 5](#)).*

Proof of Lemma G.1. In [Algorithm 5](#), we form $G = SP \in \mathbb{R}^{k \times r}$ and then compute its QR factorization to obtain $R_1 \in \mathbb{R}^{r \times r}$. We then compute $B = PR_1^{-1}$, followed by

$$H = B^\top B = (R_1^{-1})^\top P^\top PR_1^{-1}.$$

A Cholesky decomposition of H yields R_2 , and finally

$$Q = BR_2^{-1} = PR_1^{-1}R_2^{-1}.$$

In [Algorithm 2](#), each shard j holds exactly the corresponding block $P^{(j)}$ and the matching submatrix $S^{(j)}$. The local computations result in $G^{(j)} = S^{(j)}P^{(j)}$, and the all-reduce sum $\sum_{\text{TP}}(G^{(j)})$ exactly reconstructs

$$\sum_j S^{(j)}P^{(j)} = SP.$$

Thus the global G as input to QR decomposition is identical in both algorithms, so the computed R_1 is the same. Likewise,

$$B^{(j)} = P^{(j)}R_1^{-1} \implies \sum_j (B^{(j)})^\top B^{(j)} = (R_1^{-1})^\top P^\top PR_1^{-1} = B^\top B,$$

so the sum in the second phase reproduces H , yielding the same Cholesky factor R_2 . Finally, each shard updates

$$\bar{P}^{(j)} \leftarrow B^{(j)} R_2^{-1} = P^{(j)} R_1^{-1} R_2^{-1},$$

and stacking the $\bar{P}^{(j)}$ gives exactly $\bar{P} = P R_1^{-1} R_2^{-1}$ as desired. \square

H Details on Computational Complexity

We provide a more detailed analysis of the number of FLOPs needed by Dion versus Muon for optimizing a single $m \times n$ parameter matrix. For simplicity, we omit all element-wise operations (e.g. updating momentum with gradient), as they do not affect the asymptotic runtime.

The following FLOP counts are assumed:

- **Matrix multiplication:** Multiplying $m \times n$ and $n \times p$ matrices requires $2mnp$ FLOPs.
- **QR decomposition:** For a $m \times n$ matrix with $m \geq n$, the Householder QR algorithm requires $2mn^2 - (2/3)n^3$ FLOPs [Higham, 2022].
- **Cholesky decomposition:** For a $n \times n$ square matrix, Cholesky decomposition requires $n^3/3$ FLOPs [Higham, 2022].
- **Solve triangular:** Solving a linear system involving a $n \times n$ triangular matrix and a vector, using forward- or back-substitution, requires n^2 FLOPs. Therefore, computing AR^{-1} for a $m \times n$ matrix A and $n \times n$ triangular matrix R requires m matrix-vector solves, for a total of mn^2 FLOPs.

H.1 Dion

We break down the per-device FLOPs required by Dion in Algorithm 3. As shorthand, FS and TP denote the respective number of devices along each dimension, and $m \times n$ is the size of the entire unsharded matrix. Equivalently, $m_j = m/\text{TP}$ and $n_i = n/\text{FS}$.

Step	Operation	Shape	FLOPs
Power iteration	$B^{(i,j)} Q^{(i)}$	$m_j \times n_i$ and $n_i \times r$	$2mnr/(\text{FS} \cdot \text{TP})$
	$(B^{(i,j)})^\top P^{(j)}$	$n_i \times m_j$ and $m_j \times r$	$2mnr/(\text{FS} \cdot \text{TP})$
Orthogonalization	$S^{(j)} P^{(j)}$	$1.25r \times m_j$ and $m_j \times r$	$2.5mr^2/\text{TP}$
	$\text{QR}(G)$	$1.25r \times r$	$(11/6)r^3$
	$P^{(j)} R_1^{-1}$	$m_j \times r$ and $r \times r$	mr^2/TP
	$(B^{(j)})^\top B^{(j)}$	$r \times m_j$ and $m_j \times r$	$2mr^2/\text{TP}$
	$\text{Cholesky}(H)$	$r \times r$	$r^3/3$
	$B^{(j)} R_2^{-1}$	$m_j \times r$ and $r \times r$	mr^2/TP
Error feedback	$P^{(j)} (R^{(i)})^\top$	$m_j \times r$ and $r \times n_i$	$2mnr/(\text{FS} \cdot \text{TP})$
Weight update	$P^{(j)} (Q^{(i)})^\top$	$m_j \times r$ and $r \times n_i$	$2mnr/(\text{FS} \cdot \text{TP})$

This gives a total per-device FLOP count of

$$\frac{8mnr}{\text{FS} \cdot \text{TP}} + \frac{13mr^2}{2 \cdot \text{TP}} + \frac{13r^3}{6}.$$

We find it particularly advantageous that the most computationally intensive operations with $O(mnr)$ runtime benefit from sharding across both FS and TP axes. A secondary $O(mr^2)$ component is sharded across TP but replicated across FS. Only the $O(r^3)$ component must be replicated across both FS and TP axes. Therefore, rank reduction not only lowers communication but also offers substantial computation speedup. A linear decrease in r leads to a cubic decrease in redundant work performed.

It is worth recalling the Dion variants discussed in Appendix E. If the FS and TP axes are swapped, the $O(mr^2)$ operations are sharded across FS instead of TP. Although this swapping results in a greater communication cost over the typically slower FS sharding axis, the tradeoff may be computationally favorable in a scenario where $\text{FS} \gg \text{TP}$, such as in Appendix A.

H.2 Muon

Each Newton-Schulz iteration requires three matrix multiplications. For a $m \times n$ matrix with $m \geq n$, a single iteration takes $2mn^2 + 2n^3 + 2mn^2$ FLOPs [Jordan et al., 2024b]. Using the default five iterations requires a total of $20mn^2 + 10n^3$ FLOPs.³

Remarkably, Dion’s FLOP count remains smaller than Muon’s—even in the “worst-case” scenario with neither sparsity nor sharding. Letting $r = n$ and $\text{FS} = \text{TP} = 1$, we have

$$\frac{8mnr}{\text{FS} \cdot \text{TP}} + \frac{13mr^2}{2 \cdot \text{TP}} + \frac{13r^3}{6} = \frac{29}{2}mn^2 + \frac{13}{6}n^3 \approx 14.5mn^2 + 2.17n^3.$$

Hence, it follows that

$$\underbrace{14.5mn^2 + 2.17n^3}_{\text{maximum Dion FLOPs}} < \underbrace{20mn^2 + 10n^3}_{\text{Muon FLOPs}}.$$

Muon is surprisingly compute-intensive, especially considering its strong empirical performance in the NanoGPT speedrun. Consider that the singular value decomposition (SVD) can be computed with $14mn^2 + 8n^3$ FLOPs [Higham, 2022]. To orthogonalize a matrix $A \in \mathbb{R}^{m \times n}$ with SVD, we decompose $A = U\Sigma V^\top$ and then multiply UV^\top , which requires

$$14mn^2 + 8n^3 + 2mn^2 = 16mn^2 + 8n^3 < 20mn^2 + 10n^3.$$

Muon requires more FLOPs than SVD! Its speed can only be attributed to its exclusive use of matrix multiplication and addition—which are highly parallelizable and optimized for by modern accelerator hardware—and not due to intrinsic algorithmic efficiency.

I Details on Communication Requirements

In 3D-parallel Dion (Algorithm 3), we perform I/O operations along three separate parallelization dimensions. For simplicity, we consider optimization of a single $m \times n$ parameter matrix. We analyze only the *optimizer-update communication*; forward-pass and backward-pass collectives that are common to all optimizer algorithms are not included. An element-wise optimizer algorithm such as Adam thus incurs zero additional communication due to sharding. We discuss the amount of data transferred along each parallelization dimension in isolation, treating all other dimensions as unused. A summary of results here is given in Table 2.

- **DP:** Dion requires two all-reduce operations for $\hat{P} \in \mathbb{R}^{m \times r}$ and $\hat{Q} \in \mathbb{R}^{n \times r}$, for a total of $(m+n)r$ elements. This replaces the standard DP gradient synchronization required before applying Adam and Muon, which takes place on the gradient matrix $G \in \mathbb{R}^{m \times n}$ of mn elements.
- **FS:** For power iteration, Dion computes a matrix multiplication $\hat{B}^{(i)}Q^{(i)}$ sharded over the FS dimension, which requires an all-reduce of the partial results of size $m \times r$. Additionally, column normalization for $Q^{(i)}$ requires summing the partial column norms $c^{(i)} \in \mathbb{R}^r$. Thus, the total FS communication volume is $(m+1)r$. For comparison, Adam is applied element-wise and does not require any communication due to sharding. Muon requires the full $\mathbb{R}^{m \times n}$ matrix for Newton-Schulz, and requires an all-gather of mn elements [Liu et al., 2025a].
- **TP:** Dion shards the power iteration matrix $Q^{(j)} \in \mathbb{R}^{n \times r_j}$ to reduce optimizer state memory usage, which requires an all-gather of size $n \times r$ at the start of the algorithm. The matrix multiplication $\hat{B}^{(j)}P^{(j)}$ is sharded over the TP dimension, which requires an all-reduce of size $n \times r$. Distributed orthonormalization with randomized Cholesky QR requires two sharded matrix multiplications, resulting in two all-reduces with sizes $k \times r$ and $r \times r$. Using a random sketch oversampling factor $k = 1.25r$ gives a total communication volume of $2nr + 2.25r^2$ elements. As in the case of FS, Adam does not require any communication, and Muon requires an all-gather of size mn .

For efficient GPU utilization, it is desirable that communication operations overlap with computation. Except for the TP all-gather of Q , all communication takes place in the critical path of Algorithm 3.

³Further speedups may be achievable by exploiting symmetric matrix properties, as discussed in [Newhouse et al., 2024]. Under carefully optimized kernel implementations, this could reduce the total cost to approximately $15mn^2 + 5n^3$.

Communication cannot begin until the previous computation finishes, and the next computation cannot begin until the communication itself finishes. However, we note that the critical path only holds for a single parameter matrix, and different matrices can be optimized independently from each other. This allows for an efficient software implementation to overlap communication by simultaneously optimizing different layers of the model.

J Hyperparameter Tuning

Our default model configurations are given in Table 4. Unless otherwise specified, our experiments use the 120M parameter model size. All models use the GPT2 tokenizer with a vocabulary size of 50304. We use rotary position embeddings [Su et al., 2023], non-parametric RMSNorm, and omit biases from linear layers. The activation function for MLP layers is squared ReLU [So et al., 2022].

We train all models on the FineWeb [Penedo et al., 2024] or FineWeb-Edu [Lozhkov et al., 2024] datasets. Unless otherwise specified, we train on an approximately Chinchilla-optimal number of tokens [Hoffmann et al., 2022] (tokens $\approx 20 \times$ model parameters) for each model size. We use NVIDIA H100 or AMD MI300X GPUs for all experiments.

Model	d_{model} / Layers / Heads	Batch Size	Total Steps	Total Tokens
120 M	768 / 12 / 6	1.0 M	3 K	3.1 B
350 M	1024 / 24 / 32	2.1 M	4 K	8.4 B
1.3 B	2048 / 24 / 32	4.2 M	6 K	25.2 B
3 B	3072 / 24 / 32	8.4 M	30 K	63 B

Table 4: Default configurations for each model size.

J.1 Results in Section 5.1

We use the model configurations detailed in Table 4. The target validation losses are chosen to match the loss achieved by AdamW after training on approximately 80% of the Chinchilla-optimal number of tokens. The losses selected are 3.52 for the 120M parameter model, 3.23 for 354M, 2.85 for 1.3B, and 2.7 for 3B.

The AdamW learning rates are selected based on the scaling analysis of Bi et al. [2024]. Specifically, we estimate the optimal value by aligning training FLOPs with the trends reported in their scaling results [Bi et al., 2024, Figure 3]. The resulting learning rates are 0.002 for 120M, 0.0016 for 354M, 0.0012 for 1.3B, and 0.001 for 3B models. We use a 10% warmup schedule and set the weight decay to 0.01 for matrix parameters and 0 for non-matrix parameters.

For both Muon and Dion, we use a fixed learning rate of 0.01 across all model sizes. This choice is supported by our hyperparameter transfer results (Figure 4), which show that the optimal learning rate remains stable across scale.

J.2 Results in Section 5.2

For this experiment, we sweep across batch sizes {256, 1024, 4096, 16384} while maintaining the sequence length at 1024 tokens. We measure the number of training steps required to reach a target validation loss of 3.4 by interpolating the validation loss curve. All models are 120M parameters and are trained on the FineWeb dataset.

Figure 3 shows the step count ratio relative to AdamW. We provide the (interpolated) numerical step counts used in the plot as follows:

Batch Size (# tokens)	AdamW	Muon	Dion (1)	Dion (1/4)
256 K	14172	18017	16662	17209
512 K	4174	3546	2624	4044
4096 K	2248	1318	1062	1631
16384 K	1577	870	663	1118

Dion and Muon use a learning rate of 0.03 and $\mu = 0.95$, with non-matrix parameters optimized by AdamW using a learning rate of 0.002 and $(\beta_1, \beta_2) = (0.9, 0.95)$. The AdamW baseline uses the same AdamW hyperparameters to train the entire model. Weight decay is set to 0.01 across all optimizers, except for the embedding and unembedding layers which always use weight decay 0.

J.3 Results in Section 5.3

All runs are trained on the FineWeb-Edu dataset using a sequence length of 1024. We employ a constant learning rate schedule with a 10% linear cooldown for all runs, and a 10% linear warmup for AdamW and DeMo. For AdamW, we use $(\beta_1, \beta_2) = (0.9, 0.95)$ and sweep over learning rates $\{0.001, 0.003, 0.01\}$ and weight decay values $\{0, 0.01\}$. For Muon and Dion, we fix the momentum parameter to $\mu = 0.95$ and sweep over learning rates $\{0.003, 0.01, 0.03\}$. We use Adam as the scalar optimizer for all Muon and Dion runs, with a learning rate of 0.002 and $(\beta_1, \beta_2) = (0.9, 0.95)$. For DeMo, we sweep over learning rates $\{0.0003, 0.001, 0.003\}$ and use the default compression decay value of 0.999.

For batch size 0.5M, we perform a full sweep across all rank fraction levels and observe that the optimal learning rate remains consistent across rank fraction values for both Dion and DeMo. For batch size 4M, we sweep rank fraction level 1 for Dion and 1/32 for DeMo.

We define the *update density* as the fraction of degrees of freedom retained in each parameter update.

- For Dion, the update density is given by $\delta = r/d$, where r is the low-rank factor and d is the hidden dimension of the Transformer model (d_{model} in Table 4).
- For DeMo, the update density is computed as $\delta = k/s^2$, where k is the number of retained DCT coefficients per chunk, and $s \times s$ is the size of each chunked matrix block. To control the update density across experiments, we set the chunk size as $s = 2k$, which ensures a consistent scaling of density as k varies.

As a proportion of the whole matrix, the data-parallel communication required for both Dion and DeMo is approximately 2δ . Dion synchronizes two low-rank matrices with shapes $m \times r$ and $n \times r$. DeMo exchanges a sparse matrix with k/s^2 nonzero elements, represented with k indices and k values. The exact amount of data transferred depends on the shape $m \times n$ for Dion and the specific sparse matrix format for DeMo.

J.4 Results in Section 5.4

We use the configuration specified in Table 4 for the 3B model. We set the global batch size to 4096 and the sequence length to 2048, resulting in a total of approximately 8.4M tokens per batch.

For Dion, we use the Dion+Lion combination, as discussed in Appendix D, which allows a single learning rate to apply for both matrix and non-matrix parameters. This eliminates the need for separate tuning and makes Dion even more user-friendly. Lion (β_1, β_2) are set to (0.95, 0.98).

For AdamW and Muon, we use Adam with a learning rate of 0.002 to update the non-matrix parameters, a choice that has shown stable performance across scales.

Guided by the hyperparameter transferability results in Section 5.5, we use a fixed learning rate of 0.01 for both Muon and Dion, without applying learning rate warmup. This value appears to generalize well across model sizes.

For AdamW, we select a learning rate of 0.001 based on the scaling analysis of Bi et al. [2024]. Specifically, we estimate the optimal learning by aligning training FLOPs with the trends in their scaling results [Bi et al., 2024, Figure 3]. We use a 10% warmup period for AdamW and a weight decay strength of 0.01.

J.5 Results in Section 5.5

Each model size is trained with its respective Chinchilla-optimal number of tokens on the FineWeb-Edu dataset. All runs use a sequence length of 1024. For both Dion and Muon, we use $\mu = 0.95$ across all model sizes and learning rates. We use a constant learning rate schedule with no warmup

and 10% linear cooldown. We use Adam with learning rate 0.002 and $(\beta_1, \beta_2) = (0.9, 0.95)$ as the optimizer for non-matrix parameters.

K Modifications for Extreme Sparsity

Algorithm 6 Double Dion

Require: Learning rate η , momentum factors μ_1, μ_2 , ranks r_1, r_2 , initial momentums $\hat{M}_1, M_2 \in \mathbb{R}^{m_j \times n_i}$, initial warm-start matrices $Q_1 \in \mathbb{R}^{n_i \times r_1}$ and $Q_2 \in \mathbb{R}^{n_i \times r_2}$

- 1: **for** $t = 1$ **to** T **do**
- 2: Compute local gradient $\hat{G}_t^{(i,j)} \in \mathbb{R}^{m_j \times n_i}$
- Stage 1: Update \hat{M}_1 with new gradient G and compute P_1, R_1**
- 3: $(Q_1)_{t-1}^{(i)} \leftarrow \text{ALLGATHER}_{\text{TP}}((Q_1)_{t-1}^{(i,j)}) \in \mathbb{R}^{n_i \times r_1}$ ▷ unshard Q_1 along TP dimension
- 4: $(\hat{B}_1)_t^{(i,j)} \leftarrow (\hat{M}_1)_{t-1}^{(i,j)} + \hat{G}_t^{(i,j)} \in \mathbb{R}^{m_j \times n_i}$ ▷ update M_1 with new gradient
- 5: $(P_1)_t^{(j)}, (R_1)_t^{(i)} \leftarrow \text{DISTRIBUTED-POWERITER1}((\hat{B}_1)_t^{(i,j)}; (Q_1)_{t-1}^{(i)})$
- 6: $(\hat{M}_1)_t^{(i,j)} \leftarrow \mu_1(\hat{B}_1)_t^{(i,j)} - \mu_1(P_1)_t^{(j)}((R_1)_t^{(i)})^\top \in \mathbb{R}^{m_j \times n_i}$ ▷ error feedback
- 7: $(Q_1)_t^{(i)} \leftarrow \text{DISTRIBUTED-COLUMNNORMALIZE}((R_1)_t^{(i)})$
- 8: $(Q_1)_t^{(i,j)} \leftarrow \text{SHARD}_{\text{TP}}((Q_1)_t^{(i)}) \in \mathbb{R}^{n_i \times r_1}$ ▷ reshard Q_1 along TP dimension
- Stage 2: Update M_2 with P_1, R_1 and compute weight update**
- 9: $(Q_2)_{t-1}^{(i)} \leftarrow \text{ALLGATHER}_{\text{TP}}((Q_2)_{t-1}^{(i,j)}) \in \mathbb{R}^{n_i \times r_2}$ ▷ unshard Q_2 along TP dimension
- 10: $(B_2)_t^{(i,j)} \leftarrow (M_2)_{t-1}^{(i,j)} + (P_1)_t^{(j)}((R_1)_t^{(i)})^\top \in \mathbb{R}^{m_j \times n_i}$ ▷ update M_2 with PR^\top
- 11: $(P_2)_t^{(j)}, (R_2)_t^{(i)} \leftarrow \text{DISTRIBUTED-POWERITER1}((B_2)_t^{(i,j)}; (Q_2)_{t-1}^{(i)})$
- 12: $(M_2)_t^{(i,j)} \leftarrow (B_2)_t^{(i,j)} - (1 - \mu_2)(P_2)_t^{(j)}((R_2)_t^{(i)})^\top \in \mathbb{R}^{m_j \times n_i}$ ▷ error feedback
- 13: $(Q_2)_t^{(i)} \leftarrow \text{DISTRIBUTED-COLUMNNORMALIZE}((R_2)_t^{(i)})$
- 14: $X_t^{(i,j)} \leftarrow X_{t-1}^{(i,j)} - \eta(P_2)_t^{(j)}((Q_2)_t^{(i)})^\top \in \mathbb{R}^{m_j \times n_i}$ ▷ weight update
- 15: $(Q_2)_t^{(i,j)} \leftarrow \text{SHARD}_{\text{TP}}((Q_2)_t^{(i)}) \in \mathbb{R}^{n_i \times r_2}$ ▷ reshard Q_2 along TP dimension
- 16: **end for**

Certain use cases, such as large-scale geographically distributed training, may benefit from highly minimized DP communication bandwidth while providing more relaxed constraints on FS and TP communications. Observing that Dion is outperformed by DeMo [Peng et al., 2024] at particularly low update densities (1/64 or less), we describe a two-stage variant of Dion (Algorithm 6) that uses a smaller rank r_1 for the DP all-reduce and a larger rank r_2 only for FS and TP.

The first stage updates a local momentum \hat{M}_1 with the local gradient \hat{G} , producing DP-synchronized low-rank matrices P_1 and R_1 . The second stage updates a globally synchronous momentum M_2 with $P_1 R_1^\top$ and then computes the parameter update. All second stage variables are fully synchronous across the DP axis, so the DP all-reduce is unnecessary. Note the different error feedback rules—the first stage error feedback resembles that of DeMo, and we find $(\mu_1, \mu_2) = (0.999, 0.95)$ is effective.

In Figure 8 (left), we evaluate the performance of Algorithm 6 with $r_1/d = 1/128$ and $r_2/d = 1/4$, comparing to DeMo with update density 1/128 and standard Dion (Algorithm 3) with update densities 1/128 and 1/4. It is worth noting the extreme sparsity of a 1/128 rank fraction with respect to the 120M parameter model size—given that $d_{\text{model}} = 768$, we have $r_1 = 6$.

All three algorithms with 1/128 update density require equivalent DP communication, but the two-stage Dion has the best validation loss. We attribute its improvement to the use of two different error feedback rules—attempts to use either rule for both stages led to poorer results. While effective, the drawback of this modified Dion variant is that optimizer compute and memory usage are roughly doubled. We treat the findings here as preliminary and leave further improvements to future work.

As an additional communication optimization, we propose that the second stage can use one-step delayed $(P_1)_{t-1}$ and $(R_1)_{t-1}$ from the first stage to update M_2 (line 10 in Algorithm 6). Making this change allows both stages to run in parallel, and the DP all-reduce may be *overlapped with the forward and backward pass*. We study the impact of this change in Figure 8 (right). As one might expect, the

delayed update comes at a cost of slower convergence. Even then, it still outperforms standard Dion at rank fraction $1/128$, and it roughly matches DeMo without any delay. In contrast, attempting to introduce a one-step delay to DeMo’s compression output severely degraded its performance.

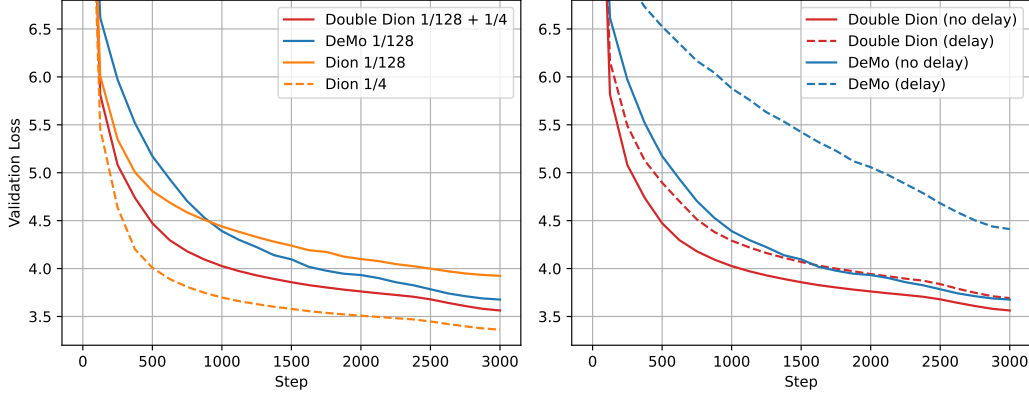


Figure 8: **Left:** Comparison between Algorithm 6, Dion, and DeMo at $1/128$ equivalent data-parallel communication requirements. Dion with rank fraction $1/4$ is also shown for reference. The modified Double Dion algorithm offers a substantial improvement over standard Dion and outperforms DeMo. **Right:** Effect of introducing a one-step delay to P_1, R_1 in Algorithm 6 and to the compressed state in DeMo. Double Dion with a one-step delay matches DeMo without any delay, while DeMo’s performance is substantially worsened by adding a one-step delay.

Additional details: We train 120M parameter models on the FineWeb dataset, using batch size 1024 and sequence length 1024. The total training steps is 3000, and a linear learning rate decay to zero is applied over the last 20% of steps. DeMo uses learning rate 0.001. Dion and Double Dion both use a base learning rate of 0.01, and we apply the scaling factors in Table 1. Non-matrix parameters are optimized using Lion with $(\beta_1, \beta_2) = (0.95, 0.98)$, and the one-step delay does not apply. All optimizers use a weight decay of 0.01 for matrix parameters and 0 for non-matrix parameters.