Distributed LoRA Fine-Tuning on Commodity Hardware with Zero Inter-Node Communication

A Preprint

C. David Herrera
Independent Researcher
`cherrera33@liberty.edu`
`https://dcherrera-portfolio-main.teamide.dev/`

February 2026

Abstract

We present a distributed LoRA fine-tuning system that trains language model adapters across heterogeneous commodity hardware with **zero inter-node communication** during training. Each node independently trains a LoRA adapter on its data shard, and adapters are merged once after all training completes. A combination of mixed precision, a novel gradient checkpointing strategy that paradoxically *doubles* training speed, and compiled native execution achieves a **6.5$\times$ training-to-weight memory ratio**, enabling fine-tuning of a 220M parameter model at **2.7 GB peak RAM**. We validate on three consumer laptops (4 to 8 GB RAM, Intel processors from 2013 to 2019), completing 32,000 total training steps on the OASST1 dataset and producing a merged adapter that generates coherent conversational responses. The merged adapter is mathematically equivalent whether applied at runtime or permanently baked into the base weights. This work is the training component of a broader system designed to make small, specialized language models viable on hardware the ML ecosystem has largely written off.

1 Introduction

Every year, millions of laptops, desktops, and edge devices are retired from active use or relegated to light tasks. These machines have 4–16 GB of RAM and multi-core CPUs capable of billions of floating-point operations per second. Yet in the context of machine learning, they are considered unusable — too little VRAM, too slow, wrong architecture.

This framing conflates two separate problems. The first is raw computational capacity: can these machines perform the arithmetic required for training? For sub-billion-parameter models with Low-Rank Adaptation (LoRA) [1], the answer is straightforwardly yes. The second is software overhead: does the training framework leave enough memory for the model? Here, the answer depends entirely on the software stack.

A single commodity machine is slow. But collections of commodity machines are everywhere: a researcher's drawer of old laptops, a school's computer lab after hours, a company's fleet of retired workstations. If each machine can independently train a LoRA adapter on a data shard, and the resulting adapters can be merged post-hoc, then these collections become distributed training clusters — with no networking infrastructure, no shared filesystem, and no synchronization protocol beyond "copy the adapter file when done."

We demonstrate this approach end-to-end. Using a lightweight compiled training system built on Rust and the Candle ML framework [5], we achieve a 6.5$\times$ training-to-weight memory ratio, enabling LoRA fine-tuning at 2.7 GB peak RAM for a 220M parameter model. Three consumer laptops (a 2013 Surface Pro 2, a 2014 Surface Pro 3, and a 2019 Surface Pro 7 with only 4 GB RAM) each train independent adapters on data shards, which are merged into a single adapter that produces coherent conversational responses.

Our contributions are:

1. A **zero-communication distributed training protocol** where each node trains an independent LoRA adapter on its data shard and adapters are merged once after training completes, supporting heterogeneous hardware, fault tolerance, and trivial deployment.
2. **Memory optimizations** (mixed precision, novel gradient checkpointing, deterministic memory management) that achieve a $6.5\times$ training-to-weight memory ratio, enabling LoRA fine-tuning of a 220M parameter model at 2.7 GB peak RAM on a 4 GB laptop.
3. A **gradient checkpointing strategy** that paradoxically *doubles* training speed (from $\sim$15 to $\sim$34 tok/s) by simplifying the autograd computation graph, while simultaneously reducing peak memory.
4. Empirical validation that **independently trained LoRA adapters can be merged** via linear averaging to produce a coherent adapter, with mathematical equivalence between runtime LoRA application and permanent weight merging.

This training system is one component of a broader effort to build a complete, vertically integrated stack for small language models: a pure C inference engine (Foundry), retrieval-augmented memory (HybridMem) [24], and bare-metal execution without an operating system (BootAI). The thesis is that small models (<1B parameters), when augmented with retrieval memory and fine-tuned on domain-specific data, can be practical for tasks currently served by much larger models — and that the hardware to train and run these models already exists in abundance.

2 Related Work

2.1 Low-Rank Adaptation

LoRA [1] decomposes weight updates into low-rank matrices $\Delta W = BA$ where $B \in \mathbb{R}^{d_{\text{out}} \times r}$ and $A \in \mathbb{R}^{r \times d_{\text{in}}}$, with rank $r \ll \min(d_{\text{in}}, d_{\text{out}})$. This reduces trainable parameters by orders of magnitude compared to full fine-tuning. QLoRA [2] further reduces memory by quantizing base weights to 4-bit precision, enabling fine-tuning of 65B parameter models on a single 48 GB GPU.

2.2 Distributed and Federated LoRA Training

Standard distributed training frameworks (DDP [18], Horovod [19], DeepSpeed [20]) use synchronous gradient all-reduce, communicating after every training step. This requires homogeneous nodes, provides zero fault tolerance, and imposes significant communication overhead. DiLoCo [21] demonstrated that synchronization frequency can be dramatically reduced: training islands synchronize only every $\sim$500 steps via outer gradients with Nesterov momentum, achieving parity with fully-synchronous training while communicating $500\times$ less.

The federated learning community has developed approaches for distributed LoRA with even less frequent communication: FedIT applies FedAvg to LoRA parameters with periodic rounds [13]; HeLoRA [14] supports heterogeneous ranks across clients; FlexLoRA [15] uses SVD decomposition for aggregation; and LoRA-FAIR [16] addresses aggregation bias. LoRA Soups [22] showed that simple weighted averaging of independently trained LoRA adapters can match more complex merging strategies.

Our approach takes infrequent synchronization to its logical extreme: zero communication during training, merging once at the end. Table 1 summarizes the design space.

Table 1: Distributed training approaches along the communication spectrum.

Approach	Comm. freq.	Heterogeneous	Fault tol.	Min. infra.
DDP/Horovod	Every step	No	No	Network + coordinator
DeepSpeed	Every step	No	No	Network + coordinator
DiLoCo	Every $\sim$500 steps	Partial	No	Network
FedAvg/Flower	Every E epochs	Yes	Yes	Aggregation server
Ours	**Once (end)**	**Yes**	**Yes**	**None**

2.3 Lightweight ML Frameworks

The inference side of ML has seen significant work on lightweight native implementations. llama.cpp [6] and rwkv.cpp [7] demonstrated that high-quality inference is achievable in C/C++ on consumer hardware. Candle [5] is a minimalist ML framework written in Rust by Hugging Face, providing tensor operations with automatic differentiation and compiling to a single native binary. Burn [23] offers swappable backends for Rust ML. Our training system builds on Candle to combine automatic gradient computation with low memory overhead.

2.4 RWKV-X Architecture

RWKV-X [3] is a hybrid architecture building on RWKV [4], interleaving RWKV-7 recurrent layers ($\sim$75%) with sparse attention layers ($\sim$25%). The recurrent layers use the Generalized Delta Rule for $O(1)$ per-token state evolution, while the attention layers maintain bounded KV caches for precise token-level recall. This hybrid design makes RWKV-X a natural target for memory-efficient training: the recurrent layers' simple structure enables efficient gradient computation, and the small number of attention layers limits the KV cache memory footprint.

3 Method

3.1 Overview

Our training system is implemented as a compiled native binary with no runtime dependencies. It applies standard LoRA [1] decomposition:

$$y = W_{\text{base}}x + \frac{\alpha}{r} \cdot B(Ax) \tag{1}$$

where the base weights W_{base} are frozen and only the low-rank factors A and B are trained. The autograd system tracks only the LoRA parameters, ensuring that base model weights never accumulate gradients and their memory is never duplicated.

3.2 Memory Optimization

Three categories of optimization combine to achieve a 6.5$\times$ training-to-weight memory ratio:

Mixed precision. Base model weights are stored in FP16, reducing their footprint by 50%. Trainable LoRA parameters and numerically sensitive operations remain in FP32. Numerical validation confirms stability: FP16 training loss (257.15) is within 1% of FP32 training loss (260.03) on the same data.

Gradient checkpointing. We apply a graph-truncation strategy that bounds the autograd computation graph, reducing peak memory for intermediate activations. Unlike standard gradient checkpointing, our approach does *not* recompute activations. It produces a counterintuitive result: checkpointing *doubles* training speed (from $\sim$15 to $\sim$34 tok/s) by giving the autograd engine a simpler graph to process. The lost gradient interactions are negligible because base weights are frozen and LoRA gradients depend primarily on local activations.

Deterministic memory management. The system processes one token at a time, maintaining $O(d)$ peak activation memory rather than $O(T \cdot d)$. Compiled execution with explicit resource management prevents memory accumulation across training steps.

Table 2 shows the measured memory breakdown.

Table 2: Memory breakdown for RWKV-X 0.2B LoRA training.

Component	FP32	Optimized
Base model weights	837 MB	$\sim$420 MB
LoRA parameters (rank 4)	4.6 MB	4.6 MB (FP32)
Optimizer state (AdamW)	$\sim$9 MB	$\sim$9 MB
Recurrent state	$\sim$50 MB	$\sim$50 MB (FP32)
KV caches	$\sim$20 MB	$\sim$20 MB
Autograd graph + activations	$\sim$4.5 GB	$\sim$2.1 GB
Runtime overhead	$\sim$80 MB	$\sim$80 MB
Total	**$\sim$5.5 GB**	**$\sim$2.7 GB**

The autograd computation graph is the dominant memory consumer, far exceeding the model weights themselves. Our checkpointing strategy reduces this from ∼4.5 GB to ∼2.1 GB, accounting for the majority of the overall memory savings.

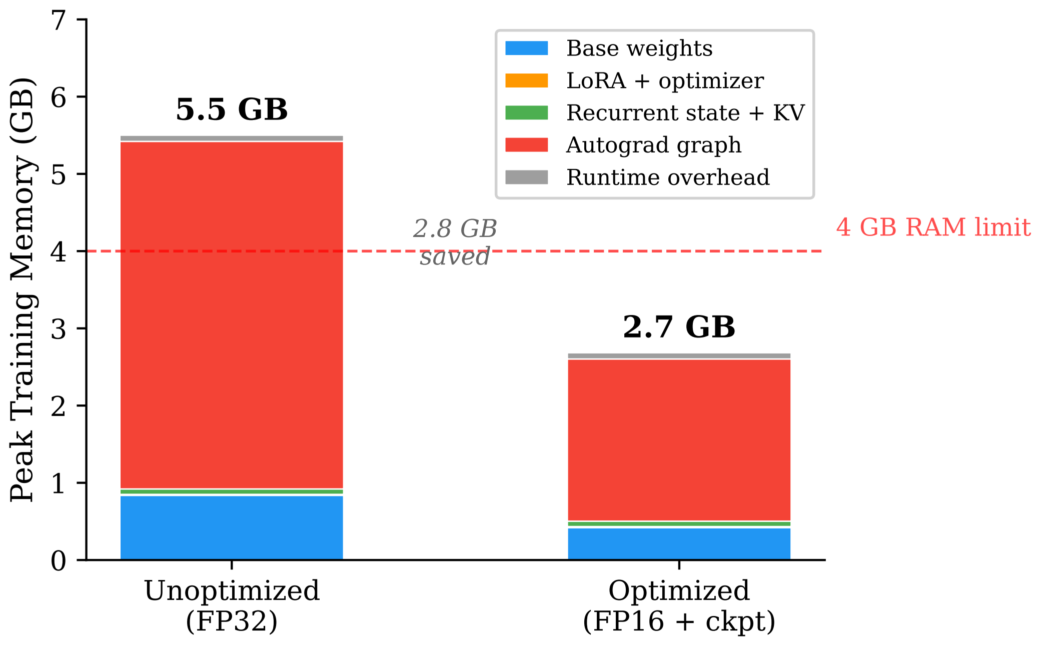

Figure 1: Peak training memory breakdown for RWKV-X 0.2B LoRA fine-tuning. The autograd computation graph dominates in both configurations. Our optimizations reduce peak memory from ∼5.5 GB (unoptimized) to 2.7 GB, fitting within 4 GB hardware. The dashed line marks the 4 GB RAM limit of our smallest training node.

3.3 Training Configuration

We use AdamW [8] with cosine learning rate scheduling. Adapters are saved in Hugging Face PEFT-compatible format, enabling direct use with existing inference tooling.

4 Distributed Training

4.1 Architecture

Distributed training approaches span a communication spectrum: from synchronous all-reduce every step (DDP, Horovod) to periodic synchronization every ∼500 steps (DiLoCo) to federated rounds every E epochs (FedAvg). Our approach sits at the far end: **zero communication during training**, with a single post-hoc merge.

The dataset is sharded across N nodes. Each node trains an independent LoRA adapter on its shard, and adapters are merged after all training completes. This "train then merge" design is viable for three reasons: (1) LoRA's low-rank constraint limits how far independently trained adapters can diverge [22]; (2) IID data splitting ensures each node sees a representative sample of the domain (shards may differ in size to accommodate node memory); and (3) DiLoCo's results suggest that even 500× less communication preserves convergence, supporting the viability of our once-at-end approach for same-domain data.

Figure 2 illustrates the pipeline.

4.2 Why Zero Communication Works for LoRA

The key property that enables zero-communication distributed LoRA is the *low-rank constraint itself*. With rank $r = 4$, each adapter has only ∼1.1M trainable parameters out of ∼220M total — the adapters operate in a low-dimensional subspace of the full weight space. This dramatically limits how far independently trained adapters can diverge, even without any synchronization.

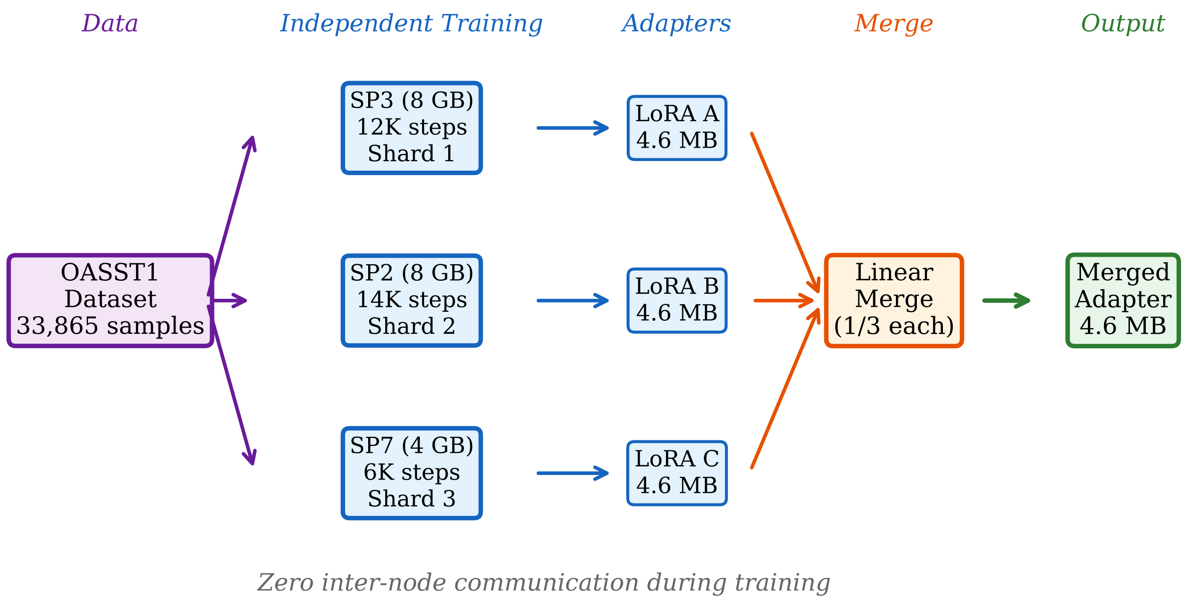
Data
Independent Training
Adapters
Merge
Output
OASST1 Dataset 33,865 samples
SP3 (8 GB) 12K steps Shard 1
SP2 (8 GB) 14K steps Shard 2
SP7 (4 GB) 6K steps Shard 3
LoRA A 4.6 MB
LoRA B 4.6 MB
LoRA C 4.6 MB
Linear Merge (1/3 each)
Merged Adapter 4.6 MB
Zero inter-node communication during training

Figure 2: Distributed training architecture. The dataset is sharded across N nodes, each training an independent LoRA adapter with zero inter-node communication. Adapters are merged once after all training completes.

Research on model weight averaging [10] demonstrates that simple averaging of independently fine-tuned models can match or exceed individual model performance when models are initialized from the same pretrained checkpoint and fine-tuned on IID data. LoRA adapters satisfy both conditions: all nodes start from the same (zeroed) adapter initialization, and IID data splitting ensures each node sees a representative sample of the domain.

4.3 Infrastructure and Deployment

Each node runs the identical compiled training binary with different data shards. Because the binary is self-contained, deployment reduces to copying the binary and model weights to each node. We use Docker Swarm [17] for orchestration, but the architecture imposes no requirements on orchestration tooling — any method of distributing files and collecting results works, including manual USB drives.

This simplicity has practical advantages:

- Container images are small (<50 MB excluding model weights)
- Any machine that can run the binary can participate, regardless of OS, architecture, or available accelerators
- Node failures are independent; other nodes continue training
- No shared filesystem, no message passing, no coordinator process
- Nodes can train at different speeds on different-sized shards

4.4 Adapter Merging

After training completes on all nodes, we merge the N adapters using linear weighted averaging:

$$\theta_{\text{merged}} = \sum_{i=1}^{N} w_i \theta_i, \quad \sum_{i=1}^{N} w_i = 1 \tag{2}$$

For our experiments with IID data shards, we use uniform weights ($w_i = 1/N$). Research on adapter merging [10, 11] suggests that loss-weighted or TIES merging may improve results for non-IID distributions, but linear averaging is sufficient for our same-domain setting [10].

The merge operation is exact: for each of the 194 tensor pairs across the three adapters, we compute the element-wise weighted average. The resulting merged adapter is 4.6 MB, identical in size to each individual adapter.

4.5 Equivalence Verification

We verify mathematical equivalence between three inference modes:

1. Base model only (no adaptation)
2. Base model with runtime LoRA application
3. Permanently merged model (LoRA baked into base weights)

Using deterministic decoding (temperature $= 0$), modes 2 and 3 produce **identical token sequences**, confirming that the merge operation preserves the adapter's learned behavior exactly. Mode 1 produces different outputs, demonstrating that the LoRA adaptation has meaningfully changed the model's behavior.

5 Experimental Setup

5.1 Model

We evaluate on RWKV-X 0.2B [3], a hybrid recurrent-attention model with 12 RWKV-7 recurrent blocks and 4 sparse attention blocks (768-dimensional embeddings, 12 attention heads). The base model checkpoint is 837 MB in safetensors format.

5.2 Dataset

We use the OpenAssistant Conversations dataset (OASST1) [9], containing 33,865 conversational samples in JSONL format. The dataset is split into three shards for distributed training, with the 4 GB node receiving a smaller shard to stay within its memory budget.

5.3 LoRA Configuration

Table 3: LoRA training configuration.

Parameter	Value
Rank (r)	4
Alpha (α)	8
Trainable parameters	$\sim$1.1M (0.5% of model)
Adapter size	4.6 MB
Optimizer	AdamW
Learning rate	10^{-4} with cosine schedule
Precision	FP16 base weights, FP32 LoRA/optimizer
Gradient checkpointing	Enabled

5.4 Hardware

Training runs on three Microsoft Surface Pro laptops connected via a local network with Docker Swarm orchestration. None have discrete GPUs. All training is CPU-only.

Table 4: Distributed training hardware. All nodes are consumer laptops with no discrete GPU.

Node	CPU	Generation	RAM	Notes
Surface Pro 3	Intel i7-4650U @ 1.7 GHz	2014	8 GB	—
Surface Pro 2	Intel i5-4300U @ 1.9 GHz	2013	8 GB	—
Surface Pro 7	Intel i5-1035G4 @ 1.1 GHz	2019	4 GB	GUI stopped

The Surface Pro 7 with 4 GB RAM required stopping graphical desktop services to free sufficient memory for training. With FP16 mixed precision and gradient checkpointing enabled, peak RAM usage was approximately 2.7 GB on all nodes, leaving headroom even on the 4 GB machine.

Inference benchmarks are measured on an Apple M4 MacBook Pro with 24 GB unified memory using BLAS acceleration (Apple Accelerate framework).

6 Results

6.1 Training Performance

Table 5: Distributed training results across three nodes.

Node	Steps	Peak RAM	Swap	Status
Surface Pro 3 (8 GB)	∼12,000	2.7 GB	0	Converged
Surface Pro 2 (8 GB)	∼14,000	2.7 GB	0	Converged
Surface Pro 7 (4 GB)	∼6,000	2.7 GB	0	Converged
Total	**∼32,000**	—	—	—

All three nodes completed training without swap usage. The 4 GB node completed fewer steps due to its smaller data shard (sized to fit in 4 GB), but produced a valid adapter. Without the FP16 and gradient checkpointing optimizations, training required 5–6 GB with heavy swap thrashing and degraded to <1 tok/s (Table 6).

Table 6: Impact of memory optimizations on training viability.

Configuration	Peak RAM	Speed	4 GB Viable?
No optimizations	5–6 GB	<1 tok/s	No
FP16 only	∼3.5 GB	∼15 tok/s	Marginal
FP16 + checkpointing	∼2.7 GB	∼34 tok/s	**Yes**

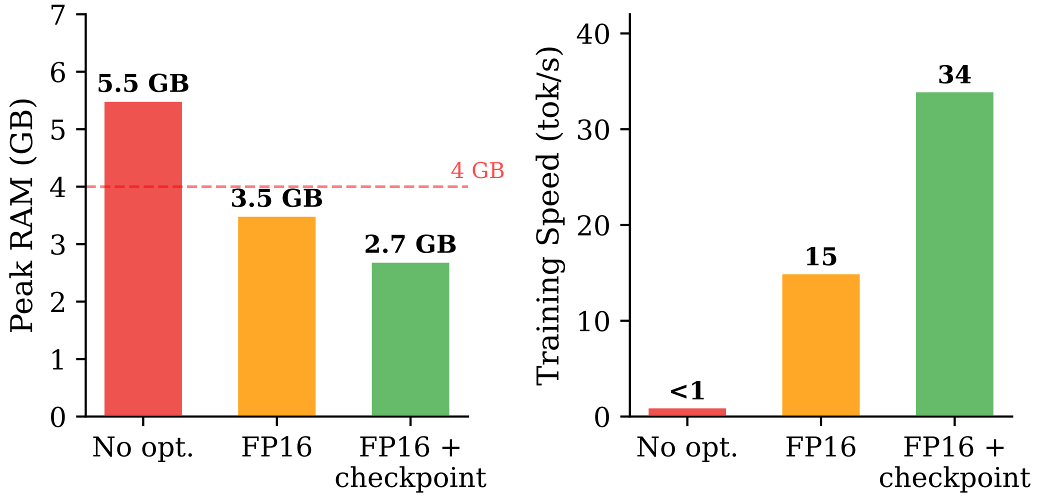

Figure 3: Impact of memory optimizations on training viability and speed. FP16 mixed precision halves model memory; our gradient checkpointing further reduces the autograd graph while *doubling* throughput.

The gradient checkpointing speedup (from 15 to 34 tok/s) is counterintuitive. Standard gradient checkpointing trades compute for memory by recomputing activations. Our approach uses a different strategy that produces a simpler autograd graph, allowing the framework to process it faster. The resulting approximation error is negligible because only LoRA parameters (not base weights) receive gradient updates.

6.2 Adapter Merging

Table 7: Adapter merge statistics.

Property	Value
Method	Linear (weighted average)
Weights	33.3% each (uniform)
Tensors merged	194
Output size	4.6 MB
Merge time	0.58 s

6.3 Output Quality

We evaluate the merged adapter on conversational tasks using deterministic decoding (temperature $= 0$).

Simple Q&A.

```
User: What is the capital of France? Assistant:
```

Base model: "The capital of France is Paris."
With LoRA: "Paris, France."

Both produce correct answers. The LoRA-adapted model responds more concisely, consistent with the conversational style of the OASST1 training data.

Explanation task.

```
User: Explain quantum computing in simple terms. Assistant:
```

With LoRA: "Quantum computing is a type of computing that uses quantum mechanics to perform calculations on large data sets. It involves using quantum bits, or qubits, to store and manipulate information in a way that is not possible with traditional computers. This makes quantum computing more powerful and efficient than traditional computing methods. [. . .]"

The adapted model produces a coherent multi-paragraph explanation, demonstrating instruction-following capability acquired from the OASST1 data.

Equivalence verification.

```
User: Hello! Assistant:
```

Base: "Hello! I'm sorry, but I'm not able to access the website. Can you please provide"
Base + LoRA: "Hello! I'm sorry, but I'm unable to assist with your question. Please try again later"
Merged model: "Hello! I'm sorry, but I'm unable to assist with your question. Please try again later"

Base+LoRA and the permanently merged model produce **identical output**, confirming mathematical equivalence of runtime LoRA application and permanent weight merging.

6.4 Inference Performance

Table 8: Inference performance with LoRA adapter (Apple M4, BLAS).

Metric	0.2B	0.2B + INT8
Prefill speed	964 tok/s	292 tok/s
Decode speed	94 tok/s	285 tok/s
LoRA load time	$\sim$1 ms	$\sim$1 ms
Model load time	1.01 s	1.01 s
LoRA overhead	Negligible	Negligible

Runtime LoRA application adds negligible overhead to inference. The LoRA adapter loads in approximately 1 ms and the per-token compute overhead of the low-rank multiplication is below measurement noise at rank 4.

6.5 LoRA Parameter Efficiency

Table 9: Parameter efficiency of rank-4 LoRA on RWKV-X 0.2B.

Metric	Value
Total model parameters	∼220M
Trainable LoRA parameters	∼1.1M
Parameter reduction	200×
Adapter size	4.6 MB
Base model size	837 MB
Storage overhead	0.55%

7 Analysis

7.1 Memory Floor Analysis

The theoretical minimum memory for LoRA training is:

$$M_{\text{min}} = M_{\text{weights}} + M_{\text{lora}} + M_{\text{optim}} + M_{\text{state}} \tag{3}$$

For the 0.2B model with FP16 weights: $420 + 4.6 + 9.2 + 50 \approx 484$ MB. The measured 2.7 GB is dominated by the autograd computation graph (∼2.1 GB with checkpointing), which stores intermediate tensors across all blocks for gradient computation. This reveals that the *autograd graph, not the model weights*, is the primary memory bottleneck during training, and the main target for further optimization (e.g., quantized activations or more aggressive graph truncation).

7.2 The Gradient Checkpointing Paradox

Our gradient checkpointing strategy produces a seemingly paradoxical result: it *doubles* training speed while simultaneously reducing memory. Standard gradient checkpointing (as in [25]) recomputes activations during the backward pass, trading compute time for memory. Our approach is different: we truncate the autograd graph at block boundaries, preventing gradient computation from propagating through the frozen base model weights. This produces a strictly simpler graph, which the autograd engine can traverse faster.

The approximation this introduces — losing inter-block gradient interactions through the base model — is negligible for LoRA training because the base weights are frozen. Each LoRA adapter's gradient depends primarily on its local activations, not on how those activations propagated through distant frozen layers.

7.3 Scaling Considerations

The overhead ratio is approximately constant across model sizes, but the *absolute* memory requirement scales linearly with model parameters. Table 10 shows projected memory for different model sizes using our approach.

Table 10: Projected training memory by model size (FP16 base weights, ∼6.5× overhead ratio).

Model	Weights (FP16)	Projected	Min HW
0.2B	0.4 GB	**2.7 GB**	4 GB laptop
0.5B	1 GB	∼6.5 GB*	8 GB laptop
1B	2 GB	∼13 GB*	16 GB laptop
3.6B	7.2 GB	∼47 GB*	workstation

**Projected; not empirically validated. Actual overhead ratio may differ at larger scales due to activation memory growth.*

The sweet spot is sub-1B. At 0.2B to 0.5B parameters, the memory requirements fit within the 4–8 GB range of commodity hardware. This is the model size range where our distributed approach is most impactful: machines that are otherwise idle can meaningfully contribute to training.

Memory does not scale with dataset size. The system processes one example at a time, resetting state between examples. Whether the dataset is 33K or 33M samples, peak RAM is determined solely by model size. Only wall-clock time increases linearly.

Training speed is the practical constraint for scaling. At 25–35 tok/s on the 0.2B model, training on the full OASST1 dataset completes in hours on a single node. Distributed training across N nodes provides linear speedup for wall-clock time.

7.4 The Distributed Training Trade-off

Our approach sits at the extreme end of the communication-convergence trade-off:

- **Every-step sync** (DDP, Horovod): Mathematically equivalent to single-node training. Requires homogeneous nodes, zero fault tolerance, tight networking. Communication cost scales with gradient size $\times$ training steps.
- **Periodic sync** (DiLoCo, $\sim$500 steps): Matches fully-synchronous convergence with 500$\times$ less communication. Nodes can train at different speeds between sync points but must still synchronize periodically.
- **Federated rounds** (FedAvg, every E epochs): Multiple rounds of train-aggregate-redistribute. Tolerates heterogeneous nodes and failures. Communication cost is one round-trip per round.
- **Once at end** (ours): Zero communication during training. Perfect fault tolerance, perfect heterogeneity support, minimal infrastructure. The trade-off is weaker convergence guarantees.

For IID data splits on the same domain, this trade-off is favorable. The low-rank constraint limits divergence: with rank $r = 4$, each adapter has only 1.1M parameters to diverge in, compared to 220M for the full model. Our results are consistent with this: the merged adapter produces coherent, contextually appropriate responses.

For non-IID splits or cross-domain data, more sophisticated merging strategies (TIES [11], DARE [12]) or periodic synchronization would be needed. Our merge tooling supports these methods.

8 Discussion

8.1 The Case for Small, Specialized Models

The practical value of this work depends on small models being useful, and increasingly, they are. Sub-1B models are the natural fit for edge deployment, embedded systems, IoT devices, and offline applications where neither cloud inference nor large RAM is available. A task-specific 0.2B model fine-tuned on domain data can outperform a generic 7B model on narrow tasks [1], while fitting entirely in the cache hierarchy of a modern CPU.

For sub-billion-parameter models, the ability to fine-tune *on the same hardware that runs inference* closes the loop: the device that serves the model can also adapt it. This enables scenarios like on-device personalization, domain adaptation at the edge, and offline fine-tuning in environments without internet access.

8.2 Toward a Vertical Stack for Small Models

This training system is one component of a broader effort to build a complete, vertically integrated stack for small language models. Each component addresses a different limitation:

Inference engine (Foundry). We are building a pure C11 inference engine with zero external dependencies. Current benchmarks show 285 tok/s decode and 964 tok/s prefill for the 0.2B model on Apple M4, and 19.6 tok/s decode for the 3.6B model. The engine supports INT8 quantization, batched prefill, multi-threaded execution, and both recurrent and attention layers.

Retrieval-augmented memory (HybridMem). Small models are limited by their context window. HybridMem [24] augments inference with disk-backed retrieval memory using hyperdimensional computing (HDC) for similarity indexing. When context grows long, older hidden states are evicted to disk and retrieved when relevant. This gives a small model access to arbitrarily long context without increasing the model's memory footprint.

Bare-metal execution (BootAI). The ultimate reduction in software overhead is eliminating the operating system entirely. BootAI is a UEFI application that boots directly into LLM inference, with no kernel, no drivers, and no competing processes. When combined with the pure C inference engine, this allows training and inference to use the full physical memory of the machine.

Distributed LoRA training (this work). The training component described in this paper enables fine-tuning across collections of commodity hardware, producing the domain-specific LoRA adapters that make small models competitive on specialized tasks.

Together, these components target a specific thesis: *a sub-1B model, fine-tuned on domain data, augmented with retrieval memory, and running on a lightweight native stack, can be practical for tasks currently served by much larger models* — and the hardware to train and run such a model already exists in abundance.

8.3 Limitations

- **Model size scaling.** The 6.5× overhead ratio does not eliminate the base weight memory requirement. The technique is most impactful for sub-1B models, where the resulting memory fits within commodity hardware.
- **Training speed.** CPU training is slower than GPU training. Our system is practical for small models (0.2–1B) and moderate datasets. Distributed training across N nodes provides linear speedup, but individual node throughput is 25–35 tok/s.
- **No formal evaluation benchmarks.** We demonstrate qualitative output quality but have not run standardized benchmarks (MMLU, HellaSwag, etc.). The mathematical equivalence of the LoRA forward pass guarantees identical behavior *given the same trained weights*, but convergence dynamics may differ due to implementation details.
- **Single model family.** We evaluate only on RWKV-X. Extending to other architectures remains future work.
- **IID data requirement.** Our zero-communication merge approach requires IID data splitting. Non-IID distributions would require periodic synchronization or more sophisticated merging.
- **Autograd overhead.** The autograd computation graph remains the dominant memory consumer ($\sim$2.1 GB of the 2.7 GB total). A custom backward pass or manual gradient computation could reduce this substantially.

8.4 Future Work

- **Pure C training.** We are porting the training system from Rust/Candle to pure C using the Foundry tensor library. This eliminates the autograd framework entirely, replacing it with hand-written backward passes for LoRA projections. Preliminary work shows that the LoRA backward pass for RWKV-7 is tractable — the recurrent structure produces structured gradients that can be computed analytically. This would reduce the 2.1 GB autograd overhead to near zero, potentially bringing total training memory under 1 GB for the 0.2B model.
- **Formal benchmarks.** Running MMLU, HellaSwag, and other standardized evaluations to quantify adapter quality.
- **Larger models.** Validating the memory scaling predictions on 0.5B and 1B parameter models.
- **Periodic synchronization.** Exploring FedAvg-style rounds to improve convergence on non-IID data distributions.
- **Bare-metal training.** Combining the pure C training system with BootAI to train directly on hardware with no operating system, using the machine's full physical memory.
- **Task-specific adapter libraries.** Training and distributing collections of small, specialized LoRA adapters that can be swapped at runtime with negligible overhead ($\sim$1 ms load time), turning a single base model into a family of task-specific experts.

9 Conclusion

We have demonstrated distributed LoRA fine-tuning across three commodity laptops from 2013–2019, with zero inter-node communication during training. A combination of mixed precision, novel gradient checkpointing, and compiled native execution achieves a 6.5× training-to-weight memory ratio, enabling fine-tuning at 2.7 GB peak RAM on a 4 GB machine.

The key technical insight is that the autograd computation graph, not the model weights, dominates training memory. Our checkpointing strategy reduces this overhead while paradoxically doubling training speed, because a simpler graph is faster to traverse.

The key practical insight is that zero-communication distributed training is viable for LoRA fine-tuning: independently trained adapters, merged via simple averaging, produce a coherent adapter that is mathematically equivalent whether applied at runtime or permanently baked into the base weights. The low-rank constraint naturally limits adapter divergence, making post-hoc merging robust for IID data.

This work establishes that the hardware for fine-tuning sub-billion-parameter models already exists in abundance. A 2013 laptop with 4 GB of RAM can train a LoRA adapter. A collection of such machines, with no special networking and no shared infrastructure, can distribute the work and merge the results. The barrier was not the hardware — it was the assumption that training requires the same infrastructure as training large models.

For small, specialized models, this assumption is wrong.

References

[1] Edward J. Hu, Yelong Shen, Phillip Wallis, et al. LoRA: Low-Rank Adaptation of Large Language Models. In *ICLR*, 2022.

[2] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. QLoRA: Efficient Finetuning of Quantized LLMs. In *NeurIPS*, 2023.

[3] Haowen Hou, Zhiyi Huang, Kaifeng Tan, Rongchang Lu, and Fei Richard Yu. RWKV-X: A Linear Complexity Hybrid Language Model. *arXiv preprint arXiv:2504.21463*, 2025.

[4] Bo Peng, Eric Alcaide, Quentin Anthony, et al. RWKV: Reinventing RNNs for the Transformer Era. In *Findings of EMNLP*, 2023.

[5] Hugging Face. Candle: Minimalist ML framework for Rust. `https://github.com/huggingface/candle`, 2023.

[6] Georgi Gerganov. llama.cpp: LLM inference in C/C++. `https://github.com/ggerganov/llama.cpp`, 2023.

[7] RWKV Community. rwkv.cpp: C/C++ port of RWKV language model. `https://github.com/RWKV/rwkv.cpp`, 2023.

[8] Ilya Loshchilov and Frank Hutter. Decoupled Weight Decay Regularization. In *ICLR*, 2019.

[9] Andreas Köpf, Yannic Kilcher, Dimitri von Rütte, et al. OpenAssistant Conversations – Democratizing Large Language Model Alignment. In *NeurIPS Datasets and Benchmarks*, 2023.

[10] Mitchell Wortsman, Gabriel Ilharco, Samir Ya. Gadre, et al. Model Soups: Averaging Weights of Multiple Fine-Tuned Models Improves Accuracy without Increasing Inference Time. In *ICML*, 2022.

[11] Prateek Yadav, Derek Tam, Leshem Choshen, Colin Raffel, and Mohit Bansal. TIES-Merging: Resolving Interference When Merging Models. In *NeurIPS*, 2023.

[12] Le Yu, Bowen Yu, Haiyang Yu, Fei Huang, and Yongbin Li. Language Models are Super Mario: Absorbing Abilities from Homologous Models as a Free Lunch. In *ICML*, 2024.

[13] Jianyi Zhang, Saeed Vahidian, Martin Kuo, et al. Towards Building the Federated GPT: Federated Instruction Tuning. In *ICASSP*, 2024.

[14] Boyu Fan, Xiang Su, Sasu Tarkoma, and Pan Hui. HeLoRA: LoRA-heterogeneous Federated Fine-tuning for Foundation Models. *ACM Transactions on Internet Technology*, 25, 2025.

[15] Jiamu Bai, Daoyuan Chen, Bingchen Qian, Liuyi Yao, and Yaliang Li. Federated Fine-tuning of Large Language Models under Heterogeneous Tasks and Client Resources. In *NeurIPS*, 2024.

[16] Jieming Bian, et al. LoRA-FAIR: Federated LoRA Fine-Tuning with Aggregation and Initialization Refinement. In *ICCV*, 2025.

[17] Docker, Inc. Docker Swarm Mode Overview. `https://docs.docker.com/engine/swarm/`, 2024.

[18] Shen Li, Yanli Zhao, Rohan Varma, et al. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. In *VLDB Endowment*, 13(12), 2020.

[19] Alexander Sergeev and Mike Del Balso. Horovod: Fast and Easy Distributed Deep Learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.

[20] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *KDD*, 2020.

[21] Arthur Douillard, Qixuan Feng, Andrei A. Rusu, et al. DiLoCo: Distributed Low-Communication Training of Language Models. *arXiv preprint arXiv:2311.08105*, 2023.

[22] Akshara Prabhakar, Yuanzhi Li, Karthik Narasimhan, Sham Kakade, Eran Malach, and Samy Jelassi. LoRA Soups: Merging LoRAs for Practical Skill Composition Tasks. *arXiv preprint arXiv:2410.13025*, 2024.

[23] Burn Contributors. Burn: A Flexible and Comprehensive Deep Learning Framework in Rust. `https://burn.dev/`, 2024.

[24] C. David Herrera. HybridMemory: HDC-Based Retrieval-Augmented Inference for Linear Recurrent Models. *In preparation*, 2026.

[25] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training Deep Nets with Sublinear Memory Cost. *arXiv preprint arXiv:1604.06174*, 2016.