

Randomized Gradient Compression for Memory-Efficient Reinforcement Learning

Ben Zhao
Iluvatar Labs Inc.
ben@iluvatarlabs.com

Marvin
Iluvatar Labs Inc.
marvinsci.com

13 October 2026*

Abstract

Reinforcement learning improves the reasoning of language models, but full-weight RL is memory-bound: dense AdamW keeps an FP32 copy of every trained weight, its gradient and two optimizer moments. Randomized Gradient Compression (RGC) trains the same weights with none of them. Each layer forms its weight gradient through a random orthogonal projection of its input, a moment-free normalized SGD rule applies the reconstructed update in row blocks, and the weights stay in BF16 with stochastic rounding. Trained with GRPO on GSM8K, RGC is on par with dense AdamW and LoRA in accuracy while using less memory than LoRA at matched rank, in training and over the full RL loop, with the gap growing with rank and sequence length. Computing generation and the loss in chunks cuts the full loop’s memory by a third: full-weight RL of Qwen3-8B peaks at 18.7 GiB and Qwen3-14B at 29.1 GiB, only 6% above the 14B model’s own BF16 weights, against 105.8 and 199.8 GiB for dense AdamW’s weights, gradients and optimizer state alone. A state-accounting model predicts RGC’s training-memory gap to LoRA before training, to within 0.12 GiB in all nine tests.

1 Introduction

Reinforcement learning (RL) is now a standard way to improve the reasoning of language models, but training the policy is memory-hungry: dense optimization keeps a gradient and optimizer moments for every trained weight, on top of the model, its activations and the generation cache. Adafactor reduces optimizer state by factoring second-moment estimates [22], MeZO avoids backpropagation with zeroth-order estimates [9], and low-rank adaptation (LoRA) freezes the base weights and trains a low-rank additive term [5], which remains effective at small ranks in RL [18]. Another family changes the pretrained matrices directly but shrinks what the optimizer sees or sends: PowerSGD transmits rank-limited gradient approximations [23], GaLore runs the optimizer inside a gradient subspace recomputed by singular value decomposition [29], Flora uses random projections that change over training [3], and GoLore draws its projections uniformly from orthonormal frames [4]. CompAct stores randomly projected layer inputs for the backward pass [20], and LoRA-FA freezes LoRA’s down-projection so that only the projected input is stored [28]. Methods that train the weights themselves usually also keep them in FP32, because BF16 or FP16 rounding can erase small updates [11]; stochastic rounding instead keeps such updates correct in expectation [2, 27, 14].

Randomized Gradient Compression (RGC) places a random orthogonal projection inside the weight-gradient computation of every selected layer, so that gradient exists only in compressed form, updates the original weights with a moment-free SGD rule, and keeps those weights in BF16 with stochastic rounding. LoRA leaves each pretrained matrix fixed and trains a rank- r term added to it; RGC changes the matrices themselves, and its accumulated change is not rank-limited. Its persistent training state is fixed by the layer shapes and the rank: for Qwen3-8B at rank 64 it is 2.06 bytes per parameter, the BF16 weights plus 3%, against 13.9 for dense AdamW. Our contributions are:

*Last updated 25 October 2026.

- an original-weight RL update that reconstructs each layer’s gradient from a redrawn, uniformly random orthogonal projection, which is unbiased and has the smallest worst-case error of any unbiased rank- r projection, and applies it in 128-row blocks with per-matrix normalized SGD to BF16 weights with stochastic rounding, with no dense gradient, no optimizer moments and no FP32 weight copy;
- a state-accounting model that predicts, before training, RGC’s training memory relative to LoRA, confirmed to within 0.12 GiB across FP32 and BF16 storage, four ranks and two model sizes;
- a full-loop recipe, with generation in chunks of responses, the output head and loss over token chunks, and RGC’s buffers released during generation, that cuts the RL loop’s memory by a third with no increase in RGC’s loop time;
- experiments on Qwen3-1.7B-, 8B- and 14B-Base in which RGC is on par with dense AdamW and LoRA in accuracy, uses less memory than LoRA at matched rank in training and over the full loop, with a gap that grows with rank and sequence length, and trains the 8B and 14B models in 18.7 and 29.1 GiB.

2 Method

Compressed accumulation. For a selected matrix $W_\ell \in \mathbb{R}^{d_\ell \times k_\ell}$, microbatch m has input $X_{\ell m}$, output $Y_{\ell m} = X_{\ell m} W_\ell^\top$ and output derivative $D_{\ell m} = \partial \mathcal{L} / \partial Y_{\ell m}$, where the loss $\mathcal{L}$ includes the full-batch token normalization. RGC accumulates

$$C_\ell = \sum_m D_{\ell m}^\top (X_{\ell m} P_\ell), \quad H_\ell = C_\ell P_\ell^\top. \quad (1)$$

As in CompAct [20], each selected layer saves the projected input $X_{\ell m} P_\ell$ in place of $X_{\ell m}$ for the backward pass, while its output and the derivative $D_{\ell m} W_\ell$ passed to earlier layers are unchanged. Because W_ℓ and P_ℓ change only between steps, $C_\ell = G_\ell P_\ell$, where $G_\ell = \sum_m D_{\ell m}^\top X_{\ell m}$ is the dense weight gradient, which is never formed. The FP32 accumulator C_ℓ is $d_\ell \times r$ and the projection P_ℓ is $k_\ell \times r$. H_ℓ is only ever formed in blocks of 128 output rows, so no dense weight gradient or dense update is allocated during training.

Random orthogonal projection. For each selected matrix, draw a standard Gaussian $k_\ell \times r$ matrix in FP32, compute its reduced QR factorization, correct the column signs of Q_ℓ by the signs of R_ℓ ’s diagonal [10, 16], and set

$$P_\ell = \sqrt{k_\ell / r} Q_\ell, \quad Q_\ell^\top Q_\ell = I_r. \quad (2)$$

This gives a uniformly random orthonormal frame, as in GoLore [4]. RGC stores P_ℓ in BF16 and releases the QR workspace before drawing the next layer’s projection. Projections come from a dedicated random-number stream (seed 1000042) that response sampling does not use, and layers draw independently. After every applied update a new projection is drawn for the next step; a step that applies no update keeps its unused projection.

Two moment-free update rules. With warmup $w_t = \min(t/5, 1)$ for scheduled step $t \geq 1$, plain SGD applies

$$W_\ell \leftarrow W_\ell - w_t \eta_r H_\ell. \quad (3)$$

The normalized rule rescales each matrix’s update by that matrix’s own weight-to-update norm ratio:

$$Z_\ell = \frac{\|W_\ell\|_F}{\|H_\ell\|_F + 10^{-12}} H_\ell, \quad W_\ell \leftarrow W_\ell - w_t \tau_r Z_\ell. \quad (4)$$

This is LARS-style layerwise scaling [25] without momentum or weight decay. Plain SGD keeps the estimated gradient’s relative magnitudes across matrices; the normalized rule moves every matrix by the same fraction $w_t \tau_r$ of its own norm. A zero H_ℓ leaves its matrix unchanged. Norms are reduced blockwise in FP64 and fixed before the update is applied; reconstruction uses FP32. Neither rule stores any optimizer state.

BF16 weight storage. The selected weights stay in BF16, as loaded, with no FP32 copy. For each block of 128 output rows, the updated block is formed in FP32 and written back to BF16 with stochastic rounding: each entry rounds to one of its two neighbouring BF16 values, with probabilities that make the stored value equal the FP32 result in expectation [2, 27, 14]. Its random bits come from a third dedicated stream (seed 2000042). Rounding to nearest would erase most updates here, which is why mixed-precision training keeps FP32 copies of the weights [11]: at the calibrated scales a typical update is about 10^{-6} to 3×10^{-6} , or 0.4–1.2% of the BF16 spacing at the weights’ root-mean-square magnitude. Under the normalized rule every matrix moves by the same fraction of its own norm, so updates are a similar fraction of the local BF16 spacing in every layer. The accumulator C_ℓ stays in FP32. We write FP32 RGC and BF16 RGC where the storage matters.

Estimator properties. For a fixed gradient $G \in \mathbb{R}^{d \times k}$ independent of a freshly drawn projection, $A = QQ^\top$ is a uniformly random rank- r orthogonal projector with $A^2 = A$ and $\mathbb{E}[A] = (r/k)I_k$. Hence

$$\mathbb{E}[PP^\top] = I_k, \quad \mathbb{E}[GPP^\top \mid G] = G, \quad (5)$$

$$\mathbb{E}\left[\|GPP^\top - G\|_F^2 \mid G\right] = \left(\frac{k}{r} - 1\right) \|G\|_F^2. \quad (6)$$

Expanding $((k/r)A - I_k)^2$ and taking expectations gives Equation 6. No unbiased reconstruction of this form does better in the worst case: for $M = PP^\top$, which is positive semidefinite with rank at most r and $\mathbb{E}[M] = I_k$, the inequality $\text{tr}(M^2) \geq (\text{tr } M)^2/r$ and Jensen’s inequality give $\text{tr } \mathbb{E}[M^2] \geq k^2/r$, so $\mathbb{E}[M^2]$ has an eigenvalue of at least k/r and some G incurs error at least $(k/r - 1) \|G\|_F^2$. The scaled orthogonal frame attains this bound for every G . Like randomized automatic differentiation [12], the estimator trades variance for memory. These identities hold in exact arithmetic; BF16 storage of P_ℓ and BF16 products make them approximate in the implementation, and the normalized rule’s nonlinear scaling does not inherit unbiasedness. The rank of each computed update is at most r , so in exact arithmetic $\text{rank}(W_T - W_0) \leq \min(d, k, Tr)$ after T applied updates: as in Flora [3], redrawing the projection lets rank- r updates add up to a higher-rank change, an effect ReLoRA obtains through periodic restarts [8].

Memory accounting. Let $N_W = \sum_\ell d_\ell k_\ell$ over the selected matrices. Above a common all-BF16 base, the persistent training state in bytes is

$$S_{\text{RGC}}(r) = s N_W + \sum_\ell (4d_\ell r + 2k_\ell r), \quad (7)$$

$$S_{\text{LoRA}}(r) = 16r \sum_\ell (d_\ell + k_\ell), \quad (8)$$

$$S_{\text{dense}} = 14N_W. \quad (9)$$

For RGC, $s = 2$ bytes per weight with FP32 storage and $s = 0$ with BF16 storage, and the sum covers the FP32 accumulators and BF16 projections. For LoRA, the factor 16 covers FP32 adapter weights, their gradients and two Adam moments; dense AdamW adds FP32 weights, gradients and two moments for every trained weight. Activations and transient buffers are excluded. For Qwen3-1.7B-Base, $N_W = 1,409,286,144$; RGC’s compressed state costs 3,325,952 bytes per rank and LoRA’s adapter state 17,432,576 bytes per rank, so RGC’s is 19% of LoRA’s. With FP32 storage the 2.625 GiB premium places RGC at ranks 64, 128 and 256 level with LoRA at ranks of about 174, 186 and 211. With BF16 storage RGC is smaller at every rank, by $r \times 14,106,624$ bytes: 0.21, 0.84, 1.68 and 3.36 GiB at ranks 16, 64, 128 and 256. For Qwen3-8B-Base, $N_W = 6,945,767,424$ over 252 matrices, RGC costs 8,257,536 and LoRA 43,646,976 bytes per rank, and dense AdamW’s 90.56 GiB increment and the 15.26 GiB BF16 model come to 105.82 GiB. For Qwen3-14B-Base, $N_W = 13,212,057,600$ over 280 matrices, RGC costs 12,206,080 bytes per rank, and dense AdamW needs 172.27 GiB above the 27.51 GiB BF16 model, 199.77 GiB in all.

Full-loop memory. An RL step’s peak is the larger of its generation and policy-training phases. Three changes lower both without changing the training algorithm. Generation decodes a step’s responses in chunks of N sequences, all from the same weights, so the key/value cache holds N sequences rather than all of them,

and each chunk stops at its own longest response. Policy training computes the frozen output head and log-softmax over 128-token chunks under activation checkpointing, so the full-vocabulary logits of a whole response are never resident; the loss is identical, and the accumulated gradient matches an FP32 output-head reference as closely as the unchunked path does (2.78% versus 2.80% relative error). RGC releases C_ℓ and P_ℓ after each update and draws P_ℓ from its stream just before the next training phase, which leaves every projection, update and rounding draw bit-identical while removing RGC’s state from the generation peak. LoRA’s adapters and Adam moments persist across steps and cannot be released.

3 Experimental setup

Model, data and reward. We train Qwen3-1.7B-Base (revision `ea980cb0a6c2ae4b936e82123acc929f1cec04c1`), Qwen3-8B-Base (revision `49e3418fbbbca6ecbdf9608b4d22e5a407081db4`) and Qwen3-14B-Base (revision `0b0bd3732e2c374d483664439ea334928b65f304`) [24, 17] on GSM8K (main, revision `740312add88f781978c0658806c59bc2815b9866`) [1, 13], split into 7,217 training and 256 development questions, with all 1,319 test questions held out. The chat template shipped with the model formats a system message, “Please reason step by step, and put your final answer within `\boxed{\}`.”, and the question. Generation stops on `<|endoftext|>` or `<|im_end|>`. The reward is 1 when Math-Verify 0.8.0 [7] judges the boxed answer equal to the reference and 0 otherwise, including answers that are missing, cannot be parsed or are truncated at the 1,024-token completion cap; truncated responses are excluded from the loss. Evaluation is greedy with the same cap and scoring.

Objective. Each step samples four responses for each of 16 prompts and accumulates over 64 microbatches of one response each. Following GRPO [21, 6], response i to question q has advantage $A_{qi} = (R_{qi} - \bar{R}_q)/(s_q + 10^{-4})$, with s_q the sample standard deviation of the group’s rewards. With usable-token mask M_{qij} and $N_+ = \max(\sum_{qij} M_{qij}, 1)$,

$$\mathcal{L}(\theta) = -\frac{1}{N_+} \sum_{qij} M_{qij} A_{qi} \exp(\log \pi_\theta(o_{qij} \mid q, o_{qi, < j}) - \text{sg}[\log \pi_\theta(o_{qij} \mid q, o_{qi, < j})]), \quad (10)$$

where `sg` stops the gradient and the normalization over all usable tokens in the rollout follows DAPO [26]. Each fresh batch is used for exactly one update, so the ratio is one in value and PPO [19] clipping never acts. There is no reference KL, entropy bonus or gradient clipping. A batch in which every unmasked response has zero advantage applies no update to the weights, the Adam moments or the projections. Sampling uses temperature 1 and top- p 1 with no top- k filtering and a 3,072-token prompt cap; every method warms up over five steps and then holds its scale constant.

Baselines and precision. All methods train the query, key, value and output projections and the MLP gate, up and down projections of every block; everything else is frozen. Dense AdamW and FP32 RGC keep these weights in FP32, and BF16 RGC keeps them in BF16. LoRA keeps a BF16 base and FP32 adapters with standard PEFT initialization, no dropout and $\alpha/r = 0.5$. Dense and LoRA use single-tensor AdamW [15] with betas (0.9, 0.95), $\epsilon = 10^{-8}$ and no weight decay. Computation uses BF16, scaled dot-product attention and non-reentrant activation checkpointing, with PyTorch 2.9.1, Transformers 4.57.1, TRL 0.24.0 and PEFT 0.17.1. Every run occupies a single RTX PRO 6000 Blackwell GPU; generation uses the key/value cache with weight-cast caching disabled for every method. Memory is the peak allocated by PyTorch. Runs with the full-loop recipe, and the second 8B rank-64 run, were held under a fixed allocator cap (22.5 GiB at 8B with the recipe, 30.5 GiB otherwise), which none exceeded.

Step-size calibration. RGC’s base scales are calibrated on the untrained model rather than tuned freely. Eight training-only rollout batches (seed 1042) are saved and replayed at unchanged weights. For each batch b , the dense gradient g_b defines the first AdamW step from zero moments at learning rate 10^{-6} ,

$$u_b = -10^{-6} \frac{g_b}{|g_b| + 10^{-8}}, \quad \delta_0 = \text{median}_b \text{RMS}(u_b) \approx 9.89 \times 10^{-7}, \quad (11)$$

with elementwise division and RMS taken over all selected weights. Replaying the same batches through RGC at rank r gives update directions $H_{b,r}$ and $Z_{b,r}$, and the base scales (Table 1) are

$$\eta_{0,r} = \frac{\delta_0}{\text{median}_b \text{RMS}(H_{b,r})}, \quad \tau_{0,r} = \frac{\delta_0}{\text{median}_b \text{RMS}(Z_{b,r})}. \quad (12)$$

Each rule tries its base scale, which matches the size of a first AdamW step at learning rate 10^{-6} , and three times that scale; all scales were fixed before training. For Qwen3-8B-Base and Qwen3-14B-Base the same procedure at rank 64 gives $\tau_{0,64} \approx 3.63 \times 10^{-5}$ and 3.98×10^{-5} .

Table 1: Calibrated RGC base scales for Qwen3-1.7B-Base. $\tau_{0,r}$ does not depend on the rank because $\|Z_\ell\|_F = \|W_\ell\|_F$.

Rank	Plain SGD $\eta_{0,r}$	Normalized SGD $\tau_{0,r}$
64	4.06×10^{-3}	2.80×10^{-5}
128	6.06×10^{-3}	2.80×10^{-5}
256	8.75×10^{-3}	2.80×10^{-5}

Grid and selection. On Qwen3-1.7B-Base, sixteen configurations each train two candidates for 200 steps from the same base checkpoint with training seed 42: dense AdamW (learning rates 3×10^{-6} and 10^{-5}); LoRA at ranks 16, 64, 128 and 256 (3×10^{-5} and 10^{-4}); BF16 RGC with normalized SGD at ranks 8, 16, 64, 128 and 256; and FP32 RGC with plain and with normalized SGD at ranks 64, 128 and 256. Each RGC configuration tries the calibrated scale and three times it; $\tau_{0,r}$ is the same at every rank. Development accuracy is measured every 50 steps (Appendix A). In each configuration the candidate with the higher development accuracy at step 200 is selected, with ties going to the lower scale, and selections were fixed before any test evaluation. On Qwen3-8B-Base, BF16 RGC runs at rank 64, three times (standard, a second run, and with the full-loop recipe), and at rank 128; LoRA runs at rank 64 and, with the full-loop recipe, at rank 16; dense AdamW runs at 3×10^{-6} . LoRA uses 3×10^{-5} , the rate selected at 1.7B for ranks 64 and above. On Qwen3-14B-Base, BF16 RGC runs at rank 64 with the full-loop recipe. Each 8B and 14B RGC run uses the calibrated scale. The recipe decodes 16 sequences at a time at 8B and 4 at 14B. Memory predictions were fixed before the runs they describe, and interrupted runs were restarted from the base checkpoint with unchanged settings. Training and full-loop memory are the maximum allocated CUDA peaks over each run’s policy-training phases and over its complete steps, generation included. For each RGC configuration, a paired bootstrap over test questions (10,000 resamples, seed 1043) gives a 95% interval for its accuracy difference from LoRA, dense AdamW and FP32 RGC.

4 Results

Quality. Training lifts test accuracy from 64.90% to between 80.59% and 83.78% in every configuration of Table 2. BF16 RGC reaches 82.94%, 83.32%, 83.70%, 80.59% and 82.11% at ranks 8, 16, 64, 128 and 256, on par with LoRA (82.11–83.78%) and dense AdamW (83.70%). At rank 64 it answers exactly as many test questions as dense AdamW, and at rank 16 it is 1.21 points above LoRA at the same rank (Table 5).

Memory. At matched rank, BF16 RGC uses less memory than LoRA, in policy training by 0.21, 0.86, 1.69 and 3.48 GiB at ranks 16, 64, 128 and 256 (3.5%, 12.8%, 21.8% and 35.0%) and over the full loop by 0.02, 0.58, 1.16 and 2.08 GiB, and the gap grows with rank (Figure 1). Against dense AdamW’s 24.66 GiB training peak and 25.37 GiB full loop, RGC uses 73.7–76.8% and 48.6–51.6% less.

Predicted and measured memory. Table 3 compares the state-accounting prediction of Section 2 with the measured difference between RGC’s and LoRA’s training peaks. All nine predictions, each fixed before its runs, match to within 0.12 GiB: with FP32 storage LoRA is smaller at ranks 64 and 128 and RGC at rank 256, and with BF16 storage RGC is smaller at every rank and on the 8B model.

Table 2: Qwen3-1.7B-Base: test accuracy and memory for the selected candidate of every configuration. RGC uses normalized SGD and BF16 weights. Train and Loop are allocated peaks for policy training and the full loop, with standard generation and loss. GiB is 2^{30} bytes. Loop time excludes setup, evaluation and checkpointing.

Method	Rank	Test (%)	Train (GiB)	Loop (GiB)	Loop (min)
Untrained base	–	64.90	–	–	–
Dense AdamW	–	83.70	24.66	25.37	133.03
LoRA + AdamW	16	82.11	5.95	12.44	172.54
LoRA + AdamW	64	83.62	6.75	13.16	171.70
LoRA + AdamW	128	83.78	7.78	13.93	209.68
LoRA + AdamW	256	83.62	9.96	15.12	161.41
RGC	8	82.94	5.71	12.27	158.41
RGC	16	83.32	5.74	12.42	168.82
RGC	64	83.70	5.89	12.57	177.90
RGC	128	80.59	6.08	12.77	147.28
RGC	256	82.11	6.48	13.04	164.86

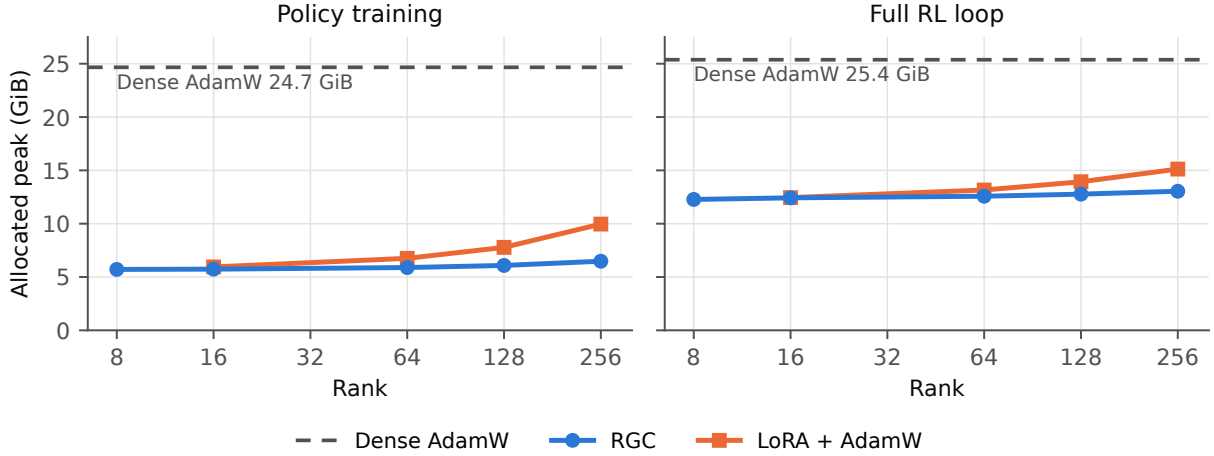


Figure 1: Qwen3-1.7B-Base: allocated peak memory against rank for policy training (left) and the full RL loop (right), with standard generation and loss.

Table 3: RGC minus LoRA training memory, at the same rank except the last row, which compares RGC at rank 128 with LoRA at rank 64: the persistent-state prediction (Section 2) versus the measured difference in allocated policy-training peaks, in GiB. RGC uses normalized SGD.

Model	RGC weights	Rank	Predicted	Measured
Qwen3-1.7B	BF16	16	−0.210	−0.209
Qwen3-1.7B	BF16	64	−0.841	−0.863
Qwen3-1.7B	BF16	128	−1.682	−1.694
Qwen3-1.7B	BF16	256	−3.363	−3.483
Qwen3-1.7B	FP32	64	+1.784	+1.783
Qwen3-1.7B	FP32	128	+0.943	+0.948
Qwen3-1.7B	FP32	256	−0.738	−0.838
Qwen3-8B	BF16	64	−2.109	−2.115
Qwen3-8B	BF16	128	−1.617	−1.631

FP32 storage and plain SGD. Table 4 lists RGC with FP32 weights. Storing the weights in BF16 lowered the training peak by 2.64–2.65 GiB at each rank, slightly more than the 2.625 GiB premium, and changed test accuracy by +0.45, −1.44 and −0.23 points, with every paired interval containing zero (Table 5). With stochastic rounding, 1.8% of the stored weights changed per applied update on average at the base scale and 4.4% at three times it, so a weight changed about 3.5 and 8.7 times over a run on average. With FP32 weights, normalized SGD beat plain SGD at every rank, by 12, 27 and 11 test questions.

Table 4: Qwen3-1.7B-Base: RGC with FP32 weights, columns as in Table 2.

RGC variant	Rank	Test (%)	Train (GiB)	Loop (GiB)	Loop (min)
Plain SGD, FP32	64	82.34	8.53	15.10	167.24
Plain SGD, FP32	128	79.98	8.73	15.30	164.01
Plain SGD, FP32	256	81.50	9.13	15.69	161.64
Normalized SGD, FP32	64	83.24	8.53	15.10	152.43
Normalized SGD, FP32	128	82.03	8.73	15.11	164.92
Normalized SGD, FP32	256	82.34	9.13	15.69	192.09

Table 5: BF16 RGC accuracy minus LoRA (same rank at 1.7B, rank 64 at 8B), dense AdamW and same-rank FP32 RGC, in test percentage points, with paired-question 95% bootstrap intervals.

RGC	Δ LoRA [95% CI]	Δ dense [95% CI]	Δ FP32 RGC [95% CI]
Qwen3-1.7B, rank 8	–	−0.76 [−2.65, +1.06]	–
Qwen3-1.7B, rank 16	+1.21 [−0.83, +3.26]	−0.38 [−2.27, +1.52]	–
Qwen3-1.7B, rank 64	+0.08 [−1.67, +1.82]	0.00 [−1.74, +1.74]	+0.45 [−1.29, +2.20]
Qwen3-1.7B, rank 128	−3.18 [−5.38, −1.06]	−3.11 [−5.23, −0.99]	−1.44 [−3.64, +0.68]
Qwen3-1.7B, rank 256	−1.52 [−3.56, +0.38]	−1.59 [−3.49, +0.30]	−0.23 [−2.20, +1.67]
Qwen3-8B, rank 64	−1.44 [−2.58, −0.30]	–	–
Qwen3-8B, rank 64, second run	−0.99 [−2.12, +0.15]	–	–
Qwen3-8B, rank 128	−0.61 [−1.82, +0.61]	–	–
Qwen3-8B, rank 64, full-loop recipe	−0.68 [−1.74, +0.30]	–	–

Long sequences. LoRA keeps each recomputed layer’s full input for its adapter gradients, whereas RGC keeps only the rank- r projection, so RGC’s advantage grows with sequence length. In a policy-training phase on Qwen3-8B-Base with sequences of 1,024 to 16,384 tokens (Table 6), RGC at rank 64 uses 1.57 GiB less than LoRA at rank 16 at 16,384 tokens, and the gap grows by 1.66 GiB from 1,024 tokens.

Table 6: Qwen3-8B-Base: allocated peak of a policy-training phase on sequences of the stated length, one sequence per microbatch, with the output head over 128-token chunks. Differences are computed before rounding.

Tokens	RGC, rank 64 (GiB)	LoRA, rank 16 (GiB)	RGC − LoRA
1,024	16.38	16.29	+0.09
4,096	17.78	18.04	−0.27
16,384	23.79	25.37	−1.57

The full-loop recipe. Without the recipe, generation sets the full loop’s peak. At 8B, decoding all 64 responses at once peaks at 27.41 GiB; chunks of 16 and 8 sequences peak at 18.68 and 17.22 GiB, and releasing RGC’s buffers removes a further 0.49–0.50 GiB, the size of its compressed state (Table 7). Computing the loss over 128-token chunks lowers the training peak by 1.92–2.02 GiB for both methods at both sizes (Table 8). Decoding one step’s 64 responses to a fixed set of prompts from the untrained model takes longer in smaller

chunks (155 seconds at 16 sequences at a time against 113 seconds for all 64 at once), but over a full run RGC’s 200-step loop took 335.9 minutes with the recipe (chunks of 16) against 333.6 minutes without, while its full-loop peak fell from 27.95 to 18.67 GiB.

Table 7: Qwen3-8B-Base: allocated generation peak and decoding time for one step’s 64 responses from the untrained model, decoded the stated number of sequences at a time. LoRA is at rank 16; “released” means RGC’s buffers are released during generation.

At once	RGC (GiB)	RGC, released (GiB)	LoRA (GiB)	RGC, released (s)	LoRA (s)
64	27.41	26.92	27.40	113	115
32	21.59	21.09	21.58	124	134
16	18.68	18.18	18.67	155	184
8	17.22	16.73	17.21	205	294

Table 8: Allocated policy-training peak on a replayed rollout batch, with the output head over whole responses and over 128-token chunks, in GiB. Changes are computed before rounding.

Model	Method	Full logits (GiB)	128-token chunks (GiB)	Change
Qwen3-1.7B	RGC, rank 64	5.88	3.86	−2.02
Qwen3-1.7B	LoRA, rank 16	5.81	3.83	−1.98
Qwen3-8B	RGC, rank 64	18.44	16.43	−2.00
Qwen3-8B	LoRA, rank 16	18.27	16.35	−1.92

Qwen3-8B-Base. Dense AdamW needs 105.8 GiB for its weights, gradients and optimizer state at 8B, and it ran out of memory during its first optimizer step. RGC trained all 252 selected matrices: its four runs reach 92.34–93.18% test accuracy, on par with LoRA’s 93.78%, from an untrained 85.82% (Table 9). At rank 64 RGC’s training peak is 2.12 GiB below LoRA’s at the same rank, against a predicted 2.11 GiB. With the full-loop recipe, RGC at rank 64 and LoRA at rank 16 both peak at 18.7 GiB over the full loop. RGC trains in 0.25 GiB less, reaches 93.10% test accuracy against LoRA’s 92.04% (+1.06 points, [−0.23, +2.35]), and completes its 200 steps in 336 minutes against LoRA’s 451.

Table 9: Qwen3-8B-Base: test accuracy and memory, columns as in Table 2.

Method	Rank	Test (%)	Train (GiB)	Loop (GiB)	Loop (min)
Untrained base	–	85.82	–	–	–
RGC	64	92.34	18.45	27.95	333.57
RGC, second run	64	92.80	18.44	27.78	343.32
RGC	128	93.18	18.94	28.27	372.47
RGC, full-loop recipe	64	93.10	16.43	18.67	335.87
LoRA + AdamW	64	93.78	20.57	29.24	326.97
LoRA + AdamW, full-loop recipe	16	92.04	16.67	18.70	451.40
Dense AdamW	–	out of memory during its first optimizer step			

Qwen3-14B-Base. With the full-loop recipe, RGC trains all 280 selected matrices of the 14B model with a full-loop peak of 29.09 GiB, 6% above the model’s own 27.51 GiB of BF16 weights, against 199.8 GiB for dense AdamW’s weights, gradients and optimizer state alone, and lifts test accuracy from 90.22% to 94.77% (Table 10).

Table 10: Qwen3-14B-Base: test accuracy and memory, columns as in Table 2.

Method	Rank	Test (%)	Train (GiB)	Loop (GiB)	Loop (min)
Untrained base	–	90.22	–	–	–
RGC, full-loop recipe	64	94.77	29.09	29.09	738.65
Dense AdamW	–	needs 199.8 GiB before activations			

5 Discussion

RGC makes full-weight RL cost little more than holding the model: the entire RL loop of Qwen3-14B peaks at 29.09 GiB, 6% above its 27.51 GiB of BF16 weights. For Qwen3-8B at rank 64 its persistent training state is 2.06 bytes per parameter, the BF16 weights plus 3%, against 13.9 for dense AdamW, and its full loop peaks at 18.67 GiB. Three properties make this work. The gradient is compressed where it is formed, so neither a dense gradient nor an optimizer state ever exists. Stochastic rounding keeps the BF16 weights correct in expectation: a weight changes 3.5 to 8.7 times over a run on average, and accuracy matches FP32 storage. And rank is nearly free: RGC’s state grows by 3.3 MB per rank at 1.7B and 8.3 MB at 8B, and accuracy at rank 8 is on par with rank 64.

Against LoRA, RGC trains the weights themselves rather than a rank- r correction, and does so with less memory at matched rank, in training and over the full loop, with a gap that grows with rank and with sequence length, the regime of long reasoning traces. Its accuracy is on par with LoRA’s, as LoRA’s is with full fine-tuning in RL [18]; RGC delivers full-weight training at LoRA’s cost. The full-loop recipe applies to any method and removes a third of the loop’s memory, at no cost in RGC’s loop time; with it, full-weight RL of the 8B and 14B models peaks at 18.7 and 29.1 GiB. The accounting model predicted every training-memory gap to LoRA before training, to within 0.12 GiB.

6 Conclusion

Randomized Gradient Compression trains every attention and MLP weight of a language model with RL at little more than the memory of the model itself, with no dense gradient, no optimizer state and no FP32 weight copy. It is on par with dense AdamW and LoRA in accuracy, uses less memory than LoRA at matched rank in training and over the full loop, and with chunked generation and loss trains Qwen3-8B and Qwen3-14B in 18.7 and 29.1 GiB. Its training-memory gap to LoRA is predictable before training.

References

- [1] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.
- [2] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1737–1746, 2015.
- [3] Yongchang Hao, Yanshuai Cao, and Lili Mou. Flora: Low-rank adapters are secretly gradient compressors. In *Proceedings of the 41st International Conference on Machine Learning*, 2024.
- [4] Yutong He, Pengrui Li, Yipeng Hu, Chuyan Chen, and Kun Yuan. Subspace optimization for large language models with convergence guarantees. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 22468–22522. PMLR, 2025.
- [5] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.
- [6] Hugging Face TRL contributors. GRPO trainer. TRL version 0.24.0 documentation and source, 2025. Implementation: https://github.com/huggingface/trl/blob/v0.24.0/trl/trainer/grpo_trainer.py.

- [7] Hynek Kydlíček. Math-Verify: Math verification library. Version 0.8.0, 2025. Source repository: github.com/huggingface/Math-Verify.
- [8] Vladislav Lialin, Sherin Muckatira, Namrata Shivagunde, and Anna Rumshisky. ReLoRA: High-rank training through low-rank updates. In *International Conference on Learning Representations*, 2024.
- [9] Sadhika Malladi, Tianyu Gao, Eshaan Nichani, Alex Damian, Jason D. Lee, Danqi Chen, and Sanjeev Arora. Fine-tuning language models with just forward passes. In *Advances in Neural Information Processing Systems*, 2023.
- [10] Francesco Mezzadri. How to generate random matrices from the classical compact groups. *Notices of the American Mathematical Society*, 54(5):592–604, 2007.
- [11] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *International Conference on Learning Representations*, 2018.
- [12] Deniz Oktay, Nick McGreivy, Joshua Aduol, Alex Beatson, and Ryan P. Adams. Randomized automatic differentiation. In *International Conference on Learning Representations*, 2021.
- [13] OpenAI. GSM8K: Grade school math 8k, main configuration. Hugging Face dataset repository, 2021. Revision 740312add88f781978c0658806c59bc2815b9866.
- [14] Kaan Ozkara, Tao Yu, and Youngsuk Park. Stochastic rounding for LLM training: Theory and practice. In *Proceedings of the 28th International Conference on Artificial Intelligence and Statistics*, volume 258 of *Proceedings of Machine Learning Research*, pages 4402–4410, 2025.
- [15] PyTorch contributors. AdamW. PyTorch 2.9 documentation, 2025.
- [16] PyTorch contributors. `torch.linalg.qr`. PyTorch 2.9 documentation, 2025.
- [17] Qwen Team. Qwen3-1.7B-Base: Model, configuration and tokenizer. Hugging Face model repository, 2025. Revision ea980cb0a6c2ae4b936e82123acc929f1cec04c1.
- [18] John Schulman and Thinking Machines Lab. LoRA without regret. Thinking Machines Lab, September 2025.
- [19] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
- [20] Yara Shamshoum, Nitzan Hodos, Yuval Sieradzki, and Assaf Schuster. CompAct: Compressed activations for memory-efficient LLM training. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 1511–1524, 2025.
- [21] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.
- [22] Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. In *Proceedings of the 35th International Conference on Machine Learning*, pages 4596–4604, 2018.
- [23] Thijs Vogels, Sai Praneeth Karimireddy, and Martin Jaggi. PowerSGD: Practical low-rank gradient compression for distributed optimization. In *Advances in Neural Information Processing Systems*, 2019.
- [24] An Yang et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.
- [25] Yang You, Igor Gitman, and Boris Ginsburg. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*, 2017.
- [26] Qiying Yu et al. DAPO: An open-source LLM reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.
- [27] Pedram Zamirai, Jian Zhang, Christopher R. Aberger, and Christopher De Sa. Revisiting BFloat16 training. *arXiv preprint arXiv:2010.06192*, 2020.
- [28] Longteng Zhang, Lin Zhang, Shaohuai Shi, Xiaowen Chu, and Bo Li. LoRA-FA: Efficient and effective low rank representation fine-tuning. *arXiv preprint arXiv:2308.03303*, 2023.
- [29] Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. GaLore: Memory-efficient LLM training by gradient low-rank projection. In *Proceedings of the 41st International Conference on Machine Learning*, 2024.

A Development trajectories

Table 11: Development results for every candidate. Each cell gives correct answers / truncated answers / mean completion tokens over 256 questions at the stated step. D is dense AdamW, L LoRA, S plain FP32 RGC, N normalized FP32 RGC and NB normalized BF16 RGC; suffix 1/2 is the lower/higher scale. Rows marked 8B or 14B are Qwen3-8B-Base and Qwen3-14B-Base; all others are Qwen3-1.7B-Base.

Candidate	Step 50	Step 100	Step 150	Step 200
D-1	234 / 1 / 253.8	231 / 0 / 215.3	223 / 0 / 239.9	230 / 0 / 302.5
D-2	182 / 36 / 447.5	176 / 3 / 439.6	190 / 0 / 282.4	128 / 60 / 637.5
L64-1	229 / 0 / 295.0	233 / 0 / 297.8	228 / 0 / 290.8	236 / 0 / 291.7
L64-2	230 / 0 / 209.1	231 / 1 / 205.8	227 / 2 / 223.8	221 / 1 / 236.1
L128-1	234 / 0 / 283.8	234 / 1 / 337.5	229 / 1 / 376.8	232 / 1 / 424.2
L128-2	221 / 4 / 193.2	210 / 18 / 317.3	217 / 14 / 481.0	216 / 4 / 468.9
L256-1	224 / 1 / 286.2	229 / 0 / 291.9	236 / 1 / 298.3	228 / 0 / 259.0
L256-2	224 / 3 / 232.2	84 / 147 / 735.5	225 / 1 / 262.3	209 / 10 / 291.8
S64-1	229 / 0 / 270.1	231 / 0 / 270.9	225 / 0 / 276.1	238 / 0 / 286.0
S64-2	224 / 2 / 307.8	226 / 1 / 271.7	235 / 1 / 285.5	233 / 0 / 309.3
S128-1	232 / 0 / 247.6	227 / 0 / 242.2	224 / 1 / 233.3	224 / 3 / 219.2
S128-2	223 / 2 / 274.4	218 / 1 / 247.5	210 / 4 / 193.4	224 / 0 / 214.3
S256-1	230 / 0 / 282.6	220 / 0 / 228.5	224 / 1 / 231.2	222 / 0 / 228.8
S256-2	187 / 3 / 162.1	0 / 256 / 1024.0	0 / 256 / 1024.0	0 / 256 / 1024.0
N64-1	235 / 0 / 263.8	225 / 0 / 290.7	232 / 0 / 295.9	227 / 0 / 282.7
N64-2	227 / 0 / 240.6	225 / 1 / 288.7	232 / 0 / 287.0	229 / 0 / 284.9
N128-1	227 / 1 / 265.8	229 / 1 / 271.8	228 / 0 / 281.1	234 / 0 / 295.6
N128-2	225 / 0 / 283.2	232 / 1 / 245.2	229 / 3 / 292.4	227 / 2 / 309.5
N256-1	228 / 0 / 324.7	227 / 0 / 334.0	234 / 2 / 348.9	235 / 2 / 343.5
N256-2	230 / 1 / 271.3	227 / 0 / 347.1	230 / 1 / 340.6	228 / 1 / 345.8
NB64-1	231 / 0 / 283.4	231 / 0 / 292.8	233 / 0 / 278.4	236 / 0 / 278.4
NB64-2	230 / 2 / 307.7	232 / 1 / 296.4	231 / 0 / 355.0	236 / 2 / 443.7
NB128-1	229 / 0 / 257.9	234 / 0 / 261.4	228 / 0 / 257.8	229 / 0 / 278.6
NB128-2	235 / 0 / 308.3	230 / 0 / 270.2	225 / 0 / 259.1	234 / 0 / 208.0
NB256-1	233 / 0 / 311.1	231 / 1 / 363.1	239 / 0 / 348.3	235 / 0 / 349.8
NB256-2	228 / 0 / 273.8	214 / 0 / 300.6	226 / 3 / 458.9	232 / 1 / 403.5
NB8-1	226 / 0 / 264.8	232 / 0 / 260.0	230 / 1 / 267.9	232 / 1 / 263.2
NB8-2	223 / 0 / 275.8	232 / 0 / 305.9	233 / 1 / 322.8	233 / 0 / 324.5
NB16-1	233 / 0 / 255.4	230 / 1 / 252.0	234 / 0 / 260.1	237 / 0 / 274.5
NB16-2	231 / 0 / 278.7	230 / 0 / 258.6	234 / 0 / 272.6	234 / 0 / 299.7
L16-1	222 / 1 / 275.1	224 / 1 / 269.9	233 / 0 / 267.8	233 / 0 / 270.5
L16-2	227 / 0 / 266.9	231 / 1 / 314.5	236 / 1 / 368.5	237 / 4 / 317.8
NB64 (8B)	242 / 0 / 290.7	243 / 1 / 291.8	243 / 0 / 305.0	245 / 1 / 309.9
L64 (8B)	245 / 0 / 296.6	246 / 0 / 296.8	248 / 0 / 301.3	246 / 0 / 314.5
NB64 (8B, second run)	246 / 0 / 293.0	241 / 0 / 294.8	244 / 1 / 298.7	245 / 0 / 308.4
NB128 (8B)	243 / 0 / 286.1	241 / 0 / 281.8	245 / 0 / 278.7	247 / 0 / 272.6
NB64 (8B, recipe)	246 / 0 / 298.0	246 / 0 / 303.8	245 / 0 / 304.1	246 / 0 / 312.7
L16 (8B, recipe)	244 / 0 / 291.8	246 / 0 / 293.0	244 / 0 / 286.9	248 / 0 / 290.2
NB64 (14B, recipe)	242 / 0 / 273.9	243 / 1 / 287.3	241 / 0 / 295.2	247 / 0 / 316.2