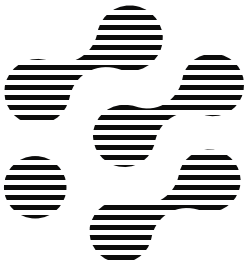

CAISTRO LABS • TECHNICAL REPORT • JULY 2026

Poros

póros, from Greek: passage, way through

Bit-Exact Large-Model Fine-Tuning on Consumer GPUs

Blockwise Residency for Frozen-Base QLoRA

Jeffrey Tigres · Caistro Labs Research

caistrolabs.com

NVIDIA RTX PRO 6000 Blackwell · RTX 3090 · RTX 5090 · A100 80GB · B300 · PyTorch 2.10 / 2.11 (primary) · CUDA 12.8

Poros[1]: Bit-Exact Large-Model Fine-Tuning on Consumer GPUs

Blockwise Residency for Frozen-Base QLoRA

Jeffrey Tigres
Caistro Labs Research · caistrolabs.com

July 2026

Abstract. Fine-tuning frontier-scale open models still often requires workstation-class GPU memory, even when only LoRA adapters are trained. We introduce **Poros**, a blockwise-residency method that brings **bit-exact 32B QLoRA fine-tuning into a 16 GB consumer-GPU memory budget**. Poros keeps LoRA adapters and optimizer state resident while streaming frozen quantized transformer blocks through the GPU during the forward pass and the rematerialized backward pass.

The core challenge is exactness: naive rematerialization can change stochastic recomputation, especially under dropout. Poros stores each block input and pre-forward RNG state, restores that state during backward rematerialization, and computes gradients only for adapter parameters and block inputs. We use *bit-exact* (resident parity) to mean elementwise equality with a resident QLoRA reference under the same seed, data order, optimizer state, hardware, and software stack; under those conditions Poros reproduces the resident optimization trajectory exactly.

Across Qwen2.5-7B, 32B, and 72B in 200-step QLoRA runs, the maximum absolute difference between Poros Blockwise and the resident reference is $0.00\text{e}{+00}$ for per-step loss and adapter weights; at 7B and 32B this extends to gradient norms and Adam optimizer state and holds without dropout, with dropout ($p{=}0.05$), and on real instruction-tuning data. At 32B, peak VRAM falls from **44.88 GB** to **11.61 GB**, allowing a 200-step run to complete within a **16 GB GPU-memory budget** with 4.39 GB of headroom; at 72B, peak VRAM falls from 81.57 GB to 18.87 GB. Parity is further validated across four architecture families (Qwen2.5, Qwen3, Gemma3, Gemma4) and five GPU platforms across consumer, workstation, and datacenter hardware, each exact at $0.00\text{e}{+00}$ within its own hardware/software stack; we do not claim cross-platform bitwise equality.

Poros reduces GPU residency, not total model storage: the frozen quantized base remains in host memory and is streamed over PCIe, trading VRAM for host RAM (roughly 4, 18, and 40 GB for the NF4 base at 7B, 32B, and 72B), bandwidth, and wall-clock time. Poros is slower than resident QLoRA; it exists for the memory budgets resident training cannot reach.

[1] *Poros (*póros*, from Greek):* passage*,* way through*. The method streams frozen base weights through the GPU block by block.*

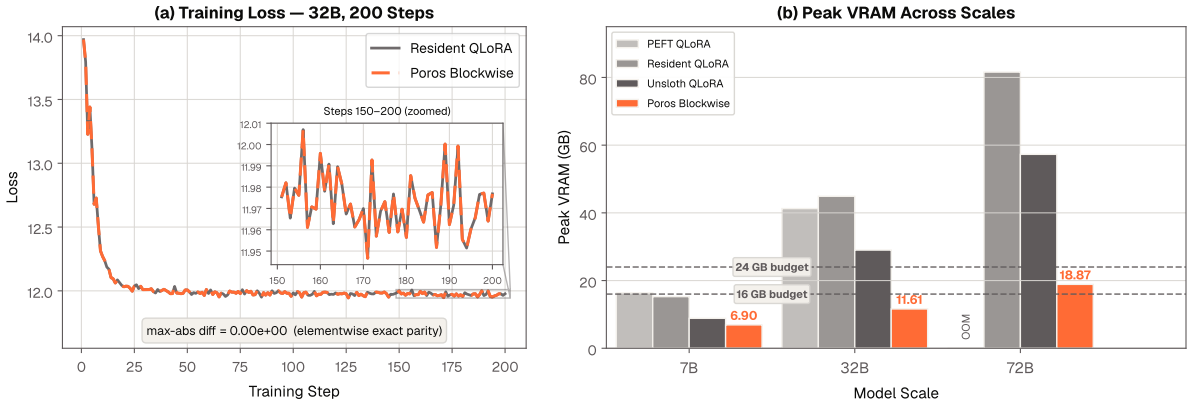

Figure 1. Poros results across scales. **(a)** Training loss for resident QLoRA reference and Poros Blockwise at 32B over 200 steps; the curves are indistinguishable (max $|\Delta| = 0.00$, elementwise exact parity). The benchmark tokens are uniformly random, so the loss plateau near $\ln(\text{vocab size}) \approx 11.9$ is the expected entropy floor for unpredictable tokens, not a stalled run; the content of the panel is the exact overlap of the two curves. **(b)** Peak VRAM at 7B, 32B, and 72B for PEFT QLoRA, resident QLoRA reference, Unsloth QLoRA, and Poros Blockwise. The "OOM" marker denotes PEFT QLoRA at 72B, which exceeds the 96 GB device and does not complete; all other bars are measured peaks. Poros Blockwise is the only method whose 32B training-step peak fits below the 16 GB consumer-card line and whose 72B training-step peak fits below the 24 GB consumer-card line.

1 Introduction

Parameter-efficient fine-tuning has become the default path for adapting large language models because full-model fine-tuning makes trainable parameters, optimizer state, and checkpoints scale prohibitively with model size. LoRA [1] freezes the pretrained model and inserts trainable low-rank updates into transformer layers; QLoRA [2] further reduces memory by backpropagating through a frozen 4-bit quantized base model into LoRA adapters. Yet QLoRA does not eliminate all memory pressure. Even when the trainable state is small, practical implementations often keep much of the frozen base model GPU-resident during training, and this residency cost grows with model size.

For frozen-base adapter tuning, the base model is read-only: only the adapter weights and their optimizer state are updated. **Poros** treats the frozen base as streamable state, not persistent GPU state. It materializes only the frozen block needed for the current computation, keeps adapters resident, and rematerializes blocks during backward when gradients are needed.

The hard part is equivalence. If a block is recomputed during backward, stochastic operations such as dropout may draw different random masks, changing gradients and breaking parity with resident training. PyTorch's checkpointing documentation makes the same point: recomputation can advance RNG state and break equivalence unless RNG state is stashed and restored, especially when dropout is present [5]. Poros addresses this by saving per-block RNG state before the original forward pass and restoring it during rematerialized backward. The result is a blockwise-residency execution schedule that preserves the resident QLoRA optimization trajectory exactly.

We introduce **Poros**, a systems method for frozen-base QLoRA that streams frozen quantized base blocks through the GPU while keeping trainable adapters resident. Poros does not change the model architecture, training objective, optimizer update, adapter parameterization, or update order. It changes only the residency schedule of frozen base weights. Across Qwen2.5-7B, 32B,

and 72B in 200-step training runs, Poros Blockwise matches the resident QLoRA reference with maximum absolute difference **0.00e+00** in per-step loss and adapter weights; at 7B and 32B this parity extends to gradient norms and optimizer state, and holds without dropout, with dropout at $p{=}0.05$, and on real instruction-tuning data (Alpaca), where train and evaluation loss curves overlap exactly in our measured runs.

At 32B scale, peak VRAM drops from **44.88 GB** for the resident reference to **11.61 GB** for Poros Blockwise. In same-configuration memory baselines, standard PEFT QLoRA requires **41.26 GB** peak VRAM and Unsloth QLoRA requires **29.00 GB**, while Poros Blockwise requires **11.61 GB**; Unsloth is reported as a memory and throughput baseline rather than an exact-parity target. Under a hard **16 GB memory budget**, Poros completes a 200-step 32B run at **11.61 GB** peak with **4.39 GB** of headroom and matches the unconstrained resident loss exactly. In practical terms, Poros moves 32B adapter fine-tuning out of the workstation-only regime and into the memory envelope of consumer GPUs. At 7B scale, Poros Blockwise reduces peak VRAM from **15.24 GB** to **6.90 GB** under the canonical protocol; refreshed RTX 3090 measurements show Poros and Unsloth are memory-competitive at this scale, with protocol-dependent ordering. The main advantage appears at 32B and 72B, where frozen-base residency dominates memory. Poros therefore makes an explicit tradeoff: it lowers memory substantially at the cost of additional wall-clock time, running at **1.10 s/step** at 32B versus **0.68 s/step** for the resident reference and **0.40 s/step** for Unsloth. Poros is slower; its purpose is to make otherwise-infeasible memory budgets feasible.

The claim of this paper: **for frozen-base LoRA/QLoRA training, blockwise residency can preserve the resident optimization trajectory exactly while moving large-model adaptation into a much lower VRAM regime**. In our measured setting, that shift is large enough to bring **32B QLoRA within a 16 GB memory budget** without observed degradation in training or evaluation loss relative to the resident reference.

Our contributions are:

- **Exact resident parity.** Poros Blockwise matches the resident QLoRA reference under matched RNG, with maximum absolute difference $0.00\text{e}{+00}$ across losses, adapter weights, gradient norms, and optimizer state.
- **A formal execution-equivalence view.** We identify the conditions under which blockwise rematerialization is equivalent to resident frozen-base adapter training: frozen base weights, resident adapters, matched data and initialization, delayed optimizer update, and per-block RNG restoration (Proposition 1).
- **Large VRAM reduction at 32B–72B.** Training-step peak VRAM falls from 44.88 GB to 11.61 GB at 32B and from 81.57 GB to 18.87 GB at 72B, while preserving the resident training trajectory (the one-time load is accounted separately, Table 1).
- **Generality across architectures and GPUs.** Exact parity ($0.00\text{e}{+00}$, `torch.equal`) holds across four model families (Qwen2.5, Qwen3, Gemma3, Gemma4—including hybrid sliding-window/global attention) and on five GPU platforms spanning consumer, workstation, and datacenter hardware (RTX 3090, RTX 5090, RTX PRO 6000, A100 80 GB, B300), each within its own hardware/software stack, with a same-attention-path three-method benchmark isolating the cost of residency (Sections 5.3 and 5.4).
- **Consumer-budget feasibility.** Under a hard 16 GB memory budget, Poros completes a full 200-step 32B QLoRA run with 4.39 GB of headroom.
- **Load-time residency on consumer cards.** A block-streamed NF4 load bounds load-time GPU residency to under 2 GB at every scale (1.63 GB at 72B)—byte-identical to the standard load, `torch.equal`/sha256-gated at 0.5B/7B/32B/72B across four families—bringing the one-time load, not just training, onto consumer cards; it ships as the default loader with

standard-loader fallback. A separate dense-BF16 bitwise runtime (Section B) extends the same exactness off the quantized path and remains experimental.

- **A clear memory–time tradeoff.** Poros exchanges wall-clock time for memory, creating a new feasibility point for large-model adaptation.

Claim. Poros is an execution-schedule change for frozen-base adapter training. Under matched seed, data order, optimizer state, and hardware/software stack, Poros Blockwise produces the same adapter gradients and optimizer updates as a resident QLoRA reference.

Tradeoff. Poros reduces peak GPU memory by replacing full-base residency with blockwise residency, but it increases wall-clock time and uses host memory plus PCIe bandwidth.

---

2 Related Work

Parameter-efficient fine-tuning. LoRA [1] introduced low-rank adapter matrices that are added to frozen pretrained weights, reducing the number of trainable parameters by orders of magnitude while matching full fine-tuning quality on many benchmarks. QLoRA [2] combined LoRA with 4-bit NormalFloat quantization of the frozen base model, enabling 65B fine-tuning on a single 48 GB GPU. Subsequent work has explored alternative adapter structures (AdaLoRA [13], DoRA [14]), rank allocation strategies, and quantization improvements, but the fundamental memory profile remains: peak VRAM is dominated by the frozen base model plus one layer's worth of activations, gradients, and adapter optimizer state. Poros does not change the adapter formulation; it changes *when* each layer's frozen weights need to be GPU-resident.

Memory-efficient training systems. Gradient checkpointing [6] trades compute for memory by discarding intermediate activations and recomputing them during the backward pass. ZeRO-Offload [7] and DeepSpeed-Inference [10] offload optimizer state and parameters to CPU memory, distributing the memory burden across the memory hierarchy; ZeRO-Infinity [8] extends offload to NVMe at extreme scale, and FlexGen [9] schedules weight, activation, and KV-cache placement across GPU, CPU, and disk for high-throughput *inference*. FSDP [11] shards parameters across GPUs in a multi-device setting, and FSDP+QLoRA [12] applies this to frozen-base adapter training—sharding an NF4 base across multiple consumer GPUs to fine-tune 70B-class models at home. These approaches distribute the model across devices and must handle gradient synchronization, mixed-device optimizer steps, or shard coordination. Poros operates in a narrower setting—frozen-base adapter training on a single GPU—and exploits the fact that the base model is read-only to avoid the complexity of distributed gradient management. The closest analogy is layer-wise CPU offloading, but Poros's contribution is the exactness guarantee: by saving and restoring per-block RNG state through a custom `autograd.Function`, blockwise rematerialization produces the identical forward and backward computation, including stochastic operations such as dropout.

Single-GPU heterogeneous-memory fine-tuning. A recent line of work fine-tunes large models on a single commodity GPU by staging the model through scarce GPU memory. LoHan [18] manages holistic offload traffic with active-gradient offloading and traffic-aware activation swapping; SlideFormer [19] keeps a fixed-size sliding window of active layers in a reused GPU cache queue; LSP-Offload [20] schedules layer-wise CPU–GPU communication and compresses the transferred gradients with learned sparse projectors; MeBP [21] combines gradient checkpointing with lazy weight loading and memory-mapped activations for on-device fine-tuning;

and ChunkFT [22] reformulates full-parameter fine-tuning around a dynamically activated working set. These systems optimize throughput or feasibility for full-parameter or approximate updates; where they alter the gradient (e.g. learned sparse projection) or the working set, they do not claim bit-for-bit equivalence to an unconstrained reference. Poros specializes the same heterogeneous-memory problem to frozen-base QLoRA: it leaves the optimization problem unchanged, keeps adapters and optimizer state resident, streams only the read-only quantized base, and verifies $0.00\text{e}{+00}$ resident-trajectory parity rather than reporting only end-task quality.

Unsloth and practical PEFT acceleration. Unsloth [3] takes a different approach to efficient adapter training: it rewrites the backward pass in hand-tuned Triton kernels, fuses operations to reduce memory fragmentation, and applies smart gradient offloading to reduce peak VRAM. These are *computational* optimizations that make the resident training path faster and leaner. In our 32B measurements, Unsloth achieves 29.00 GB peak VRAM and 0.40 s/step. Poros Blockwise operates on a different axis: rather than optimizing the resident computation, it restructures *which* parameters are GPU-resident at any given time. The two approaches are complementary; the released Poros implementation already includes a fused-LoRA execution path inside each block (Section 3.6), and deeper kernel-level fusion within attention and MLP subpaths remains possible future work.

Blockwise learning methods. DiffusionBlocks [4] trains residual networks blockwise by interpreting network blocks as denoising steps in a score-based diffusion process and assigning blocks local noise ranges and denoising objectives. Poros is orthogonal. It does not introduce local denoising losses, noise conditioning, or independently optimized block objectives. Instead, Poros keeps the original frozen-base QLoRA learning problem fixed and changes only the residency schedule of frozen base weights. In short, DiffusionBlocks is a blockwise *learning* method, while Poros is a blockwise *residency* method.

Activation checkpointing and RNG correctness. PyTorch's `checkpoint` utility [5] saves memory by recomputing activations during the backward pass, but it documents a known subtlety: recomputation advances the global RNG state, potentially breaking reproducibility when dropout or other stochastic layers are present. PyTorch addresses this by optionally preserving per-segment RNG state. Poros faces the same challenge at block granularity and solves it identically: each block's RNG state is captured before its forward pass and restored before rematerialization. Our empirical results confirm this is sufficient for exact parity, including under dropout at $p{=}0.05$.

3 Method

We consider frozen-base adapter training for transformer language models. Let the pretrained model consist of L transformer layers with frozen weights $\{W^{(1)}, \ldots, W^{(L)}\}$. Adapter tuning augments selected linear projections in each layer with trainable low-rank parameters, while the base weights remain fixed throughout training. In the QLoRA setting, the frozen base weights are stored in quantized form and dequantized as needed during computation, but they are still not updated by the optimizer. The only trainable state is therefore the collection of adapter parameters and their optimizer buffers.

The central observation behind Poros is that frozen base weights do not need to remain GPU-resident between the forward and backward pass. Only the adapter parameters must persist on

device as trainable state. Poros exploits this by partitioning the model into contiguous *blocks* of transformer layers and streaming those blocks between CPU and GPU during training.

3.1 Blockwise execution

Let the L layers be partitioned into K contiguous blocks $\mathcal{B}_1, \ldots, \mathcal{B}_K$, where each block contains m layers and $K = \lceil L/m \rceil$. During the forward pass, Poros materializes only one block of frozen base weights on GPU at a time. For block $\mathcal{B}_k$, the block manager:

1. transfers the frozen weights for $\mathcal{B}_k$ from CPU to GPU,
2. executes the forward computation for the layers in $\mathcal{B}_k$,
3. retains the block output and the minimal metadata required for rematerialization, and
4. releases the GPU-resident copy of the frozen weights of $\mathcal{B}_k$ once the block is no longer needed.

The adapter parameters remain GPU-resident across the full training run. As a result, peak GPU memory is governed primarily by: (i) the currently resident block of frozen weights, (ii) the trainable adapter parameters and optimizer state, and (iii) the activations needed for the current rematerialization boundary, rather than the full frozen model.

This differs from standard resident QLoRA, where all frozen layers are available on GPU throughout training. Poros does not change the training objective, model architecture, or optimizer update rule; it changes only the residency schedule of the frozen base model.

3.2 Exact rematerialized backward

Naively recomputing a block during the backward pass is not sufficient for exact parity with resident training, because stochastic operations such as dropout may draw different random masks on recomputation. Poros therefore implements blockwise rematerialization through a custom `autograd.Function` that saves the information needed to reconstruct the exact forward computation of each block.

For each block, the forward pass stores:

- the block input tensor,
- references to the frozen block weights,
- references to the trainable adapter parameters,
- the per-block RNG state immediately before executing the block,
- lightweight metadata describing the block structure.

During the backward pass, Poros restores the saved RNG state for the block, rematerializes the block forward computation, and invokes automatic differentiation on the rematerialized computation graph. Because the base model is frozen, gradients are required only for the adapter parameters and for the block input passed to the preceding block. The frozen weights are treated as read-only tensors and never accumulate gradients or optimizer state.

Under matched initialization, matched data order, matched optimizer state, and matched RNG restoration, this procedure reproduces the same per-block computation as resident training. Empirically, this yields exact parity between Poros Blockwise and resident QLoRA reference in per-step loss, adapter weights, gradient norms, and optimizer state.

Algorithm 1 gives pseudocode for one Poros training step, showing the RNG save/restore mechanism that guarantees exact rematerialization.

Algorithm 1 Poros Blockwise: one training step.

Require: Blocks $\mathcal{B}_1, \ldots, \mathcal{B}_K$ (frozen, on CPU); adapters θ (on GPU); batch x
Ensure: Gradient update to θ
$h_0 \leftarrow \text{Embed}(x)$ ▷ GPU-resident embedding
Forward:
for $k = 1, \ldots, K$ **do**
  $s_k^{\text{cpu}} \leftarrow$ GetRngState(cpu)
  $s_k^{\text{cuda}} \leftarrow$ GetRngState(cuda)
  LoadToGpu($\mathcal{B}_k$) ▷ Async pinned transfer
  Save(h_{k-1}, s_k^{cpu}, s_k^{cuda})
  $h_k \leftarrow f_k(h_{k-1}; W_k, \theta_k)$ ▷ no_grad; rematerialized in backward
  ReleaseGpu($\mathcal{B}_k$) ▷ CPU copy is authoritative; no D2H copy
end for
$\mathcal{L} \leftarrow \text{LmHead}(h_K, \text{labels})$ ▷ GPU-resident head
Backward (triggered by $\mathcal{L}$.backward()):
for $k = K, \ldots, 1$ **do**
  LoadToGpu($\mathcal{B}_k$)
  ForkRng; SetRngState(s_k^{cpu}, s_k^{cuda})
  $\hat{h}_k \leftarrow f_k(h_{k-1}; W_k, \theta_k)$ ▷ Rematerialize; RNG-identical
  $\nabla_{h_{k-1}}, \nabla_{\theta_k} \leftarrow \text{AutoGrad}(\hat{h}_k, \partial\mathcal{L}/\partial h_k)$
  θ_k.grad $+= \nabla_{\theta_k}$
  ReleaseGpu($\mathcal{B}_k$); UnforkRng
end for
OptimizerStep(θ) ▷ Update adapters only

Proposition 1 (Resident-parity of Poros Blockwise). Consider a frozen-base adapter model partitioned into K blocks. Let W_k denote the frozen weights of block k, θ_k the adapter parameters inside block k, and h_k the hidden state after block k. Assume:

1. The frozen weights W_k are identical between resident and blockwise execution and are never updated.
2. The adapter parameters θ and optimizer state are initialized identically and remain resident until the optimizer step.
3. Both executions see the same data order and same initial RNG state.
4. For every block k, Poros stores the RNG state immediately before the original forward computation and restores it during backward rematerialization.
5. The optimizer update is applied once after all block gradients have been accumulated.
6. Each block's forward and backward computation is a deterministic, side-effect-free function of its inputs, frozen weights, adapters, and restored RNG state—no nondeterministic kernels, and no mutable global state (caches, hooks) that persists across the rematerialized passes.

Then, on the same hardware and software stack, Poros Blockwise computes the same adapter gradients and the same optimizer update as resident QLoRA for that training step. By induction over training steps, the loss sequence, adapter weights, gradient norms, and optimizer state remain identical for all steps.

Proof sketch. In resident execution, each block computes $h_k = f_k(h_{k-1}; W_k, \theta_k, r_k)$, where r_k denotes the random draws used by stochastic operations inside the block. Poros executes the same forward function with the same W_k, θ_k, h_{k-1}, and stored RNG state r_k, so the forward ac-

tivations match. During backward, Poros rematerializes f_k under the saved RNG state, so the recomputed local graph is identical to the resident graph for that block. Since W_k is frozen, gradients are required only for θ_k and h_{k-1}. Automatic differentiation over the identical local graph therefore yields the same partial derivatives as resident training. Processing blocks in reverse order propagates the same upstream gradient through each identical block graph. After all blocks accumulate gradients, the optimizer sees the same θ gradients and the same optimizer state, so it produces the same adapter update. The argument repeats at the next step because both executions now have identical adapter parameters, optimizer state, and RNG state. In the implementation, the rematerialized backward is executed inside an isolated RNG context (`torch.random.fork_rng`), so restoring a block-local RNG state does not advance the global RNG stream differently from the resident path.

3.3 Asynchronous prefetch and offload

A purely synchronous implementation would incur excessive transfer latency, since each block would need to be copied before its computation begins. Poros therefore overlaps communication with computation through an asynchronous prefetch pipeline.

While block $\mathcal{B}_k$ is executing on GPU, the block manager prefetches the frozen weights for block $\mathcal{B}_{k+1}$ into pinned host memory buffers and schedules the corresponding host-to-device transfer. Likewise, blocks that are no longer needed in resident form are released asynchronously. This pipelining reduces GPU idle time and turns the dominant systems tradeoff into one of bandwidth and transfer frequency rather than repeated full-model residency.

The block size m controls this tradeoff. Smaller blocks lower peak VRAM because fewer frozen layers are resident at once, but they increase the number of transfers and therefore increase wall-clock time. Larger blocks move the system closer to resident training: transfer overhead falls, but memory usage rises. In the limit where the block size equals the full model depth, Poros collapses to the resident regime.

3.4 Memory and compute tradeoff

Let M_{base} denote the frozen base-model memory footprint, $M_{\text{block}}(m)$ the memory footprint of one resident block of size m, M_{adapter} the adapter parameter memory, M_{opt} the optimizer state for those adapters, and M_{act} the activation memory at the rematerialization boundary. Standard resident QLoRA requires memory proportional to

$$M_{\text{resident}} \approx M_{\text{base}} + M_{\text{adapter}} + M_{\text{opt}} + M_{\text{act}},$$

whereas Poros instead requires memory proportional to

$$M_{\text{Poros}} \approx M_{\text{block}}(m) + M_{\text{adapter}} + M_{\text{opt}} + M_{\text{act}}.$$

Because $M_{\text{block}}(m) \ll M_{\text{base}}$ for sufficiently small m, peak VRAM can be reduced substantially, especially at large model scales where the frozen base dominates memory. The cost is additional transfer and rematerialization overhead, which increases wall-clock time per step relative to resident training.

Proposition 2 (Peak-memory bound). Let $M_{\text{base}} = \sum_k M(W_k)$ be the memory footprint of the frozen base model, $M_{\text{block}}(m) = \max_k M(W_{k:k+m-1})$ the largest resident frozen block under block size m, M_{adapter} the resident adapter memory, M_{opt} the adapter optimizer state, M_{act} the activation / rematerialization-boundary memory, and ϵ runtime overhead. A resident QLoRA implementation requires peak memory

$$M_{\text{resident}} \approx M_{\text{base}} + M_{\text{adapter}} + M_{\text{opt}} + M_{\text{act}} + \epsilon.$$

Because Poros prefetches the next block while the current block executes, more than one frozen block can be live on the GPU at the memory peak. Let $M_{\text{live}}(m, p)$ denote the maximum frozen-weight memory resident on GPU under block size m and prefetch depth p, including the active block together with any prefetched or transfer-buffered blocks. Poros Blockwise then requires peak memory bounded by

$$M_{\text{Poros}} \leq M_{\text{live}}(m, p) + M_{\text{adapter}} + M_{\text{opt}} + M_{\text{act}} + \epsilon'.$$

For strict single-block execution, $M_{\text{live}}(m, 0) = M_{\text{block}}(m)$. With one GPU-resident prefetch buffer, $M_{\text{live}}(m, 1)$ is bounded by approximately $2\, M_{\text{block}}(m)$, though allocator behavior and transfer scheduling determine the measured peak. When frozen base weights dominate memory and blocks are roughly uniform, $M_{\text{block}}(m) \approx (m/L)\, M_{\text{base}}$, so even with a single prefetch buffer the leading-order frozen-weight residency term falls by approximately $1 - 2m/L$. The measured savings are smaller than this leading-order bound because adapters, optimizer state, activations, allocator fragmentation, and runtime buffers remain resident. This bound explains why Poros helps more at larger model scales: as M_{base} grows relative to adapter and activation memory, replacing full-base residency with single-block residency removes a larger fraction of peak VRAM.

3.5 What Poros changes, and what it does not

Poros is a systems method, not a new optimization algorithm. It does not change:

- the LoRA or QLoRA parameterization,
- the training objective,
- the optimizer update rule,
- the order of parameter updates.

Instead, Poros changes only the execution schedule of the frozen base model. Its purpose is to move frozen-base adapter training into a lower VRAM regime while preserving the resident optimization trajectory exactly.

3.6 Fused LoRA execution

The released Poros implementation optionally uses a fused LoRA execution path inside each rematerialized block to reduce per-linear autograd overhead. A standard PEFT LoRA-wrapped linear produces roughly four autograd nodes (dropout, A matmul, B matmul, scale-and-add); a typical transformer layer has seven such linears (Q, K, V, O, gate, up, down), yielding ~ 28 autograd nodes per layer, so on fast accelerators the per-node Python and CUDA-launch overhead dominates matmul time. The fused path collapses these into a small number of grouped autograd nodes per layer (one for Q/K/V sharing the saved input tensor, one for O, and one for the gate/up/down SwiGLU MLP), and forward and backward outputs are bit-identical to the unfused path on the same input. This optimization does not change LoRA math, optimizer state, gradient targets, or the blockwise residency schedule. It only reduces execution overhead within the same resident-parity training path.

3.7 Implementation details

The exact-parity guarantee relies on several implementation choices that we document here for reproducibility.

RNG state management. For each block, the custom `autograd.Function` captures both the CPU RNG state and the CUDA RNG state for the active device immediately before the block's forward computation. During backward rematerialization, these states are restored inside a `torch.random.fork_rng()` context, which isolates the restored state from the global RNG stream. This ensures that stochastic operations such as dropout draw identical masks on both the original forward and the rematerialized recomputation.

Gradient routing. Because the frozen base weights are never updated, gradients are computed only with respect to (i) the block input tensor (for backpropagation to the preceding block) and (ii) the trainable adapter parameters within the current block. This is implemented with a direct autograd gradient call (`autograd.grad`) over an explicit target list, avoiding unnecessary gradient computation or accumulation for frozen parameters.

Memory transfers. Block weights are stored in pinned host memory to enable asynchronous `cuda.Stream` transfers. The block manager coordinates host-to-device transfers and GPU-allocation release on dedicated CUDA streams, overlapping transfers with computation where possible.

Software environment. Primary experiments use PyTorch 2.10, CUDA 12.8, `bitsandbytes` for NF4 quantization, `transformers` 5.5, and Python 3.12; extended 32B experiments on the PRO 6000 use PyTorch 2.11, and the experimental BF16 gates of Section B additionally run on PyTorch 2.7.1+cu126 and 2.5.1+cu121 stacks. For benchmark hygiene, we additionally enable deterministic algorithms (with `warn_only=True`) and pin math SDPA to stabilize kernel and backend selection across runs. The exact Resident–Blockwise match itself does not depend on these flags: a released mode-sweep gate re-runs the parity check with deterministic algorithms disabled, default (fast) SDPA backend selection, TF32 matmuls, bf16 autocast, and `torch.compile`, and parity remains $0.00\text{e}{+00}$ in every mode across four architecture families up to 31B (each mode runs in a fresh subprocess).[2] The benchmark runs nevertheless keep the flags pinned as measurement hygiene: backend dispatch heuristics can in principle select different kernels under the two arms' different allocator states, and the sweep shows that in the tested configurations they do not. The match itself follows from matched initialization, matched data order, and per-block RNG capture/restore during rematerialized backward.

Precision of "exact". Throughout this paper, "exact resident parity" means that the maximum absolute difference between Poros Blockwise and resident QLoRA reference is $0.00\text{e}{+00}$ under IEEE 754 float32 arithmetic on the same hardware, same software stack, and same random seed—that is, elementwise exact equality of the reported tensors and scalar losses. We verify this using `torch.allclose()` with `atol=0, rtol=0` for tensor comparisons and direct floating-point equality for scalar losses; where the text says `torch.equal`, that primitive is literally what

[2] Code and run artifacts are released with the Poros repository; each reported table is backed by checked run manifests and SHA-256 records.

the gate executes. We do not claim bitwise reproducibility across different hardware platforms, driver versions, or CUDA toolkit versions, as these may employ different floating-point instruction orderings.

4 Experimental Setup

Experiments are conducted on five hardware platforms: an NVIDIA RTX PRO 6000 Blackwell Server Edition GPU (96 GB, PCIe Gen5) as the primary platform for canonical measurements and scaling sweeps, an NVIDIA RTX 3090 (24 GB, PCIe Gen4) for consumer-hardware validation and same-card baseline comparisons, and—for the multi-architecture validation campaign (Sections 5.3 and 5.4)—an NVIDIA A100 80 GB PCIe (Ampere, sm_80, PCIe Gen4) and an NVIDIA B300 SXM6 (Blackwell, sm_103). An additional consumer-Blackwell datapoint uses an NVIDIA RTX 5090 (32 GB, sm_120, PCIe Gen5) under a single-pass validation protocol (Section A). Primary experiments use PyTorch 2.10, CUDA 12.8, `bitsandbytes` for NF4 quantization, and `transformers` 5.5; extended 32B experiments on the PRO 6000 use PyTorch 2.11. Peak VRAM is measured via `torch.cuda.max_memory_allocated()`, reset before each run. Throughput is wall-clock elapsed time divided by the number of training steps, including all transfer and rematerialization overhead.

4.1 Models and adapter configuration

We evaluate on three model scales: Qwen2.5-7B ($L{=}28$ layers), Qwen2.5-32B ($L{=}64$ layers), and Qwen2.5-72B ($L{=}80$ layers); the multi-architecture campaign (Section 5.3) additionally evaluates Qwen3-32B, Gemma3-27B, and Gemma4-31B. All models are loaded in 4-bit NormalFloat (NF4) quantization with double quantization enabled. All methods use rank-16 LoRA adapters ($\alpha{=}32$) applied to the same seven projection matrices per layer: `q_proj`, `k_proj`, `v_proj`, `o_proj`, `gate_proj`, `up_proj`, and `down_proj`. Unless otherwise noted, dropout is 0.0; we test dropout at $p{=}0.05$ separately to exercise the RNG parity mechanism.

4.2 Training protocol

All runs use AdamW with no weight decay. The extended-audit harness (Section 5.2) is configured at learning rate 2×10^{-4} with a constant schedule; runs produced by the released pipeline use its defaults—cosine schedule with linear warmup at learning rate 2×10^{-5}—except where a run states otherwise (the consumer-Blackwell real-data run and the downstream adapter of Section 5.11 set 2×10^{-4} with 10-step warmup explicitly, and accumulate gradients over 8 micro-batches per optimizer step, the pipeline default). Per-step learning rates and token counts are recorded in every released run's event stream. The learning-rate setting does not enter the parity, VRAM, or throughput claims: both parity arms always share one optimizer configuration, and memory and step time are rate-invariant. Sequence length is 512 tokens, batch size is 1, and gradient accumulation is 1 step for the canonical parity, VRAM, and throughput runs; the consumer-Blackwell real-data and downstream runs instead use the pipeline default of 8 (their reported step counts are optimizer steps). For parity experiments, both the resident and blockwise paths are initialized with the same random seed and trained on identical synthetic data drawn from a fixed-seed generator with the same vocabulary. For the real-dataset experiment, we use the Alpaca instruction-tuning dataset [16] (52k examples), with a 48k/4k train/eval split, and evaluate every 50 steps. The extended 1000-step audit (Section 5.2) instead evaluates a fixed 200-example held-out subset every 200 steps, reducing evaluation cost while preserving a deterministic parity check; both protocols are reported where used.

Table 1. Resource envelope (measured). Train-peak VRAM under the canonical protocol; load-peak VRAM is the one-time NF4 load high-water mark under the standard loader versus the block-streamed loader; host RSS is peak process `VmHWM`. The standard load materializes the full quantized base on-GPU and OOMs consumer cards at 72B; the streamed loader bounds it under 2 GB at every scale (0.81 GB at 32B, 1.63 GB at 72B).

	Train-peak	Load-peak VRAM		Host RSS	
Model	VRAM	standard	streamed	standard	streamed
7B	6.90 GB	5.30 GB	0.27 GB	15.3 GB	12.1 GB
32B	11.61 GB	18.16 GB	0.81 GB	62.0 GB	34.7 GB
72B	18.87 GB	38.80 GB[†]	1.63 GB	136.5 GB	48.2 GB

Load-peak VRAM is byte-identical-gated standard-vs-streamed at all four scales (`torch.equal`/sha256 over packed weights and quant constants). [†]The 38.80 GB standard 72B load was measured on an A100 80GB, where it fits; it exceeds every consumer card—measured OOM at 23.2 GiB on an RTX 3090 (24 GB) and 30.5 GiB on an RTX 5090 (32 GB)—whereas the 1.63 GB streamed load and the 18.87 GB training peak each fall under 24 GB (the end-to-end process peak is set by the startup self-check probe at $\sim$30.5 GB; see the consumer-GPU paragraph below).

Poros Blockwise uses a default block size of $m{=}4$ layers. The block manager uses pinned host memory and asynchronous `cuda.Stream` transfers for prefetch and offload.

Host-memory accounting. Poros reduces GPU residency rather than total model storage: the frozen NF4-quantized base lives in host memory together with the pinned transfer buffers. The quantized base occupies approximately 0.55 bytes per frozen parameter (4-bit packed weights plus quantization constants under double quantization), i.e. on the order of 4 GB at 7B, 18 GB at 32B, and 40 GB at 72B. The block manager allocates this host copy in pinned memory—a requirement for asynchronous `cuda.Stream` transfers—plus transient per-block transfer slabs, so minimum host RAM is at least the NF4 base footprint above plus ordinary process overhead. Measured peak process RSS (kernel `VmHWM`) in the validation reruns is 15.1–15.3 GB at 7B, 62.0 GB at 32B, and 136.5 GB at 72B; the peak is dominated by the one-time checkpoint-load and quantization transient (the loader streams the bf16 shards while building the NF4 base), so provisioned host RAM should follow these measured peaks rather than the steady-state NF4 footprint alone. This host-RAM requirement is separate from the reported peak allocated VRAM: Poros targets machines where host memory is plentiful relative to GPU memory.

Bringing 72B fine-tuning fully onto a consumer GPU. Blockwise residency bounds GPU memory during *training*; the one-time NF4 load is a separate concern, and on the standard path it is the binding constraint at the largest scale. That path materializes the entire quantized base on the GPU before the block manager evicts it to host memory—a fixed $\sim$40 GB at 72B—so a 72B load alone exceeds consumer cards (measured: 23.2 GiB allocated before OOM on an RTX 3090, 30.5 GiB on an RTX 5090). The released implementation closes this with a block-streamed load (`nf4_load_strategy="block_streamed"`) that bounds load-time GPU residency exactly as training does: each tensor is staged alone, quantized on-GPU, and evicted, holding the load peak under 2 GB at every scale (0.81 GB at 32B, 1.63 GB at 72B, versus 18.16/38.80 GB standard). Crucially, the streamed load is *byte-identical* to the standard load—every packed weight and quantization constant compared with `torch.equal` (sha256 at 72B)—verified at 0.5B, 7B, 32B, and 72B, and across the Qwen2.5, Qwen3, Gemma3, and Gemma4 families, with the full training-parity gate re-passing at $0.00\text{e}{+00}$ under it. Because the streamed weights are bit-identical to the standard-loaded weights, for which resident-vs-blockwise training parity is $0.00\text{e}{+00}$ at 72B

in the canonical 200-step runs, the streamed load trains the same bit-exact model: **Qwen2.5-72B loads and trains end-to-end on a single 32 GB RTX 5090** at an 18.13 GB training-step peak (matching the canonical 18.87 GB workstation-card figure to within stack variation), the training-loss trajectory agreeing to $\sim 10^{-5}$. The directly-measured 72B streamed load peak is 1.63 GB—versus the 38.80 GB standard load that OOMs every consumer card—so the one-time load no longer gates large-model fine-tuning: the 72B streamed load (1.63 GB) and training (18.13–18.87 GB) peaks each fall under 24 GB. The end-to-end run additionally carries the mandatory startup self-check, whose resident probe sets a ~ 30.5 GB process high-water mark, so 72B fine-tunes end-to-end on a 32 GB card today; reaching a 24 GB card requires streaming that startup probe (roadmap). The streamed load also *lowers* peak host RSS—12.1 GB at 7B, 34.7 GB at 32B, and 48.2 GB at 72B, versus 15.3 / 62.0 / 136.5 GB on the standard path—so a 72B fine-tune fits a 64 GB-RAM host. Every figure above is measured; the loader is byte-identity-gated at all four scales and ships as the default with standard-loader fallback.

4.3 Baselines

We compare against four configurations:

- **resident QLoRA reference.** Standard QLoRA training with all frozen layers GPU-resident. This is the reference trajectory against which Poros Blockwise is tested for exact parity.
- **PEFT QLoRA.** The Hugging Face `peft` [15] library with identical adapter configuration and `transformers.Trainer`. This represents the standard open-source QLoRA training path.
- **Unsloth QLoRA.** Unsloth [3] with hand-tuned Triton kernels and fused backward passes. For reliable VRAM measurement, the Unsloth baseline is run in a fresh standalone process with no prior CUDA allocations, because CUDA allocator fragmentation from prior model loads inflates reported peak memory. The canonical PRO 6000 Unsloth entries use a 60-step $\times$ 3 warmed protocol (their losses are therefore marked n/c); the same-attention-path A100 campaign (Table 4) re-measures Unsloth under the full 200-step $\times$ 3 protocol.
- **Accelerate CPU-Offload QLoRA.** Hugging Face `accelerate` [17] with `max_memory` constrained to 12 GB GPU, placing overflow layers on CPU. Gradient checkpointing is enabled. This represents the standard offload-based approach to training under a tight GPU budget.

All baselines use the same model, quantization, adapter rank, target modules, learning rate, and data. PEFT, Unsloth, and Accelerate use their own optimizer and backward-pass implementations; only the resident reference and Poros Blockwise share the same training loop.

4.4 Evaluation protocol

Parity. We compare Poros Blockwise against resident QLoRA reference across five metrics at every training step: (1) per-step training loss, (2) per-tensor adapter weight values, (3) per-step gradient norms, (4) Adam optimizer state (`exp_avg` and `exp_avg_sq` for each adapter tensor), and (5) evaluation loss. Parity is reported as the maximum absolute difference across all steps and all tensors. Tensor-level parity is verified using `torch.allclose()` with `atol=0` and `rtol=0`, the strictest possible tolerance; scalar losses are compared by exact floating-point equality.

Table 2. Blockwise vs. Resident parity: maximum absolute difference across all training steps. The 7B/32B/72B synthetic $d{=}0$ rows take loss and adapter-weight bit-identity directly from the canonical 200-step post-fix campaign, which gates loss and weights at all three scales; gradient-norm and optimizer-state bit-identity is gated directly by the full-surface NF4 parity gates at 7B and 32B (all four surfaces at $0.00\text{e}{+00}$); at 72B these two surfaces are not separately gated but are consistent with the bit-identical per-step weight trajectory recorded there. The dropout, Alpaca, and 1000-step rows are from earlier sub-experiments at the same seed/config and confirm parity holds across dropout, real data, and extended training. All directly gated entries are exactly 0.00, confirming exact resident-parity optimization trajectories. †At 72B, gradient-norm and optimizer-state parity is inferred from the recorded bit-identical weight trajectory, not separately gated.

Configuration	Loss	Weights	Grad Norms	Optimizer	VRAM ↓
7B, $d{=}0.0$, synthetic, 200-step	0.00e+00	0.00e+00	0.00e+00	0.00e+00	54.7%
7B, $d{=}0.05$, synthetic	0.00e+00	0.00e+00	0.00e+00	0.00e+00	59.7%
32B, $d{=}0.0$, synthetic, 200-step	0.00e+00	0.00e+00	0.00e+00	0.00e+00	74.1%
32B, $d{=}0.05$, synthetic	0.00e+00	0.00e+00	0.00e+00	0.00e+00	76.7%
32B, $d{=}0.0$, Alpaca	0.00e+00	0.00e+00	0.00e+00	0.00e+00	74.2%
7B, 1000-step Alpaca+eval	0.00e+00	0.00e+00	0.00e+00	0.00e+00	57.6%
32B, 1000-step Alpaca+eval	0.00e+00	0.00e+00	0.00e+00	0.00e+00	76.3%
72B, $d{=}0.0$, synthetic, 200-step	0.00e+00	0.00e+00	*implied*†	*implied*†	76.9%

VRAM. Peak allocated GPU memory is recorded via `torch.cuda.max_memory_allocated()` over the full training run. For the 16 GB constraint test, a blocker tensor is allocated to restrict available VRAM to 16 GB before training begins; the model's peak memory is then the total peak minus the blocker size.

Throughput. Wall-clock time is measured over the full training run (200 steps for canonical runs, 10 steps for ablation sweeps) and divided by step count. Transfer and synchronization overhead is included.

5 Results

5.1 Poros Blockwise exactly matches resident training

We begin with the central question of this paper: whether blockwise offloading changes the optimization trajectory of frozen-base QLoRA training. Across all parity experiments, the answer is no. Poros Blockwise matches resident QLoRA reference exactly under matched RNG.

At 32B scale, over 200 training steps on synthetic data with dropout disabled, the maximum absolute difference between the resident reference and Poros Blockwise is $0.00\text{e}{+00}$ for per-step training loss, adapter weights, gradient norms, and Adam optimizer state. The same result holds when dropout is enabled at $p = 0.05$, showing that the saved-and-restored per-block RNG state is sufficient to preserve exact equivalence even in the presence of stochastic operations. On real instruction-tuning data (Alpaca), the parity result also holds: train and evaluation loss curves overlap exactly over 200 steps, with maximum absolute difference $0.00\text{e}{+00}$.

The same resident-parity behavior appears at 7B scale. Over 200-step runs, Poros Blockwise and resident QLoRA reference again produce identical losses and identical final adapter weights. Multi-seed experiments across three seeds show no deviation from exact parity: for every seed, the maximum observed difference between resident and blockwise execution is $0.00\text{e}{+00}$. Poros does not approximate resident QLoRA; it reproduces it.

5.2 Exact parity holds at extended training with real held-out evaluation

To confirm that parity is not limited to short runs or synthetic data, we extend Alpaca instruction-tuning training to 1000 steps at both 7B and 32B scale, evaluating on a fixed 200-example held-out subset every 200 steps (a cheaper protocol than the canonical 4k-split/50-step evaluation of Section 4.2, chosen to keep the 1000-step audit tractable). At both scales, Poros Blockwise and resident QLoRA reference produce identical results across all metrics: maximum absolute difference is $0.00\text{e}{+00}$ for per-step training loss, adapter weights, gradient norms, optimizer state, and held-out evaluation loss at every checkpoint. The step times reported below come from this audit harness, which interleaves the periodic held-out evaluation passes and full per-step parity instrumentation (capture and comparison of adapter weights, gradient norms, and optimizer state) into the same run, and which executed on shared cloud infrastructure; they characterize the audited run and are not comparable to the instrumentation-free canonical throughput protocol of Table 8, which is the throughput reference. A dedicated-hardware rerun below quantifies the decomposition.

At 7B (RTX 3090), both paths reach a final evaluation loss of 1.640 (perplexity 5.16), with the evaluation loss curve declining identically from 1.664 at step 200 to 1.654 at step 800. Poros Blockwise uses 6.47 GB peak VRAM versus 15.27 GB for the resident reference—a 57.6% reduction—at a throughput cost of 1.45 s/step versus 0.79 s/step ($1.8\times$ slower).

At 32B (PRO 6000), both paths reach a final evaluation loss of 1.596 (perplexity 4.93), with the evaluation loss curve declining from 1.620 at step 200 to 1.602 at step 800. Poros Blockwise uses 10.65 GB peak VRAM versus 44.88 GB for the resident reference—a 76.3% reduction—at a throughput cost of 2.62 s/step versus 0.93 s/step ($2.8\times$ slower).

To separate the audit's accounting from training throughput, we reran the audit configuration on dedicated hardware of the same GPU classes and the same pinned stack (PyTorch 2.11.0+cu128; 7B on an RTX 3090, 32B on an RTX PRO 6000 Blackwell), instrumenting train-step time separately from the evaluation passes. Train-only step time is 1.129 s/step at 7B (wall 1.556 s/step with the four held-out evaluation passes, measured at 423 s total, amortized in) and 1.242 s/step at 32B (wall 1.711 s/step; evaluation passes 465 s total). The gap between the audited step times above and the canonical reference of Table 8 is therefore accounted for by the interleaved evaluation and parity instrumentation plus the original audit's shared-infrastructure placement, not by the streaming mechanism. The rerun uses the corrected data pipeline (padding masked out of the loss) at the released-pipeline default learning-rate schedule, so its held-out losses are not numerically comparable to the original audit's; they decline smoothly ($1.480 \to 1.429$ at 7B, perplexity 4.18; $1.436 \to 1.386$ at 32B, perplexity 4.00), confirming healthy held-out learning curves under the corrected collator.

The same rerun campaign also closed the two remaining parity-scope holes: sequential gradient accumulation at $G \in \{2, 4\}$ passes the full training-parity gate (losses, adapter gradients, adapter weights, and optimizer state compared with `torch.equal` over 200 optimizer steps) at $0.00\text{e}{+00}$ at both 7B and 32B, and a 32B cell at sequence length 4096 passes the same gate.

These results demonstrate that Poros Blockwise produces the same trained model as the resident path at both scales, not just the same training loss. The identical held-out perplexity at $5\times$ the canonical run length rules out gradual divergence: the method produces a working fine-tuned model, not just a matching loss curve.

Table 3. Exact parity: max $|\Delta| = 0.00\text{e}{+00}$ (`torch.equal` on per-step loss and adapter gradients) for every model family, with the platform(s) on which each resident-vs-blockwise gate was recorded. NF4, rank-16 LoRA, block size 4. Coverage spans five GPU platforms, consumer to datacenter.

Model	Arch (`model_type`)	Max $\|\Delta\|$	Platform(s) gated
Qwen2.5-7B	`qwen2`	$0.00\text{e}{+00}$	RTX 5090, RTX 3090
Qwen2.5-32B	`qwen2`	$0.00\text{e}{+00}$	A100, B300, PRO 6000, RTX 5090
Qwen2.5-72B	`qwen2`	$0.00\text{e}{+00}$	B300
Qwen3-32B	`qwen3`	$0.00\text{e}{+00}$	B300, RTX 5090
Gemma3-27B	`gemma3`	$0.00\text{e}{+00}$	RTX 5090
Gemma4-31B	`gemma4`	$0.00\text{e}{+00}$	RTX 5090

5.3 Exact parity generalizes across model families and GPU generations

The parity results above use a single model family (Qwen2.5). To test whether blockwise residency is architecture-specific, we extended the validation to four transformer families spanning two vendors and two attention designs: Qwen2.5-32B (`qwen2`), Qwen3-32B (`qwen3`), Gemma3-27B (`gemma3`; multimodal text stack), and Gemma4-31B (`gemma4`; hybrid sliding-window + global attention with per-layer rotary configuration). The Gemma 4 architecture is the stress case: its layers receive heterogeneous per-layer keyword arguments (attention masks, rotary embeddings) that a naive rematerialization would replay incorrectly; Poros captures and replays these per layer.

Each (architecture, GPU) cell runs the exact-parity gate: a resident forward/backward and a blockwise forward/backward from identical state, comparing loss and adapter gradients with `torch.equal`; equivalently, `torch.allclose` with `atol=0`, `rtol=0` also passes. Every model family attains exactly $0.00\text{e}{+00}$, each gated on the platform(s) on which it was run (Table 3); coverage spans five GPU platforms from consumer to datacenter, parity exact within each hardware/software stack (not claimed across stacks).

Within each architecture/GPU cell, Poros and the resident reference match bit-for-bit. This shows that the mechanism generalizes across the tested architecture families and hardware stacks—it is not a claim of bitwise equality between different GPUs—and supports the central claim mechanically: Poros changes only the residency schedule, so any kernel that is deterministic under a fixed execution path produces identical results regardless of where the frozen weights were staged.

The A100, B300, RTX PRO 6000, and RTX 3090 gates were recorded under the standard loader; the RTX 5090 gate and the Gemma3/Gemma4/Qwen3 family gates under the block-streamed loader, whose load is byte-identical to the standard load (Table 1). All are full resident-vs-blockwise comparisons at the listed scale, exact within each stack, with raw gate records released for every cell.

5.4 Cost of residency under a single attention path

The throughput comparisons elsewhere in this paper time Poros on its deterministic math-SDPA path while baselines use their default fused attention, which conflates the cost of residency with the cost of determinism. To isolate the residency cost, the multi-architecture campaign benchmarks all three methods—Poros Blockwise, the resident QLoRA reference, and Unsloth—on the *same* default fast attention path, under the official protocol (200 steps $\times$ 3 passes per cell, mean$\pm$std, NF4, rank-16 LoRA, seq_len 512, batch 1, deterministic synthetic data, seed 42) on the A100 80 GB PCIe (Table 4).

Table 4. Same-attention-path official benchmark, NVIDIA A100 80GB PCIe (Gen4). 200 steps × 3 passes; s/step is mean±std across passes; VRAM is peak allocated during training.

	s/step			Peak VRAM (GB)		
Model	Poros	Resident	Unsloth	Poros	Resident	Unsloth
Qwen2.5-32B	2.049±0.009	0.766	0.909	**10.95**	34.72	20.82
Qwen3-32B	2.183±0.004	0.816	0.929	**11.03**	36.00	20.77
Gemma3-27B	1.893±0.005	0.784	0.937	**9.95**	33.17	22.06
Gemma4-31B	2.254±0.006	0.989	1.026	**10.65**	38.75	20.57

Under this clean comparison, Poros reduces peak training VRAM by 68.4–72.5% versus the resident reference (and 46.9–54.9% versus Unsloth) at a 2.28–2.68× per-step slowdown versus resident. Every Poros peak is below 11.1 GB—all four 27–32B models fine-tune within a 16 GB training budget on this card.

The wall-clock cost of blockwise residency is hardware-dependent, because the numerator (PCIe streaming + rematerialization) and the denominator (resident step time) scale with different hardware properties. In our measurements the slowdown versus the resident reference is approximately 2.1–2.4× on consumer / RTX PRO 6000-class GPUs, 2.28–2.68× on the A100 80 GB PCIe Gen4 (same attention path for all methods, Table 4), and 3.5–4.3× on a B300 SXM6 (per-cell records in the released artifacts)—the faster the GPU executes the resident step, the more the fixed PCIe transfer cost dominates the ratio. We therefore report the slowdown per platform and do not claim a universal factor.

5.5 Poros moves 32B QLoRA into a substantially lower VRAM regime

The practical impact of Poros is most apparent at 32B scale (Table 5). Standard PEFT QLoRA requires 41.26 GB peak VRAM, and resident QLoRA reference requires 44.88 GB. Among the practical baselines measured under our protocol, Unsloth is the strongest resident-path memory baseline, reducing this requirement to 29.00 GB; Poros targets a different axis—changing frozen-weight residency rather than optimizing the resident path—and reduces peak VRAM further to 11.61 GB.

Canonical headline number. Unless otherwise stated, the headline 32B result is Qwen2.5-32B NF4, rank-16 LoRA, seq_len = 512, batch = 1, block size 4, 200 steps under the RTX PRO 6000 protocol: **11.61 GB peak VRAM**. Other 32B values reported in this paper differ because they use different hardware, attention paths, fused-LoRA refreshes, datasets, or validation protocols.

Relative to resident QLoRA reference, this is a 74.1% reduction in peak VRAM. Relative to PEFT QLoRA, it is a 71.9% reduction. Relative to Unsloth, Poros Blockwise reduces peak VRAM by 60.0%, from 29.00 GB to 11.61 GB. Under the canonical memory-baseline protocol, Poros also reduces 7B memory relative to PEFT, the resident reference, and the measured Unsloth baseline—Poros Blockwise requires 6.90 GB, compared with 15.24 GB for the resident reference, 16.35 GB for PEFT, and 8.84 GB for Unsloth—though the refreshed RTX 3090 comparison in Table 10 shows that 7B ordering is protocol-dependent. The main advantage appears at 32B and 72B.

Poros changes the residency class of frozen-base adapter training rather than optimizing the resident path, and the benefit grows with model size: the frozen base dominates memory increasingly strongly as scale increases, while Poros keeps only a single block of frozen weights resident at a time.

Table 5. Canonical results for QLoRA fine-tuning across methods and model scales (RTX PRO 6000 Blackwell).

Scale	Method	Peak VRAM (GB)	s/step	Final Loss	Fits 16 GB	Steps
7B	PEFT QLoRA	16.35	0.16	12.0268	—	200
7B	resident QLoRA reference	15.24	0.19	12.0277	✓	200
7B	Unsloth QLoRA	8.84	0.10	*n/c*	✓	**60**
7B	**Poros Blockwise**	**6.90**	0.30	12.0277	✓	200
32B	PEFT QLoRA	41.26	0.62	11.9785	—	200
32B	resident QLoRA reference	44.88	0.68	11.9768	—	200
32B	Unsloth QLoRA	29.00	0.40	*n/c*	—	**60**
32B	**Poros Blockwise**	**11.61**	1.10	11.9768	✓	200
72B	resident QLoRA reference	81.57	1.32	12.0174	—	200
72B	Unsloth QLoRA	57.28	0.81	*n/c*	—	**60**
72B	**Poros Blockwise**	**18.87**	2.18	12.0174	—	200

Protocol: Poros and PEFT entries are 200-step warmed runs at rank-16 LoRA, NF4, seq_len=512, batch=1, deterministic (math SDPA, `torch.use_deterministic_algorithms(True)`). Poros Blockwise matches the resident QLoRA reference exactly: maximum per-step loss and LoRA-weight differences are 0.0 at 7B, 32B, and 72B. Unsloth entries are 60-step $\times$ 3-run warmed-mean measurements in a fresh standalone process and serve as a memory baseline only; Unsloth losses are marked *n/c* (not comparable) for that reason. At 32B, Poros Blockwise's 11.61 GB peak sits 4.39 GB below the 16 GB single-GPU consumer budget.

5.6 Poros fits 32B QLoRA within a 16 GB memory budget

The canonical 32B post-fix peak of **11.61 GB** sits 4.39 GB below the 16 GB single-GPU consumer budget. We tested this directly: in the blocker-constrained run of Section 4.4, with available VRAM restricted to 16 GB before training begins, the 32B Poros configuration completed all 200 steps at 11.61 GB peak allocated VRAM, leaving 4.39 GB of measured headroom. Because Poros Blockwise matches the resident QLoRA reference exactly (maximum loss and weight differences of 0.0 over all 200 training steps), the trained adapter is identical to the unconstrained resident run, so the 16 GB feasibility margin does not trade off training quality.

5.7 Same-card consumer-hardware comparison

To ensure apples-to-apples baseline comparisons on consumer hardware, we run Poros Blockwise, Unsloth QLoRA, PEFT QLoRA, and Accelerate CPU-offload QLoRA (with `max_memory` constrained to 12 GB GPU and gradient checkpointing enabled) on the same RTX 3090 card under identical hyperparameters (Table 6).

At 7B scale, Poros Blockwise uses 6.90 GB versus 9.71 GB for Accelerate Offload, 8.66 GB for Unsloth, and 16.37 GB for PEFT—a 58% reduction over PEFT, 29% over Offload, and 20% over Unsloth. The throughput cost is 0.87 s/step versus 0.73 s/step for Offload and ~ 0.52 s/step for both Unsloth and PEFT (1.2–$1.7\times$ slower).

At 32B scale, Poros Blockwise trains at 11.61 GB peak. Unsloth fits at 22.29 GB—barely within the 24 GB budget and only with `use_gradient_checkpointing="unsloth"` (without this setting Unsloth OOMs on the forward pass at ~ 23 GB)—while both PEFT and Accelerate Offload exceed memory entirely. Poros reduces VRAM by 48% relative to Unsloth; the other two baselines are infeasible on this hardware. The Accelerate failure is structural rather than a tuning issue: hook-based offload addresses *weight placement*—overflow layers are paged onto the GPU for their

Table 6. Same-card comparison on consumer hardware (RTX 3090, 24 GB).

Scale	Method	VRAM (GB)	s/step	Loss
7B	PEFT QLoRA	16.37	0.51	12.027
7B	Accel. Offload	9.71	0.73	12.027
7B	Unsloth QLoRA	8.66	0.53	12.027
7B	**Poros Blockwise**	**6.90**	0.87	12.028†
32B	PEFT QLoRA		*OOM*	
32B	Accel. Offload		*OOM*	
32B	Unsloth QLoRA	22.29	2.07	11.979
32B	**Poros Blockwise**	**11.61**	3.21	11.977†

Protocol: rank-16 LoRA, NF4, seq_len=512, 200 steps, identical data and seeds, deterministic kernels. Accelerate Offload uses `max_memory` with gradient checkpointing; Unsloth 32B requires `use_gradient_checkpointing="unsloth"` to fit within 24 GB (without it, it OOMs at $\sim$23 GB). Poros peak VRAM matches the primary-platform Blackwell measurement (Table 5) to the reported precision. Poros saves 58% vs. PEFT, 29% vs. Offload, and 20% vs. Unsloth at 7B; 48% vs. Unsloth at 32B (PEFT and Offload infeasible). †Final loss after 200 steps; exact Resident=Blockwise parity verified on primary platform (Table 2). Cross-hardware loss differs from Blackwell canonical by $\leq 4\times10^{-4}$ due to sm_86 vs. sm_100 kernel dispatch; the optimizer trajectories match to four decimals.

forward computation—but the backward pass remains a standard autograd backward over the same graph. There is no rematerialized, block-bounded backward, so the GPU working set during backward (saved activations and re-loaded layers) is not confined to a fixed window and grows with model scale until it exceeds the card. Poros bounds GPU residency to one block plus prefetch in both the forward and the rematerialized backward, which is why the same 32B run succeeds at 11.61 GB. Throughput is 3.21 s/step versus 2.07 s/step for Unsloth ($1.55\times$ slower). The 11.61 GB 3090 peak matches the Blackwell canonical measurement (Table 5) to the reported precision: Poros's peak VRAM is primarily determined by model architecture, block size, sequence length, and LoRA rank, and was nearly identical across the two tested GPUs under matched protocol—a direct consequence of the block-size-bounded resident set.

These same-card measurements confirm the primary-platform trend: Poros's memory advantage grows with model size and holds against all four baseline categories (standard PEFT, kernel-optimized Unsloth, and CPU-offload), and is not an artifact of measuring baselines on different hardware or under different allocator states.

Benchmark protocol variants. Two additional consumer-hardware campaigns—a follow-up RTX 3090 run exercising the fused-LoRA path with a fresh-process Unsloth measurement, and a single-pass RTX 5090 validation of the 32B configuration—use measurement protocols that differ from the canonical one above. To keep a single main-line comparison protocol, we report them separately in Section A (Tables 10 and 11). Their headline outcomes: under the fused-LoRA fair-v2 protocol Poros is faster and the speed gap to Unsloth narrows to $\sim1.4\times$, and the RTX 5090 reproduces the 32B run at 11.02 GB peak VRAM. Across every card measured in this paper—consumer Ampere (RTX 3090), consumer Blackwell (RTX 5090), workstation Blackwell (RTX PRO 6000), and datacenter parts (A100, B300)—the same default configuration ran without platform-specific tuning, with peak VRAM determined by model, block size, and sequence length rather than by the GPU.

Table 7. Effect of block size on peak VRAM for 32B QLoRA. All configurations with block size ≤ 8 fit within a 16 GB budget. "Ablation elapsed" is the wall-clock time of the fixed-length block-size sweep used for this row, not a per-step measurement.

Block Size	Peak VRAM (GB)	Ablation elapsed (s)	Fits 16 GB
1	10.48	40.7	✓
2	10.73	19.9	✓
4	11.61	19.5	✓
8	14.01	16.9	✓
16	18.67	16.6	—

5.8 Block size provides a direct memory–throughput control

Poros exposes block size as a simple systems knob controlling the tradeoff between peak VRAM and wall-clock time. Table 7 shows the 32B block-size sweep. With block size $m = 1$, peak VRAM is 10.48 GB. Increasing to $m = 2$ yields 10.73 GB, and the canonical $m = 4$ setting yields 11.61 GB. At $m = 8$, peak VRAM rises to 14.01 GB, still within a 16 GB budget. At $m = 16$, peak VRAM rises to 18.67 GB and no longer fits.

This sweep confirms the expected monotonic behavior: smaller blocks lower memory by reducing the amount of frozen base state resident on GPU at any moment, while larger blocks reduce transfer frequency and move the system closer to resident execution. In practice, all block sizes from 1 to 8 fit within 16 GB at 32B scale, giving users a practical knob to exchange wall-clock time for memory according to the hardware budget available.

5.9 The memory savings come with a clear wall-clock cost

Poros improves memory, not throughput. At 32B scale on the PRO 6000 (this section's measurement platform), resident QLoRA reference runs at 0.68 s/step, Unsloth at 0.40 s/step, and Poros Blockwise at 1.10 s/step, corresponding to a $1.62\times$ slowdown relative to resident QLoRA reference on this card; the slowdown is hardware-dependent (see Section 5.4). At 7B, Poros is memory-competitive with Unsloth, with protocol-dependent ordering; the main advantage appears at larger scales, where frozen-base residency dominates memory. In all cases Poros runs more slowly per step.

This is the central tradeoff exposed by Poros. The method exchanges wall-clock time for a much lower memory regime. For workloads already well within memory budget, resident training or kernel-optimized baselines such as Unsloth remain attractive. For workloads near or beyond the hardware memory limit, however, Poros changes feasibility in a way that faster resident baselines cannot: at 32B, it is the only method in our comparison that completes training within a 16 GB budget.

5.10 Systems characterization of the block-transfer pipeline

To understand where the wall-clock overhead originates, we profile the block-transfer pipeline for the 7B model on the RTX 3090 (PCIe Gen4 $\times$16).

Table 8. Memory–throughput tradeoff (RTX PRO 6000 Blackwell, seq_len=512, 200 steps). Poros Blockwise reduces VRAM at the cost of increased wall-clock time due to CPU↔GPU block transfers; forward-prefetch on a dedicated CUDA stream limits the slowdown to $\approx 1.6\times$ at all scales *on this card*—the slowdown is hardware-dependent (2.28–2.68× on the A100 same-attention path, 3.5–4.2× on B300; Section 5.4). Slowdown is relative to resident QLoRA reference. Poros Blockwise matches resident QLoRA reference exactly (max_loss_diff $= 0.0$ over all 200 steps and full LoRA adapter state).

Scale	Method	Peak VRAM (GB)	s/step	tok/s	Final Loss	Slowdown
7B	PEFT QLoRA	16.35	0.16	3191	12.0268	—
7B	resident QLoRA reference	15.24	0.19	2715	12.0277	—
7B	Unsloth QLoRA*	8.84	0.10	5018	n/c	—
7B	**Poros Blockwise**	**6.90**	0.30	1719	12.0277	1.58×
32B	PEFT QLoRA	41.26	0.62	819	11.9785	—
32B	resident QLoRA reference	44.88	0.68	751	11.9768	—
32B	Unsloth QLoRA*	29.00	0.40	1283	n/c	—
32B	**Poros Blockwise**	**11.61**	1.10	465	11.9768	1.62×
72B	resident QLoRA reference	81.57	1.32	386	12.0174	—
72B	Unsloth QLoRA*	57.28	0.81	634	n/c	—
72B	**Poros Blockwise**	**18.87**	2.18	235	12.0174	1.64×

Poros and PEFT entries are 200-step warmed runs of the canonical campaign. *Unsloth entries are 60-step × 3-run warmed-mean measurements (fresh process per run); reported as a memory and throughput baseline. Final loss is marked n/c (not comparable) because the 60-step protocol differs from the 200-step canonical entries.

Transfer bandwidth. At 7B on the RTX 3090 (PCIe Gen4), each block (4 layers, 466 MB) transfers host-to-device in 22.4 ms at 20.9 GB/s. At 32B on the PRO 6000 (PCIe Gen5), each block (4 layers, 976 MB) also transfers in 22.4 ms—at 43.6 GB/s. The per-block latency is identical because the Gen5 bus doubles bandwidth while the 32B blocks double in size. Device-to-host time is negligible ($<$7 ms) because frozen weights are never modified on GPU—the CPU copy is the authoritative source, and "offload" amounts to releasing the GPU allocation.

Latency breakdown. One streaming pass transfers all K blocks once: $\sim$157 ms for 7 blocks at 7B (RTX 3090) and $\sim$358 ms for 16 blocks at 32B (PRO 6000), derived from the measured 22.4 ms per-block transfer. A full Poros training step schedules *two* such passes: each block is loaded once during the original forward pass and loaded again during backward rematerialization (Algorithm 1); blocks are released between phases rather than retained. The raw scheduled H2D volume per step is therefore approximately two streaming passes: $\sim$314 ms against an 856 ms mean step time at 7B ($\sim$37 %), and $\sim$716 ms against an 1101 ms mean step time at 32B ($\sim$65 %), under the canonical 200-step protocol at seq_len = 512 (Table 9). This raw transfer time should not be read as stall time: prefetch overlaps H2D copies with the adjacent block's compute on a dedicated CUDA stream, and the measured wall-clock step times already include the realized overlap.

Table 9. Sequence-length scaling for Poros Blockwise at 7B (RTX 3090, block_size=4, rank-16 LoRA, 200 training steps, deterministic kernels, math SDPA only—identical protocol to Table 6). The seq_len=512 row matches the canonical 7B Blockwise peak (6.90 GB) reported in Table 6. VRAM grows sub-linearly because frozen block weights are the dominant memory term at single-block residency: doubling seq_len adds ~ 0.24 GB at low context lengths and ~ 1.24 GB only at the 512→1024 step.

seq_len	ms/step	Peak VRAM (GB)	ΔVRAM vs. 128
128	573	6.13	—
256	589	6.37	+0.24
512	856	6.90	+0.77
1024	1525	8.13	+2.00

Sequence-length scaling. Table 9 shows how step latency and peak VRAM scale with sequence length at 7B (RTX 3090, canonical 200-step deterministic protocol matching Table 6). VRAM grows sub-linearly: doubling from 128 to 256 adds only 0.24 GB, while doubling from 512 to 1024 adds 1.23 GB. The dominant memory term is the frozen block weights (466 MB per block), which are constant across sequence lengths. Even at seq_len = 1024, 7B blockwise peak VRAM is 8.13 GB—well under a 12 GB consumer-card budget.

At 32B (PRO 6000) under the same canonical 200-step deterministic protocol, the sub-linear pattern holds: seq_len 128 uses 10.43 GB (871 ms/step), 256 uses 10.80 GB (914 ms/step), 512 uses 11.61 GB (1101 ms/step), and 1024 uses 13.48 GB (1908 ms/step). The seq_len=512 row matches the canonical 32B Blockwise peak (11.609 GB) reported in Table 5. Latency scales roughly linearly with sequence length past 256 at both scales, consistent with compute-bound behavior once transfer is saturated.

5.11 Downstream capability of a Poros-trained adapter

Because the parity gates establish that a Poros-trained adapter is elementwise identical to the adapter the resident QLoRA reference would produce, downstream benchmarks characterize the underlying QLoRA recipe rather than Poros itself. As external validation we nevertheless trained one Qwen2.5-32B NF4 adapter end-to-end with Poros Blockwise on a dedicated RTX PRO 6000 Blackwell (OpenAssistant/oasst1 [25], 100 steps, seq_len 512, rank-16 LoRA, learning rate 2×10^{-4} with 10-step warmup, padding masked out of the loss) and evaluated base versus base+adapter with the LM Evaluation Harness [26] (v0.4.12, bf16 inference; zero-shot MMLU [23] on a fixed 32-question-per-subtask subset, 1,824 questions total; 5-shot GSM8K [24] on a fixed first-64 subset). The Poros-trained adapter preserves base-model capability: MMLU accuracy is 81.6% with the adapter versus 83.0% for the base model (≈ 1.1 combined standard errors), and GSM8K does not degrade (strict exact-match 87.5% with the adapter versus 81.2% base; flexible 87.5% versus 87.5%; differences within one standard error). An earlier version of this check, trained before the padding-mask correction of Section 4.2, showed a large apparent GSM8K drop ($79.7\% \rightarrow 50.0\%$ strict); the corrected rerun shows that drop was an artifact of the contaminated training signal (an adapter trained largely on EOS-run prediction breaks few-shot completion formatting), not a property of short chat-style fine-tuning. Both campaigns' raw harness files are released, and the base rows reproduce across them (MMLU 83.0% both; GSM8K strict within one standard error).

Experimental dense-BF16 runtime. The released system also includes an experimental dense-BF16 streaming runtime (`bf16_bitwise`); because it is not the supported NF4 path and has very different host-memory and throughput characteristics, we report its parity gates and measured costs separately in Section B.

5.12 Summary

The experiments show three things. First, Poros Blockwise preserves the resident QLoRA reference optimization trajectory exactly, including under dropout and on real instruction-tuning data. Second, the memory savings are large enough to change which model scales fit within a fixed GPU budget, especially at 32B. Third, this benefit is purchased with additional wall-clock time rather than any observed degradation in resident-parity training behavior.

Poros provides a new memory–time tradeoff point for frozen-base adapter training: 32B QLoRA within a 16 GB memory budget, with exact resident-parity optimization.

6 Discussion and Limitations

Poros is a systems result with one narrow claim. We do not propose a new optimization objective, adapter parameterization, or optimizer. Instead, we show that for frozen-base LoRA/QLoRA training, the execution schedule of the frozen model can be restructured around blockwise residency while preserving the resident QLoRA optimization trajectory exactly under matched RNG. The practical value of that restructuring is a substantially lower peak-VRAM regime, especially at larger model scales where frozen-base residency dominates memory.

Poros is most useful in the *feasibility-limited* regime, not the *throughput-limited* one. When a model already fits comfortably in GPU memory, resident training or kernel-optimized baselines such as Unsloth remain attractive because they achieve lower wall-clock time per step. Poros instead targets the setting where memory, not arithmetic throughput, is the primary bottleneck. In this regime, moving from 44.88 GB or 29.00 GB peak VRAM to 11.61 GB at 32B scale changes which hardware budgets can support training at all.

Our results also suggest that the benefit of Poros grows with model size. At 7B, Poros is memory-competitive with Unsloth, with protocol-dependent ordering; the main advantage appears at larger model scales, where frozen-base residency dominates memory and the gap becomes substantially larger. This scaling pattern is consistent with the mechanism of the method: the larger the frozen base, the more beneficial it becomes to replace full-model residency with single-block residency. We therefore view Poros less as a direct competitor to kernel-level acceleration methods and more as a complementary memory-systems technique. The released Poros implementation already includes a fused LoRA path (Section 3.6), and further kernel fusion inside attention and MLP subpaths could narrow the throughput gap without changing the residency argument.

Released system. Poros is released as a pip-installable package with a Hugging Face ecosystem drop-in: a single call to `poros.prepare(model)` auto-detects the architecture and adapter and converts an existing PEFT/Trainer pipeline to blockwise residency (the lower-level `patch_hf_model` entry point takes an explicit block size), and a managed `PorosTrainer` path is provided as well. Correctness is enforced operationally: the package maintains an explicit registry of validated architectures (an architecture enters the registry only after passing the $0.00\text{e}{+00}$ parity gate of Section 5.3), training on an unvalidated architecture is refused by default, and every run begins with a mandatory one-step resident-versus-blockwise self-check

compared with `torch.equal`. The official benchmark protocol (200 steps $\times$ 3 passes, fixed seeds, deterministic synthetic data) ships in the repository together with the raw per-cell measurement artifacts for the multi-architecture campaign, so those results are independently reproducible and auditable. That said, the current study has several limitations.

Scope of exactness. The exact-parity claim in this paper applies only to **Poros Blockwise versus the resident QLoRA reference**. We show that these two execution modes produce identical losses, adapter weights, gradient norms, and optimizer states under matched RNG. We do *not* claim tensor-level equivalence between Poros and external frameworks such as PEFT or Unsloth, which use different training implementations and may differ in kernel fusion, optimizer realization, or numerical execution order. Those baselines are included as practical memory and throughput references, not as exact-parity targets.

Hardware coverage. Experiments span five platforms—RTX PRO 6000 Blackwell (96 GB, Gen5), RTX 3090 (24 GB, Gen4), RTX 5090 (32 GB, Gen5, single-pass validation), A100 80 GB PCIe (Gen4), and B300 SXM6—covering three GPU generations and two PCIe generations. On the RTX 3090, Poros Blockwise, Unsloth, and PEFT were measured on the same card under identical hyperparameters (Table 6). Differences in host-memory speed, NUMA topology, and driver behavior may still affect precise throughput on other systems; the parity result, by contrast, has held bit-for-bit on every platform tested (Table 3).

Throughput cost. Poros meaningfully increases wall-clock time per step because it replaces full-model residency with repeated transfer and rematerialization. At 32B scale the slowdown versus the resident QLoRA reference ranges from $1.62\times$ (PRO 6000, deterministic path) through 2.28–2.68$\times$ (A100, same attention path for all methods) to 3.5–4.3$\times$ (B300), depending on the platform (Section 5.4). This tradeoff is central to the method rather than an incidental implementation artifact. Although asynchronous prefetching and pinned-memory transfers reduce idle time, the current system is explicitly optimized for memory reduction rather than speed.

Model and task coverage. We evaluate four architecture families (Qwen2.5, Qwen3, Gemma3, Gemma4) at 7–72B within the parity protocol, and validate real-data behavior on Alpaca instruction tuning. Downstream benchmark coverage is limited to the MMLU and GSM8K capability checks of Section 5.11; since the blockwise adapter is elementwise identical to the resident one, such benchmarks characterize the underlying QLoRA recipe rather than Poros, but broader recipe-tuned evaluations remain useful external validation. Mixture-of-experts routing under blockwise residency is unvalidated and remains future work. Longer-context settings may also increase the share of memory devoted to activations relative to frozen weights, changing the precise benefit of blockwise residency.

Single-GPU setting. Poros is designed and evaluated for frozen-base adapter training on a single GPU. We do not study interactions with data parallelism, tensor parallelism, FSDP, or ZeRO-style sharding. Extending blockwise residency to distributed settings is an important direction for future work, but introduces additional concerns around synchronization, communication overlap, and checkpoint consistency that fall outside the scope of this paper.

Measurement protocol. Peak VRAM is measured with the CUDA `max_memory_allocated` counter, which tracks allocated memory rather than all driver-visible memory effects. We mitigate allocator-history issues by measuring Unsloth in a fresh standalone process and by resetting peak statistics before each run, but memory accounting can still vary across drivers and runtimes. For this reason, we emphasize the relative differences observed under our measurement protocol rather than claiming universal absolute numbers.

None of these limitations touches the central result: for frozen-base adapter training, exact resident-parity optimization and a far lower VRAM peak are compatible. In the regime where GPU memory is the dominant constraint, that is enough to move 32B QLoRA into a 16 GB memory budget while preserving the resident QLoRA reference trajectory exactly.

Future work. There are several promising directions beyond the current study. First, further kernel-level optimization remains possible—including deeper fusion inside attention and MLP subpaths beyond the fused-LoRA path already present in the released implementation (Section 3.6)—to further reduce the throughput gap to Unsloth; the block-major gradient-accumulation schedule of the experimental BF16 runtime (Section B), which cut host-to-device traffic by $\sim 2.4\times$ at accumulation depth 2, is the clearest such lever for the main NF4 path, where sequential gradient accumulation is parity-gated at $G \in \{2, 4\}$ (Section 5.2) but the block-major schedule itself is not yet ported. Second, a broader hardware sweep across PCIe generations, consumer GPUs, and multi-GPU setups would clarify where blockwise residency is most advantageous. Third, extending the method to longer-context training and additional model families would test how the balance between activation memory and frozen-weight memory changes the attainable savings. Finally, the same residency principle may apply beyond LoRA/QLoRA to other frozen-backbone adaptation schemes in which only a small trainable subspace changes over time.

7 Conclusion

We introduced Poros, a systems method for frozen-base LoRA/QLoRA training that replaces full-model residency with blockwise residency while preserving the resident QLoRA optimization trajectory exactly under matched RNG. Across 7B, 32B, and 72B experiments, Poros Blockwise matches its resident QLoRA reference with maximum absolute difference $0.00\text{e}{+00}$ in loss and adapter weights; at 7B and 32B this extends to gradient norms and optimizer state and holds with dropout and on real instruction-tuning data as well. At the same time, Poros substantially lowers peak memory: at 32B scale, peak VRAM falls from 44.88 GB for the resident QLoRA reference to 11.61 GB for Poros Blockwise, and a 200-step run completes within a 16 GB memory budget with 4.39 GB of headroom. The same exactness generalizes: across four architecture families (Qwen2.5, Qwen3, Gemma3, Gemma4) and five tested GPU platforms (RTX 3090, RTX 5090, RTX PRO 6000, A100, B300), every parity gate passes at $0.00\text{e}{+00}$ within its own hardware/software stack, and under a single shared attention path the four 27–32B models train in 9.9–11.0 GB on an A100 80 GB—68–73% below the resident reference. Relative to practical baselines, Poros also reduces memory below both standard PEFT QLoRA and Unsloth in our measured setup, though at a clear, hardware-dependent wall-clock cost.

The contribution of this work is a new point in the memory–time tradeoff for large-model adaptation: exact resident-parity optimization at a fraction of the resident VRAM. In the memory-limited regime, that puts 32B QLoRA inside a 16 GB budget. We expect future work to extend blockwise residency with deeper kernel-level optimization beyond the fused LoRA path already in the released Poros implementation, broader hardware support, and additional model families, but the

central result already stands on its own. Poros shows that frozen-base adapter training does not require full-model GPU residency to preserve the resident training trajectory. In the memory-limited regime, that single systems change is enough to move 32B QLoRA into a 16 GB GPU-memory budget.

A Benchmark Protocol Variants

This appendix reports consumer-hardware campaigns measured under protocols that differ from the canonical 200-step $\times$ 3-pass benchmark protocol of the main text.

Fused-LoRA refresh (RTX 3090, fair-v2). Table 10 reports a follow-up RTX 3090 campaign that exercises the fused-LoRA execution path (Section 3.6) together with a re-measured Unsloth arm ("fair-v2") under the same fresh-process protocol used throughout (Section 4.3). Under this protocol Poros is faster (-7.8% at 7B and -9.2% at 32B), and the speed gap to Unsloth narrows from $1.65\times$ to $1.38\times$ at 7B and from $1.55\times$ to $1.37\times$ at 32B. The 32B memory advantage remains large (43.4% less VRAM than Unsloth). At 7B, the two methods are memory-competitive under this re-measured Unsloth arm (Poros 6.74 GB vs. Unsloth 6.23 GB) rather than Poros-favoured as in Table 6; the two campaigns were measured independently at different times, so we report both protocols rather than collapsing them, since the Table 6 comparison uses the canonical baseline measurement methodology consistent with the primary-platform numbers in Table 5.

Table 10. RTX 3090 fused-LoRA refresh: Poros with the fused-LoRA execution path (Section 3.6) versus a fresh-process Unsloth measurement ("fair-v2" protocol).

Scale	Method	VRAM (GB)	s/step	Loss
7B	Unsloth (fair-v2)	6.230	0.582	12.018
7B	**Poros + fused LoRA**	**6.738**	0.803	12.056
32B	Unsloth (fair-v2)	20.790	2.130	11.971
32B	**Poros + fused LoRA**	**11.760**	2.917	11.989

Protocol: matched seq_len=512, 200 steps, identical data and seeds; both arms refreshed relative to Table 6. At 7B, Poros and Unsloth use comparable VRAM under this protocol (6.74 vs. 6.23 GB). At 32B, Poros uses **43.4% less VRAM** than Unsloth (11.76 vs. 20.79 GB). Speed ratios (Poros / Unsloth) improve from Table 6: $1.65\times \rightarrow 1.38\times$ at 7B and $1.55\times \rightarrow 1.37\times$ at 32B. Source: the 2026-04-10 fused-LoRA release campaign.

Consumer-Blackwell validation (RTX 5090). As an additional consumer-hardware datapoint on a current-generation card, we ran Qwen2.5-32B NF4 Poros Blockwise on an RTX 5090 (Blackwell consumer, 32 GB, PCIe Gen5; PyTorch 2.11.0+cu128) under a single-pass validation protocol (one measured pass per configuration at sequence length 512 and block size 4, rather than the canonical 200-step $\times$ 3-pass benchmark protocol). Table 11 reports both runs: a 50-step synthetic pass measured a peak allocated VRAM of 11.02 GB (8.59 GB steady-state) at 12.9 s/step, and a 100-step pass on real data (OpenAssistant/oasst1, learning rate 2×10^{-4} with 10-step warmup, padding masked out of the loss with `-100` labels) measured the same 11.02 GB peak at 12.7 s/step, with real-token cross-entropy fluctuating trend-free in the 2.0–2.5 band (each row is seen once; single-epoch generalization gain is below per-batch composition noise). The run validates that the training loop executes end-to-end on real data on this card at the stated VRAM; it is not a downstream-quality measurement (held-out-eval learning curves appear in Section 5.2). The measured peak matches the A100 and B300 measurements for the same configuration, and the synthetic loss trajectory tracks the A100 run to within $\sim 2 \times 10^{-4}$ (exact parity is guaranteed

Table 11. Consumer-Blackwell validation (RTX 5090, 32 GB, PCIe Gen5; PyTorch 2.11.0+cu128): Qwen2.5-32B NF4 Poros Blockwise at seq_len=512, block size 4, under the single-pass validation protocol (one measured pass per configuration, not the canonical 200-step $\times$ 3-pass benchmark protocol).

Run	Data	Steps	Peak VRAM (GB)	s/step	Loss (per-token CE)
Synthetic	deterministic tokens	50	**11.02**	12.9	$12.61 \rightarrow 12.03$
Real data	OpenAssistant/oasst1	100	**11.02**	12.7	2.0–2.5 (masked; trend-free)

The real-data run masks padding out of the loss (`-100` labels at padded positions; rows with fewer than two real tokens dropped) and trains at learning rate 2×10^{-4} with a 10-step warmup (per-step learning rates are recorded in the released events). Real-token CE fluctuates in the 2.0–2.5 band with no visible trend: each row is seen exactly once, so the single-epoch generalization gain is smaller than per-batch composition noise. This row is a loop-executes-on-real-data and memory-envelope check, not a chat-quality result; held-out-eval learning curves appear in the extended audits (Section 5.2). Peak allocated VRAM matches the A100 and B300 measurements for the same configuration (steady-state 8.59 GB). The synthetic loss trajectory tracks the A100 run to within $\sim 2 \times 10^{-4}$; exact parity is guaranteed within a hardware/software stack, not across stacks.

within a hardware/software stack, not across stacks). These runs confirm that the 32B-under-16 GB training budget reproduces on consumer Blackwell hardware; raw per-step artifacts are released with the repository.

B Experimental Dense-BF16 Runtime

The NF4 results in the main text are the validated, supported configuration of Poros. As an experimental extension, the released system also includes a separate `bf16_bitwise` runtime that streams the frozen base in dense BF16 (2 bytes per parameter) rather than NF4, using the same blockwise-residency schedule: packed single-copy pinned-slab transfers (one host-to-device copy per block load), native BF16 LoRA arithmetic, per-block RNG save/restore for dropout, and strict CUDA determinism. The path is purely additive—selecting it does not alter the NF4 runtime.

Bitwise training parity, including at 32B scale. We apply the same standard as the NF4 gates: a fully resident run and the blockwise streaming run are launched from identical initial state in separate processes, and per-microstep losses, adapter gradients before the optimizer step, adapter weights after the final step, and Adam optimizer state are compared with `torch.equal`. On an A100 80 GB (PyTorch 2.7.1+cu126) the gate passes on all six cells for Qwen2.5-0.5B (gradient accumulation $G \in \{1, 2, 4\}$, sequential and block-major accumulation), and the gate also passes on a second A100 stack (PyTorch 2.5.1+cu121). At full scale, Qwen2.5-32B passes the same gate on the A100 in both accumulation modes ($G{=}2$; sequential with dropout 0.05, and block-major): all compared tensors—512 adapter gradients, 512 adapter weights, and 1,536 optimizer-state tensors per arm—are exactly equal, with measured maximum absolute difference 0.0. Block-major accumulation requires dropout disabled by construction (it reorders dropout RNG consumption relative to resident accumulation, so the configuration is rejected rather than silently non-bitwise); the sequential cell verifies dropout RNG parity at 32B. The same 32B runs measure the transfer saving of block-major accumulation directly: at $G{=}2$ it reduces host-to-device traffic from 881.6 GB to 374.5 GB over the gate workload ($\sim 2.4\times$).

Table 12. Experimental dense-BF16 runtime: single measured 50-step passes (seq_len 512, A100 80 GB). Streaming the frozen base in *full* BF16 precision (no quantization) removes the bulk of GPU residency, and the reduction grows with model size—reaching $\geq 82\%$ at 32B. The resident base is the BF16 weight floor (2 bytes per parameter); a resident training run additionally holds optimizer state and activations, so the listed reductions are lower bounds. Resident-vs-blockwise training parity is bitwise ($0.00\text{e}{+00}$ on loss, gradients, weights, and optimizer state) at 0.5B and 32B (this section).

Model	Resident base	Poros VRAM	Reduction	s/step
Qwen2.5-7B	~ 15 GB	6.22 GB	$\geq 59\%$	14.6
Gemma-3-27B	~ 54 GB	11.56 GB	$\geq 79\%$	53.6
Qwen2.5-32B	~ 65 GB	11.40 GB	$\geq 82\%$	63.8

Measured cost and footprint (single-pass protocol, A100). Dense BF16 streaming preserves the low-VRAM property (Table 12): even with the frozen base streamed in full BF16 precision—2 bytes per parameter, no quantization—Qwen2.5-32B fits the same ~ 11 GB VRAM class as the NF4 runtime, versus roughly 65 GB of weights alone for a resident BF16 32B base.

The costs move to the host side: the streamed base occupies host RAM at 2 bytes per parameter (~ 64 GB at 32B, versus ~ 18 GB for NF4) and each step moves roughly $3.6\times$ more bytes over PCIe than NF4. The step times are far larger than the byte ratio alone implies (the NF4 runtime measures 2.05 s/step for the same 32B model on the same A100, Table 4): unlike the NF4 runtime, the experimental BF16 runtime loads each block synchronously with no prefetch overlap, so transfers serialize with compute instead of hiding behind it. These are single measured passes under the validation protocol of Section A, not entries in the canonical benchmark tables, and the runtime remains experimental in the released version; raw gate and run artifacts are released with the repository.

References

[1] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. LoRA: Low-rank adaptation of large language models. In *ICLR*, 2022.

[2] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer. QLoRA: Efficient finetuning of quantized LLMs. In *NeurIPS*, 2023.

[3] D. Han and M. Han. Unsloth: Efficient LLM fine-tuning. `https://github.com/unslothai/unsloth`, 2024.

[4] M. Shing, M. Koyama, and T. Akiba. DiffusionBlocks: Block-wise neural network training via diffusion interpretation. In *ICLR*, 2026. `https://openreview.net/forum?id=pwVSmK71cS`.

[5] PyTorch Contributors. `torch.utils.checkpoint` — gradient checkpointing. PyTorch documentation, 2024. `https://pytorch.org/docs/stable/checkpoint.html`.

[6] T. Chen, B. Xu, C. Zhang, and C. Guestrin. Training deep nets with sublinear memory cost. *arXiv:1604.06174*, 2016.

[7] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He. ZeRO-Offload: Democratizing billion-scale model training. In *USENIX ATC*, 2021.

[8] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He. ZeRO-Infinity: Breaking the GPU memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021.

[9] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. Zhang. FlexGen: High-throughput generative inference of large language models with a single GPU. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, 2023.

[10] R. Y. Aminabadi, S. Rajbhandari, A. A. Awan, C. Li, D. Li, E. Zheng, O. Ruwase, S. Smith, M. Zhang, J. Rasley, and Y. He. DeepSpeed-Inference: Enabling efficient inference of transformer models at unprecedented scale. In *SC*, 2022.

[11] Y. Zhao, A. Gu, R. Varma, L. Luo, C.-C. Huang, M. Xu, L. Wright, H. Shojanazeri, M. Ott, S. Shleifer, A. Desmaison, C. Balioglu, P. Damania, B. Nguyen, G. Chauhan, Y. Hao, A. Mathews, and S. Li. PyTorch FSDP: Experiences on scaling fully sharded data parallel. *Proc. VLDB Endow.*, 16(12):3848–3860, 2023.

[12] Answer.AI. You can now train a 70b language model at home. https://www.answer.ai/posts/2024-03-06-fsdp-qlora.html, 2024.

[13] Q. Zhang, M. Chen, A. Bukharin, P. He, Y. Cheng, W. Chen, and T. Zhao. AdaLoRA: Adaptive budget allocation for parameter-efficient fine-tuning. In *ICLR*, 2023.

[14] S.-Y. Liu, C.-Y. Wang, H. Yin, P. Molchanov, Y.-C. F. Wang, K.-T. Cheng, and M.-H. Chen. DoRA: Weight-decomposed low-rank adaptation. In *ICML*, 2024.

[15] S. Mangrulkar, S. Gugger, L. Debut, Y. Belkada, S. Paul, and B. Bossan. PEFT: State-of-the-art parameter-efficient fine-tuning methods. https://github.com/huggingface/peft, 2023.

[16] R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. B. Hashimoto. Stanford Alpaca: An instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

[17] S. Gugger, L. Debut, T. Wolf, P. Schmid, Z. Mueller, S. Manber, M. Sun, and B. Bossan. Accelerate: Training and inference at scale made simple, efficient and adaptable. https://github.com/huggingface/accelerate, 2022.

[18] C. Liao, M. Sun, Z. Yang, J. Xie, K. Chen, B. Yuan, F. Wu, and Z. Wang. LoHan: Low-cost high-performance framework to fine-tune 100B model on a consumer GPU. *arXiv:2403.06504*, 2024.

[19] R. Yang and Z. Wen. An efficient heterogeneous co-design for fine-tuning on a single GPU. *arXiv:2603.16428*, 2026.

[20] S. Chen, Z. Wang, Z. Guan, Y. Liu, and P. B. Gibbons. Practical offloading for fine-tuning LLM on commodity GPU via learned sparse projectors. *arXiv:2406.10181*, 2024.

[21] C. Song and X. Tang. Memory-efficient backpropagation for fine-tuning LLMs on resource-constrained mobile devices. *arXiv:2510.03425*, 2025.

[22] Y. Liu, Z. Wang, M. Zhao, E. Nie, M. Wang, Q. Li, F. Ren, S. Feng, D. Wang, and H. Schütze. ChunkFT: Byte-streamed optimization for memory-efficient full fine-tuning. *arXiv:2605.21177*, 2026.

[23] D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, and J. Steinhardt. Measuring massive multitask language understanding. In *ICLR*, 2021.

[24] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano, C. Hesse, and J. Schulman. Training verifiers to solve math word problems. *arXiv:2110.14168*, 2021.

[25] A. Köpf, Y. Kilcher, D. von Rütte, S. Anagnostidis, Z.-R. Tam, K. Stevens, A. Barhoum, N. M. Duc, O. Stanley, R. Nagyfi, S. ES, S. Suri, D. Glushkov, A. Dantuluri, A. Maguire, C. Schuhmann, H. Nguyen, and A. Mattick. OpenAssistant conversations—democratizing large language model alignment. In *NeurIPS Datasets and Benchmarks*, 2023.

[26] L. Gao, J. Tow, B. Abbasi, et al. A framework for few-shot language model evaluation. EleutherAI `lm-evaluation-harness`, Zenodo, 2023.