ChunkFT: Byte-Streamed Optimization for Memory-Efficient Full Fine-Tuning

Yongkang Liu[1] **Zijing Wang**[1] **Mengjie Zhao**[1] **Ercong Nie**[2] **Mingyang Wang**[3,4]
Qian Li[5*] **Feiliang Ren**[1] **Shi Feng**[1] **Daling Wang**[1] **and Hinrich Schütze**[3,4]
[1]Northeastern University, China; [2]Shanghai Jiao Tong University, China
[3]CIS, LMU Munich, Germany;[4]MCML, Germany; [5]Shandong University, China;

Abstract

This work presents CHUNKFT, a memory-efficient fine-tuning framework that reformulates full-parameter fine-tuning around a dynamically activated working set. CHUNKFT enables gradient computation for arbitrary sub-tensors without modifying the network architecture, providing an algorithmic foundation for optimizing arbitrary sub-networks while avoiding standard dense gradient computation. We provide a theoretical convergence analysis of CHUNKFT in the deterministic setting. Empirically, we apply CHUNKFT to fine-tune Llama 3-8B and Llama 3-70B using a single RTX 4090-24GB GPU and 2× H800-80GB GPUs, respectively. Full-parameter fine-tuning of a 7B model with a 1K input length requires only 13.72GB of GPU memory. The results demonstrate the effectiveness of CHUNKFT in memory usage, running time, and optimization quality. Moreover, downstream evaluations on language understanding, mathematical reasoning, and MT-Bench show that CHUNKFT consistently outperforms existing memory-efficient baselines. Notably, CHUNKFT achieves performance comparable to, and in some cases exceeding, full-parameter fine-tuning. Our repository is on `https://github.com/misonsky/chunk`.

1 Introduction

Full-parameter fine-tuning (FFT) of language models (LMs) has proven to be a successful paradigm in various downstream tasks [1, 2, 3]. However, a major obstacle in fine-tuning is the substantial memory required, which escalates as models increase in size and complexity, thereby limiting the scalability and accessibility for those with limited computational resources [4, 5, 6, 7].

To mitigate the memory constraints, Parameter-Efficient Fine-Tuning (PEFT) has been introduced to address memory limitations by updating only a small subset of parameters, while still achieving results comparable to full-model fine-tuning [8, 9, 10, 11, 12, 13]. Although PEFT methods have yielded promising results, existing studies indicate that they exhibit performance gaps compared to FFT in areas such as complex reasoning tasks and robustness [14].

In response to these challenges, MEFT (Memory-Efficient Fine-Tuning) presents a promising solution. MEFT typically reduces memory usage during FFT by improving optimizer design [4, 15, 16, 17] or optimizing the backpropagation (BP) process [18, 19]. Optimizer-based methods typically optimize only gradients and optimizer states, without addressing activation storage; consequently, their overall memory savings remain limited [19, 4, 15]. BP process optimization methods generally adopt layer-wise scheduling, represented by HiFT [19] and LISA [20]. These methods update only a subset of parameters at each step and achieve full-parameter updates through an iterative process, thereby reducing GPU memory usage for activations, gradients, and optimizer states simultaneously.

*Corresponding authors.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

Table 1: Algorithm feature summary. Here, M is the number of model parameters, r is the LoRA rank, m is the weight matrix dimension, L is the number of layers in the LLM, and K is the number of rotating parameter chunks in CHUNKFT or the number of layer-wise partitions in LOMO, BAdam, and HiFT. *Dense gradient* denotes whether dense gradients are materialized for the trainable scope. *Optimizer-state continuity* denotes whether historical optimizer states are reused across updates. The last column counts parameter-gradient computation only, normalized by dense full-model parameter-gradient computation, with one update per selected partition ($T = 1$). It excludes activation-gradient propagation for every method; see Section 3.2 and Appendix E for the cost definition and detailed comparison.

Method	Memory	Full param. training	Momentum & 2nd moment	Update precision	Grad. accum.	Sparse gradient	Optimizer-state continuity	Partition upper bound	Param.-gradient cost per coverage cycle
Adam	$18M$	✓	✓	Float32	✓	✗	✓	–	$1\times$
LOMO	$2M + \frac{2M}{K}$	✓	✗	Float16	✗	✗	✗	–	$1\times$
LoRA	$2M + \frac{36rM}{m}$	✗	✓	Float32	✓	✗	✓	–	$O(\frac{r}{m})$
BAdam	$2M + \frac{16M}{K}$	✓	✓	Float32	✓	✗	✗	$K \leq L$	$\approx 1\times$
HiFT	$2M + \frac{16M}{K}$	✓	✓	Float32	✓	✗	✓	$K \leq L$	$\approx 1\times$
APOLLO	$6M + \mathcal{O}(rM/m)$	✓	✗	Float32	✓	✗	✓	–	$1\times$
CHUNKFT	$2M + \frac{16M}{K}$	✓	✓	Float32	✓	✓	✓	$K \gg L$	$\approx 1\times$

Nevertheless, these methods suffer from two key shortcomings: **GPU memory fluctuations** and **wasteful computational overhead**. In language models, layer sizes vary across embeddings, attention layers, and MLP blocks, causing the mutable-state footprint to fluctuate and leading to unstable memory peaks that severely limit GPU usability.

To decouple update granularity from layer boundaries and avoid dense parameter-gradient materialization within partially active tensors, we seek a method that (i) partitions trainable tensors at a finer granularity than individual layers using their actual byte-level training cost, (ii) materializes gradients only on the active support during backward, and (iii) keeps optimizer residency, state placement, and activation re-materialization synchronized to that same support so that memory savings are accompanied by a stable temporal memory profile.

We pursue this perspective through CHUNKFT, a memory-centric fine-tuning framework that activates only a scheduled subset of trainable parameter slices at each optimization step. Chunks are balanced by a byte budget rather than parameter count. We implement arbitrary sub-tensor gradient updates within the PyTorch framework. During BP, CHUNKFT materializes gradients only for the active slices into auxiliary buffers. To accelerate training, we use asynchronous prefetching and offloading to overlap state transfers with computation. Fine-tuning is thereby transformed from a globally dense procedure into a locally active one, with memory residency concentrated on a rotating working set rather than the full parameter space. This design yields both practical and conceptual benefits. Practically, it lowers peak device memory while preserving the original training objective and remaining compatible with standard transformer training pipelines. Furthermore, a theoretical analysis is provided to confirm the convergence of ChunkFT.

Our contributions are threefold: **(1)** We propose a memory-efficient fine-tuning framework CHUNKFT, which reformulates full-parameter fine-tuning around a dynamically activated working set. We provide a convergence analysis for ChunkFT, demonstrating that ChunkFTs update rule yields a convergent scheme. **(2)** We develop chunk-local training operators that enable gradient computation on arbitrary sub-tensors without modifying the original network architecture, making it broadly applicable to existing standard deep learning frameworks and transformer architectures. **(3)** We apply CHUNKFT to fine-tune Llama3-8B and Llama3-70B models using *a single* RTX 4090-24GB GPU and 2× H800-80GB GPUs, respectively. Extensive evaluations on language understanding, mathematical reasoning, and MT-Bench benchmarks show that CHUNKFT consistently outperforms existing memory-efficient baselines while achieving performance comparable to full-parameter fine-tuning.

2 Related work

Parameter-Efficient Fine-Tuning. LoRA (Low-Rank Adaptation) reduces memory usage by optimizing low-rank adaptation matrices instead of all pretrained weights [8]. By restricting updates to a low-dimensional subspace, LoRA substantially lowers the number of trainable parameters and preserves optimizer states for the adapted parameters. Several LoRA variants have been proposed to enhance performance and support multi-task learning [21, 22, 23, 24]. Although these methods can achieve competitive results, their limited representation space constrains model performance [14].

Algorithm 1 ChunkFT: Rotating Chunk Full-Parameter Optimization

Require: Model parameters θ, number of chunks K, loss $\mathcal{L}$, learning rate η, AdamW hyperparameters $(\beta_1, \beta_2, \epsilon, \lambda)$, optimizer steps per active chunk: T

Estimate byte-level training cost $B(\theta_l)$ for each parameter tensor θ_l
Partition trainable parameters into byte-balanced chunks $\{\mathcal{C}_k\}_{k=0}^{K-1}$ based on $B(\theta_l)$
for $k = 0$ to $K-1$ **do**
    Initialize CPU-resident fp32 master weights $\theta_{\text{CPU}}^{(k)}$ *// for mixed precision*
    Initialize CPU-resident AdamW states $m_{\text{CPU}}^{(k)}, v_{\text{CPU}}^{(k)}$ *// supports other optimizers*
    Set local update counter $n^{(k)} \leftarrow 0$
end for
Add chunk-aware backpropagation for modules such as Embedding, Linear, and LayerNorm
Initialize active index $\mathcal{I} \leftarrow 0$
for training step $t = 0, 1, 2, \ldots$ **do**
    if $t \bmod T = 0$ **then**
        Select active chunk $k \leftarrow \mathcal{I} \bmod K$
        Asynchronously load $\theta_{\text{CPU}}^{(k)}, m_{\text{CPU}}^{(k)}, v_{\text{CPU}}^{(k)}$ to GPU
        Enable gradient accumulation only for parameter slices $\theta^{(k)}$ in $\mathcal{C}_k$
    end if
    Compute prediction $\hat{y} \leftarrow f(x; \theta)$
    Compute loss $\ell \leftarrow \mathcal{L}(\hat{y}, y)$
    Run chunk-aware backward to obtain $g^{(k)} = \nabla_{\theta^{(k)}} \ell$ instead of dense θ_{grad}
    $n^{(k)} \leftarrow n^{(k)} + 1; \quad m^{(k)} \leftarrow \beta_1 m^{(k)} + (1-\beta_1) g^{(k)}$
    $v^{(k)} \leftarrow \beta_2 v^{(k)} + (1-\beta_2)(g^{(k)})^2; \quad \hat{m}^{(k)} \leftarrow m^{(k)} / (1 - \beta_1^{n^{(k)}})$
    $\hat{v}^{(k)} \leftarrow v^{(k)} / (1 - \beta_2^{n^{(k)}}); \quad \theta^{(k)} \leftarrow \theta^{(k)} - \eta \left(\hat{m}^{(k)} / (\sqrt{\hat{v}^{(k)}} + \epsilon) + \lambda \theta^{(k)} \right)$
    Clear temporary chunk gradients
    if $t \bmod T = T - 1$ **then**
        Asynchronously offload updated $\theta^{(k)}, m^{(k)}, v^{(k)}$ to CPU
        Release GPU-resident states of chunk k after offload completes
        $I \leftarrow I + 1$
    end if
end for

Memory-efficient Full Fine-Tuning. MEFT typically optimizes either the optimizer design or the backpropagation process, reducing the memory footprint of gradients, optimizer states, and activations to enable efficient full-parameter fine-tuning [4, 16]. LOMO [4] reduces memory usage by fusing gradient computation with parameter updates and avoiding the storage of conventional optimizer states. While this enables low-memory FFT, it sacrifices momentum and second-moment estimates and does not naturally support gradient accumulation, which may affect optimization stability. BAdam [18] and HiFT [19] further explore partition-based training, where only a subset of parameters or layers is updated at each step. Recent work also studies optimizer-state compression [25, 17, 26]. APOLLO [17] reduces optimizer memory by using low-rank or projected optimizer statistics, thereby improving memory efficiency while retaining full-parameter training capability.

As shown in Table 1, CHUNKFT partitions parameters into fine-grained rotating chunks and updates only a sparse subset of parameters at each step. Unlike LOMO, CHUNKFT retains momentum and second-moment states and supports gradient accumulation. Unlike BAdam and HiFT, its partition granularity is not restricted to layer-wise decomposition, allowing the number of chunks (K) to be much larger than the number of layers (L). As a result, CHUNKFT combines full-parameter trainability, optimizer-state continuity, sparse gradients, and controllable memory usage, while restricting parameter-gradient computation to the active slices. Over one coverage cycle with one update per chunk, their aggregate parameter-gradient cost is approximately that of one dense update. These properties make CHUNKFT an alternative to existing memory-efficient fine-tuning techniques.

3 Method

We present CHUNKFT, a memory-centric optimization framework for large-scale transformer adaptation. For each parameter tensor, we estimate its size as numel $\times$ element_size and accumulate parameters in model order until a predefined byte budget is reached. Large matrices that exceed the

budget are split along the row dimension, allowing one tensor to span multiple chunks. During training, only the active chunk is updated while the others remain frozen. This yields a training regime in which memory is organized around an active working set rather than the full trainable parameter space.

3.1 Algorithm Description

Algorithm 1 summarizes the training procedure of CHUNKFT. Given trainable parameters θ, CHUNKFT first estimates the byte-level training cost $B(\theta_l)$ for each parameter tensor θ_l, including parameter storage, gradient buffer, fp32 master copy, and optimizer states. The trainable parameters are then partitioned into K byte-balanced chunks $\{\mathcal{C}_k\}_{k=0}^{K-1}$, so that each chunk induces a comparable training-time memory footprint. For each chunk $\mathcal{C}_k$, CHUNKFT initializes CPU-resident fp32 master weights for mixed precision training and optimizer states. For AdamW, these states are the first- and second-moment estimates $m_{\text{CPU}}^{(k)}$ and $v_{\text{CPU}}^{(k)}$. A local update counter $n^{(k)}$ is maintained for each chunk for bias correction, since chunks are updated at different global steps.

During training, CHUNKFT follows a rotating schedule. Given a chunk update interval T, the same chunk is optimized for T consecutive steps before switching to the next one in a round-robin order. When chunk $\mathcal{C}_k$ becomes active, its fp32 master weights and optimizer states are asynchronously loaded from CPU to GPU, and gradient accumulation is enabled only for the parameter slices in $\mathcal{C}_k$. The forward pass still uses the full model, $\hat{y} = f(x;\theta)$, so the training objective remains identical to standard full-parameter fine-tuning. The backward pass still propagates the activation gradients required by the chain rule, but materializes parameter gradients only for the active slices:

$$g^{(k)} = \nabla_{\theta^{(k)}} \mathcal{L}(\hat{y}, y), \tag{1}$$

where $\theta^{(k)}$ denotes the active parameter slices in $\mathcal{C}_k$. Inactive chunks do not allocate gradient buffers. This is implemented by chunk-aware backward operators for memory-dominant modules such as Embedding, Linear, and LayerNorm. The optimizer update is then applied only to the active chunk.

All inactive chunks remain unchanged. After the active chunk has been updated for T steps, its updated master weights and optimizer states are asynchronously offloaded back to CPU, GPU buffers are released, and the active index advances to the next chunk. Thus, CHUNKFT performs full-parameter optimization over time while requiring gradient and optimizer-state residency only for the active chunk at each step. With byte-balanced chunks, the peak optimizer-related GPU memory is approximately

$$\max_k B(\mathcal{C}_k) \approx \frac{1}{K} \sum_l B(\theta_l), \tag{2}$$

up to temporary transfer and buffering overhead.

Convergence result. We provide a convergence analysis for CHUNKFT in the deterministic case, aiming to establish that rotating chunk optimization yields a convergent block-coordinate descent scheme. The formal theorem and proof are provided in Appendix G; here we state the key takeaways.

Theorem 1. *Let the trainable parameters be partitioned into K chunks.* CHUNKFT *cyclically updates one active chunk at a time, performing T inner updates for each active chunk. Let R denote the number of completed chunk rotations, where one rotation means that all K chunks have been updated once. For a sufficiently small learning rate, the average active-chunk gradient satisfies*

$$\frac{1}{RTK} \sum_{r=0}^{R-1} \sum_{i=1}^{K} \sum_{t=0}^{T-1} \left\| \nabla_i \mathcal{L}(\theta_i^{r,t}) \right\|^2 = \mathcal{O}\left(\frac{1}{RTK} \right). \tag{3}$$

Consequently, over completed chunk rotations, CHUNKFT *converges to a first-order stationary point. For fixed K and T, it reaches a δ-approximate stationary point within $\mathcal{O}(\delta^{-2})$ active-chunk updates.*

This result shows that CHUNKFT has the same convergence behavior as block coordinate descent.

3.2 Parameter-Gradient Cost

We compare the parameter-gradient computation costs reported in Table 1, assuming gradient checkpointing is disabled. Let P denote the cost of computing all dense model-parameter gradients once.

A *coverage cycle* selects each trainable parameter for one update; for rotating methods, this accounting unit uses one optimizer step per partition ($T = 1$). For LoRA, coverage refers only to its trainable adapter parameters. This metric excludes activation-gradient propagation, forward computation, optimizer operations, and data transfers.

ChunkFT. Let P_k denote the cost of computing parameter gradients for active chunk $\mathcal{C}_k$. The chunks form a disjoint partition of the trainable parameters. Under an approximately additive parameter-gradient cost model,

$$P^{\text{cycle}}_{\text{CHUNKFT}} = \sum_{k=1}^{K} P_k \approx P, \qquad \frac{P^{\text{cycle}}_{\text{CHUNKFT}}}{P} \approx 1. \tag{4}$$

Thus, CHUNKFT distributes approximately one dense parameter-gradient computation across K steps, materializing only the active slices at each step. Its sub-tensor operators avoid computing a full tensor's parameter gradient when only part of that tensor is active. Byte-balanced chunks control memory allocation, but do not necessarily have equal computation costs; $P_k \approx P/K$ additionally requires approximately uniform computation per allocated byte. For the actual interval T, one rotation contains KT optimizer steps and has parameter-gradient cost approximately TP, under fixed per-step costs.

Comparison with other methods. Adam, LOMO, and APOLLO compute all model-parameter gradients in one step, giving a normalized parameter-gradient cost of approximately $1\times$. LOMO's fused updates change gradient residency, while APOLLO's projection and optimizer operations fall outside this metric. BAdam and HiFT also have approximately $1\times$ parameter-gradient cost over a coverage cycle when their disjoint active partitions cover the model and inactive parameter gradients are not computed. An implementation that computes inactive parameter gradients before masking them incurs additional work; such work is not intrinsic to block-coordinate optimization. For LoRA, the parameter-gradient cost of rank-r adapters associated with an $m \times m$ weight matrix scales as $\mathcal{O}(r/m)$ relative to the dense weight gradient, giving the corresponding entry in Table 1.

The approximately $1\times$ cycle-level parameter-gradient cost is therefore shared by the full-parameter methods, rather than a unique total-computation advantage of CHUNKFT. Its distinction is the finer update granularity and chunk-local gradient materialization. Computing an early chunk's parameter gradient still requires activation-gradient propagation through later layers. This additional cost is excluded consistently for all methods, so the parameter-gradient comparison alone does not imply a $1/K$ reduction in total backward cost or establish an end-to-end speedup. Appendix E provides the detailed parameter-gradient cost comparison.

3.3 Memory Consumption Analysis

We analyze the memory usage of CHUNKFT. Consider a model with M billion trainable parameters, and measure memory in GB. With mixed-precision AdamW, FP16 model parameters require $2M$ memory, while the FP32 master weights, gradients, first-moment states, and second-moment states each require $4M$. Thus, the total optimizer-related memory is approximately

$$2M + 4M + 4M + 4M + 4M = 18M. \tag{5}$$

In contrast, CHUNKFT keeps the full up-to-date FP16 model on GPU, costing $2M$, but materializes FP32 gradients, master weights, and AdamW states only for the active chunk. If the parameters are divided into K equal-sized chunks, the active chunk contains M/K parameters, so these active-chunk states cost

$$4\frac{M}{K} + 4\frac{M}{K} + 4\frac{M}{K} + 4\frac{M}{K} = \frac{16M}{K}. \tag{6}$$

Therefore, the total memory required by CHUNKFT is $2M + \frac{16M}{K}$.

Unlike layer-wise partitioning, CHUNKFT partitions parameters by a byte-level memory budget rather than by layer identity. As a result, K is not limited by the number of transformer layers ($K_{\text{ChunkFT}} \gg K_{\text{BAdam}}$), and the per-chunk GPU memory pressure is more balanced.

Table 2: Actual memory costs of fine-tuning Llama 2-7B with gradient checkpointing on the BoolQ [27] dataset. The maximum input sequence length is 1024. The reported results do not use DeepSpeeds memory optimization techniques. MP indicates mixed precision.

Method	Traing	batch=1	batch=4	batch=8	batch=16	batch=32	batch=64
Vanilla	precision	OOM	OOM	OOM	OOM	OOM	OOM
LoRA(rank=16)	MP	14.22	16.58	20.20	26.21	39.12	64.93
	FP32	13.68	15.55	18.03	23.00	32.94	52.82
Galore(rank=256)	MP	68.71	68.71	69.11	71.27	OOM	OOM
	FP32	69.02	68.71	69.93	76.19	OOM	OOM
BAdam(k=34)	MP	28.99	41.99	58.90	OOM	OOM	OOM
	FP32	18.06	30.41	46.89	OOM	OOM	OOM
HiFT(k=34)	MP	27.55	39.53	54.91	OOM	OOM	OOM
	FP32	18.03	30.37	46.88	OOM	OOM	OOM
APOLLO(rank=1)	MP	53.21	53.21	53.62	53.85	54.09	OOM
	FP32	53.21	53.21	54.44	57.72	64.27	OOM
ChunkFT(k=34)	MP	26.70	26.70	26.71	29.98	47.34	OOM
	FP32	13.72	16.92	21.23	29.85	47.08	OOM

Table 3: Time spent fine-tuning Llama 2-7B with gradient checkpointing on BoolQ dataset (with 1000 training examples and 500 validation examples). All results are obtained from experiments conducted on a single H800-80GB.

Method	batch=1	batch=4	batch=8	batch=16	batch=32
LoRA(rank=16)	3653s	4128s	4154s	4255s	4265s
Galore(rank=128)	4882s	4982s	5050s	5150s	5310s
BAdam(k=34)	4567s	4688s	5690s	—	—
HiFT(k=34)	4936s	5356s	6501s	—	—
APOLLO(rank=1)	4622s	4724s	4854s	4990s	5232s
ChunkFT(k=34)	4480s	4632s	4855s	4913s	5150s

4 Experiment Results

In this section, we evaluate the proposed CHUNKFT for fine-tuning LLMs. The baselines include Adam [29], LOMO [4], LoRA [30], GaLore [16], HiFT [19], BAdam [18], and the recent method APOLLO [17]. All CHUNKFT experiments for RoBERTa-large [31] are conducted on a single RTX 4090-24GB GPU, while the others are conducted on H800-80GB GPUs. Our implementation is based on the Transformers library. The detailed experiment setup can be found in Appendix D.

4.1 Memory Consumption and Wall-clock Running Time

Memory consumption. We report the actual memory consumption of different fine-tuning methods in Table 2. LoRA consumes the least GPU memory due to its non-full-parameter fine-tuning setting. Surprisingly, CHUNKFT achieves memory consumption comparable to LoRA, while requiring the least memory among all full-parameter fine-tuning methods. Compared with GaLore, BAdam, HiFT, and APOLLO, CHUNKFT supports a maximum batch size of 8 on a 24GB GPU for benchmarks with a sequence length of 1024, surpassing all baseline MEFT methods. Under FP32 precision, CHUNKFT consumes only 13.72GB at batch size 1, which is comparable to LoRA, while it remains feasible at batch size 32 where BAdam and HiFT already run out of memory. These results demonstrate that CHUNKFT effectively controls the memory costs, making it practical for fine-tuning large language models under memory-constrained settings. Mixed precision does not always lead to lower memory consumption in our experiments. At small batch sizes, mixed precision can even consume more memory than FP32. This is mainly because automatic mixed-precision training maintains additional FP32 master weights, scaling-related buffers, and casted intermediate tensors for numerical stability, while the activation memory is relatively small when the batch size is low. As a result, the extra overhead introduced by mixed precision may outweigh the memory saved by using lower-precision activations. Nev-

Table 5: Comparison of memory fluctuations across different fine-tuning methods on BoolQ dataset (batch size is 8).

Method	Min	Max	Jitter Rate
LoRA(rank=16)	19.35	20.20	0.04
Galore(rank=128)	68.61	69.11	0.01
BAdam(k=34)	37.59	58.90	0.44
HiFT(k=34)	34.53	54.91	0.46
APOLLO(rank=1)	53.33	53.62	0.01
ChunkFT(k=34)	26.34	26.71	0.01

Table 4: SuperGLUE [28] benchmark scores of RoBERTa-large (standard deviation).

Method	BoolQ	COPA	WSC	RTE	MultiRC	WiC	Average
Adam	86.5	59.5	**68.7**	**87.6**	76.6	70.5	74.9
LoRA	81.5	56.7	62.6	79.8	69.5	59.5	68.3
Galore	85.2	68.3	63.5	74.4	77.2	71.5	73.4
BAdam	85.4	69.4	65.5	76.3	77.3	64.6	73.1
HiFT	87.3	71.5	67.7	83.5	**78.6**	66.5	75.9
ChunkFT	**88.5**(0.4)	**73.6**(0.6)	66.4(0.3)	85.8(0.7)	77.5(0.4)	**73.5**(0.7)	**77.6**

ertheless, as the batch size increases, activation memory becomes more dominant, and mixed precision becomes more effective in reducing the overall memory footprint.

Wall-clock running time comparison. As shown in Table 3, we conduct experiments on fine-tuning the Llama 2-7B model for three epochs with each method and report the average wall-clock time per epoch. Overall, CHUNKFT demonstrates strong efficiency and scalability across different batch sizes. Although LoRA achieves the shortest training time, this is expected since LoRA only updates a small set of low-rank adapter parameters. Compared with BAdam and HiFT, CHUNKFT shows clear advantages. Under comparable settings, CHUNKFT is consistently faster than BAdam and HiFT. For instance, at batch size 8, CHUNKFT takes 4855 seconds, compared with 5690 seconds for BAdam and 6501 seconds for HiFT.

ChunkFT is also competitive with GaLore and APOLLO. It is faster than GaLore across all reported batch sizes. At batch size 32, CHUNKFT takes 5150 seconds, while GaLore requires 5310 seconds. Compared with APOLLO, CHUNKFT has similar runtime and is slightly faster at larger batch sizes. For example, CHUNKFT takes 4913 seconds and 5150 seconds at batch sizes 16 and 32, respectively, compared with 4990 seconds and 5232 seconds for APOLLO. These results indicate that the chunk-wise mechanism of CHUNKFT does not introduce substantial additional overhead and can even improve efficiency. These wall-clock measurements reflect the combined implementation and scheduling costs.

Table 6: SuperGLUE [28] benchmark scores of Llama 2-7B using different optimization methods.

Method	BoolQ	RTE	WIC	MultiRC	SST-2	COPA	Average
Zero-Shot	65.3	50.4	51.3	57.6	81.3	61.5	61.2
ICL	67.4	55.4	53.4	59.5	81.4	84.4	66.9
MeZO	76.2	72.3	61.6	70.2	95.4	86.5	77.0
S-MeZO	81.4	80.4	64.5	73.3	95.5	87.5	80.4
LoRA	84.3	83.4	67.4	80.3	94.3	86.5	82.7
Adam	86.5	84.5	69.5	80.3	95.4	85.6	83.6
Galore	84.3	84.3	68.4	81.3	94.3	86.5	83.2
BAdam	85.6	83.5	65.6	80.3	95.4	84.4	82.5
HiFT	**86.7**	84.7	67.6	81.5	94.7	86.5	83.6
APOLLO	86.5	84.1	70.3	81.6	94.3	86.2	83.8
ChunkFT	85.7(0.3)	**85.7**(0.5)	**71.2**(0.6)	**82.7**(0.5)	**96.5**(0.6)	**87.6**(0.5)	**84.9**

Memory jitter comparison. Table 5 compares the memory fluctuation of different fine-tuning methods. Overall, CHUNKFT exhibits highly stable memory usage, with memory varying only from 26.34GB to 26.71GB and a jitter rate of 0.01. This is comparable to LoRA, GaLore, and APOLLO, which also achieve a jitter rate of 0.01, while requiring substantially less memory than both methods. In particular, CHUNKFT reduces the maximum memory consumption by 61.35% compared with GaLore and by 50.18% compared with APOLLO.

In contrast, BAdam and HiFT show much larger memory fluctuations. BAdam has a jitter rate of 0.44, with memory increasing from 37.59GB to 58.90GB, while HiFT has a jitter rate of 0.46, increasing from 34.53GB to 54.91GB. In LLMs, parameter sizes can vary substantially across layers/blocks, which is a major cause of memory jitter in HiFT and BAdam. CHUNKFT maintains a much narrower memory range, suggesting that its stream mechanism effectively mitigates memory thrashing caused by parameter-size variations across different layers/blocks.

Table 7: Math benchmark scores of Llama 3-8B using different optimization methods.

Method	GSM8K	Aqua	MMLU-Math	SAT-Math	MATH	NumGLUE	Average
Vanilla	25.9	22.8	33.7	39.5	12.8	34.5	28.2
Adam	54.5	40.8	44.3	51.4	18.4	55.4	44.1
LoRA	47.5	44.9	45.3	50.9	14.5	**56.9**	43.3
LOMO	32.1	28.0	40.0	39.5	13.1	37.1	31.6
Galore	33.1	37.4	41.2	42.7	15.0	36.9	34.4
BAdam	48.1	42.5	50.5	56.8	15.7	53.0	44.4
HiFT	51.5	43.3	48.4	55.5	16.3	54.5	44.9
APOLLO	53.6	**45.6**	50.7	55.4	18.6	55.6	46.6
ChunkFT	**54.9**(0.4)	**45.6**(0.3)	**51.4**(0.5)	**57.8**(0.5)	**19.9**(0.7)	56.4(0.4)	**47.7**

Table 8: Math benchmark scores of Llama 3-70B using different optimization methods.

Method	GSM8K	Aqua	MMLU-Math	SAT-Math	MATH	NumGLUE	Average
Vanilla	52.4	46.5	52.2	58.2	21.2	37.9	44.7
LoRA	73.3	59.5	58.3	64.1	34.2	64.8	59.0
BAdam	**78.2**	62.5	63.5	76.5	25.5	65.7	62.0
APOLLO	75.6	63.6	63.7	73.4	28.6	64.8	61.6
ChunkFT	77.9(0.5)	**65.6**(0.7)	**65.4**(0.4)	**76.8**(1.2)	**29.9**(1.1)	**66.4**(0.6)	**63.7**

4.2 Downstream Performance Evaluation.

Natural language understanding. Tables 4 and 6 report the performance of different optimization methods on natural language understanding benchmarks. On RoBERTa-large, CHUNKFT obtains the highest average score of 77.6, outperforming Adam, LoRA, GaLore, BAdam, and HiFT. In particular, CHUNKFT achieves the best results on BoolQ, COPA, and WiC, with scores of 88.5, 73.6, and 73.5, respectively. Compared with Adam FFT, CHUNKFT improves the average score from 74.9 to 77.6, while also yielding notable gains on BoolQ and COPA. On Llama 2-7B, CHUNKFT also achieves the best average score of 84.9. It obtains the highest scores on RTE, WiC, MultiRC, SST-2, and COPA, and remains competitive on BoolQ. Compared with Adam and HiFT, both of which achieve an average score of 83.6, CHUNKFT improves the average performance by 1.3 points. It also outperforms APOLLO, GaLore, BAdam, and LoRA, indicating that CHUNKFT is not only memory- and time-efficient, but also preserves strong downstream task performance. These results suggest that CHUNKFT provides a favorable trade-off between efficiency and effectiveness.

Math benchmarks. Tables 7 and 8 report the results on math benchmarks for Llama 3-8B and Llama 3-70B. On Llama 3-8B, CHUNKFT attains the highest average score of 47.7, outperforming all baselines, including APOLLO (46.6), HiFT (44.9), BAdam (44.4), and Adam (44.1). Specifically, CHUNKFT achieves the best or tied-best performance on 5 out of 6 benchmarks. The advantage of CHUNKFT becomes even clearer on Llama 3-70B. CHUNKFT achieves the best overall average of 63.7, surpassing BAdam (62.0), APOLLO (61.6), and LoRA (59.0). It obtains the best results on 5 out of 6 benchmarks. These results suggest that CHUNKFT provides a favorable trade-off between optimization efficiency and downstream reasoning performance. Furthermore, the small variances shown in parentheses indicate that CHUNKFT is not only effective but also stable across runs. Overall, the results on both Llama 3-8B and Llama 3-70B verify that CHUNKFT is a robust and scalable optimization method for mathematical reasoning tasks.

Table 9: MT-Bench scores of instruction-tuned Llama 2-7B and Llama 3-8B on Alpaca-GPT4.

Model	Vanilla	Adam	LOMO	LoRA	Galore	BAdam	HiFT	APOLLO	ChunkFT
Llama 2-7B	3.9	5.3	4.1	4.9	4.9	4.9	5.6	5.4	**5.6**
Llama 3-8B	5.5	6.3	5.8	6.2	5.7	6.7	6.6	6.6	**6.8**

MT-bench results. Table 9 reports the MT-Bench scores of instruction-tuned Llama 2-7B and Llama 3-8B on Alpaca-GPT4. Overall, CHUNKFT achieves the best performance on both models, showing that the proposed method is effective for instruction-following generation. For Llama 2-7B, CHUNKFT obtains an MT-Bench score of 5.6, matching HiFT and outperforming Adam, LoRA, GaLore, BAdam, APOLLO, LOMO, and the vanilla model. Compared with Adam FFT, CHUNKFT improves the score from 5.3 to 5.6. For Llama 3-8B, CHUNKFT achieves the highest score of 6.8, outperforming all baselines. In particular, it improves over Adam by 0.5 points, over LoRA by 0.6 points, and over APOLLO by 0.2 points. Although BAdam and HiFT also achieve strong performance, CHUNKFT still provides the best overall result. These results demonstrate that CHUNKFT

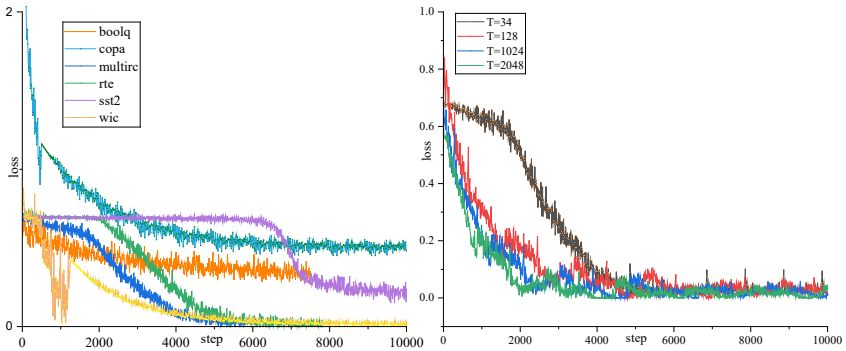

Figure 1: Loss convergence behavior of CHUNKFT (Llama 2-7B). **Left:** training loss curves of CHUNKFT on six natural language understanding tasks. **Right:** loss curves of training a single block with different update intervals T on BoolQ.

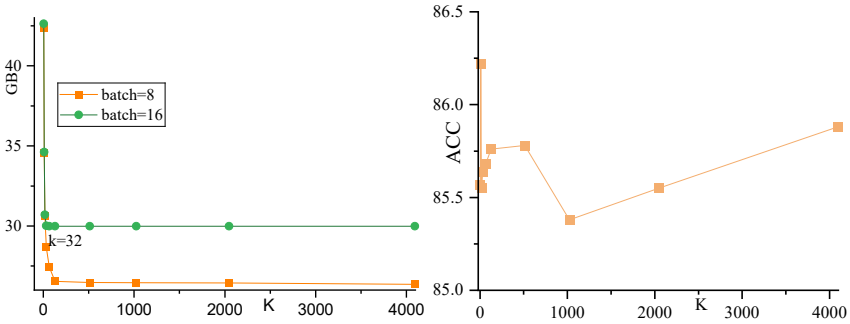

Figure 2: Effect of the chunk number K on memory and performance. **Left:** peak GPU memory under different chunk numbers K. **Right:** BoolQ accuracy under different chunk numbers K.

achieves top MT-Bench scores, confirming its effectiveness as a practical optimization method for instruction tuning.

4.3 Ablation Study

Loss convergence. Figure 1 (left) shows the training loss curves of CHUNKFT on six natural language understanding tasks. Overall, CHUNKFT exhibits stable convergence behavior across all tasks, with the loss decreasing steadily throughout training. Specifically, SST-2, WiC, and COPA converge relatively quickly to near-zero loss, while BoolQ and RTE decrease more gradually and plateau at higher loss values. MultiRC also shows a slower decay pattern, reflecting its relatively higher complexity. These results indicate that the chunk-wise optimization strategy of CHUNKFT maintains stable and effective training dynamics across a diverse set of tasks.

Update intervals. Figure 1 (right) studies the effect of the update interval T when optimizing a single block. The results show that different choices of T do not substantially affect the overall convergence trend: all settings eventually converge to a low loss region and exhibit stable optimization behavior. This indicates that CHUNKFT is not sensitive to the exact update interval in terms of final convergence. However, the update interval can influence the convergence speed. When T is small, such as T=34, blocks are switched more frequently, which may interrupt local optimization and slow down the early-stage loss decrease. In contrast, larger intervals allow each active block to be optimized for more consecutive steps, leading to faster initial convergence. Therefore, while the update interval has limited impact on whether CHUNKFT converges, overly frequent block switching may reduce optimization efficiency.

Impact of chunk number on memory. Figure 2 (left) shows the effect of the chunk number K on GPU memory usage for Llama 2-7B on BoolQ under batch sizes 8 and 16. Overall, increasing K significantly reduces memory consumption. Moreover, after K reaches a moderate value, the memory footprint becomes nearly flat, suggesting that further increasing the number of chunks brings diminishing returns. The reason is that, as the number of chunks increases, fewer parameters

are active at each step, which reduces the amount of optimizer state and intermediate memory that needs to be materialized simultaneously. Such small variations are negligible in practice.

Impact of chunk number on performance. Figure 2 (right) presents the corresponding BoolQ accuracy under different chunk numbers K. We observe that the performance remains highly stable across a wide range of chunk settings. Specifically, all results fall within a narrow range of less than 1% point, which is within normal experimental fluctuation. Although there are small variations across different K values, no clear degradation trend is observed as the model is divided into more chunks. This suggests that the chunk-wise training strategy does not materially harm optimization quality or downstream generalization. Combined with the memory results, these findings indicate that CHUNKFT achieves a favorable trade-off: it substantially reduces memory usage while maintaining essentially unchanged task performance.

5 Conclusion

In this paper, we propose ChunkFT, an efficient and scalable fine-tuning method for large language models. CHUNKFT partitions model parameters into byte-stream chunks and selectively activates only a subset of parameters during each update, thereby reducing unnecessary gradient computation and memory consumption. Its byte-stream allocation further balances chunk sizes and mitigates memory jitter caused by layer-wise parameter variations. In addition, CHUNKFT supports gradient updates on arbitrary sub-tensors, decoupling the update granularity from the model architecture and enabling flexible, stable, and resource-efficient fine-tuning.

Acknowledgments and Disclosure of Funding

This work is supported by the National Natural Science Foundation of China (Grant No.62502081, No.62676083, No.62402293, No.62506216), and the Fundamental Research Funds for the Central Universities (Grant No. N2523011).

References

[1] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th annual meeting of the association for computational linguistics*, pages 7871–7880, 2020.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pages 4171–4186, 2019.

[3] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[4] Kai Lv, Yuqing Yang, Tengxiao Liu, Qipeng Guo, and Xipeng Qiu. Full parameter fine-tuning for large language models with limited resources. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8187–8198, 2024.

[5] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *Advances in neural information processing systems*, 36:10088–10115, 2023.

[6] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[7] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: international conference for high performance computing, networking, storage and analysis*, pages 1–16. IEEE, 2020.

[8] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022.

[9] Elad Ben Zaken, Yoav Goldberg, and Shauli Ravfogel. Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked language-models. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 1–9, 2022.

[10] Jiayang Yu, Yihang Zhang, Bin Wang, Peiqin Lin, Yongkang Liu, and Shi Feng. Ssmlora: Enhancing low-rank adaptation with state space model. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 4493–4506, 2025.

[11] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *International conference on machine learning*, pages 2790–2799. PMLR, 2019.

[12] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4582–4597, 2021.

[13] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. In *Proceedings of the 2021 conference on empirical methods in natural language processing*, pages 3045–3059, 2021.

[14] Yongkang Liu, Xingle Xu, Ercong Nie, Zijing Wang, Shi Feng, Daling Wang, Qian Li, and Hinrich Schütze. Look within or look beyond? a theoretical comparison between parameter-efficient and full fine-tuning. *arXiv preprint arXiv:2505.22355*, 2025.

[15] Kai Lv, Hang Yan, Qipeng Guo, Haijun Lv, and Xipeng Qiu. Adalomo: Low-memory optimization with adaptive learning rate. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 12486–12502, 2024.

[16] Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. Galore: Memory-efficient llm training by gradient low-rank projection. In *Forty-first International Conference on Machine Learning*.

[17] Hanqing Zhu, Zhenyu Zhang, Wenyan Cong, Xi Liu, Sem Park, Vikas Chandra, Bo Long, David Z Pan, Zhangyang Wang, and Jinwon Lee. Apollo: Sgd-like memory, adamw-level performance. *Proceedings of Machine Learning and Systems*, 7, 2025.

[18] Qijun Luo, Hengxu Yu, and Xiao Li. Badam: A memory efficient full parameter optimization method for large language models. *Advances in Neural Information Processing Systems*, 37:24926–24958, 2024.

[19] Yongkang Liu, Yiqun Zhang, Qian Li, Tong Liu, Shi Feng, Daling Wang, Yifei Zhang, and Hinrich Schütze. Hift: A hierarchical full parameter fine-tuning strategy. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 18266–18287, 2024.

[20] Rui Pan, Xiang Liu, Shizhe Diao, Renjie Pi, Jipeng Zhang, Chi Han, and Tong Zhang. Lisa: Layerwise importance sampling for memory-efficient large language model fine-tuning. *Advances in Neural Information Processing Systems*, 37:57018–57049, 2024.

[21] Wenhan Xia, Chengwei Qin, and Elad Hazan. Chain of lora: Efficient fine-tuning of language models via residual learning. *arXiv preprint arXiv:2401.04151*, 2024.

[22] Chunlin Tian, Zhan Shi, Zhijiang Guo, Li Li, and Chengzhong Xu. Hydralora: An asymmetric lora architecture for efficient fine-tuning. *Advances in Neural Information Processing Systems*, 37:9565–9584, 2024.

[23] Yongkang Liu, Xing Li, Mengjie Zhao, Shanru Zhang, Zijing Wang, Qian Li, Shi Feng, Feiliang Ren, Daling Wang, and Hinrich Schütze. High-rank structured modulation for parameter-efficient fine-tuning. *arXiv preprint arXiv:2601.07507*, 2026.

[24] Yuxuan Hu, Tian Tian, Xiaodong Chen, Zhe Zhao, Tao Tao, Weifang Zhang, Yuanfeng Li, Yuhang Liang, Cuiping Li, Hong Chen, et al. Efficient low-rank adaptation for sparse large language model. *Tsinghua Science and Technology*, 31(4):2292–2303, 2026.

[25] Yushun Zhang, Congliang Chen, Ziniu Li, Tian Ding, Chenwei Wu, Diederik P Kingma, Yinyu Ye, Zhi-Quan Luo, and Ruoyu Sun. Adam-mini: Use fewer learning rates to gain more. *arXiv preprint arXiv:2406.16793*, 2024.

[26] Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 8-bit optimizers via block-wise quantization. In *International Conference on Learning Representations*.

[27] Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. Boolq: Exploring the surprising difficulty of natural yes/no questions. In *Proceedings of the 2019 conference of the north American chapter of the association for computational linguistics: Human language technologies, volume 1 (long and short papers)*, pages 2924–2936, 2019.

[28] Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. Superglue: A stickier benchmark for general-purpose language understanding systems. *Advances in neural information processing systems*, 32, 2019.

[29] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[30] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Liang Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *Iclr*, 1(2):3, 2022.

[31] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[32] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623, 2023.

[33] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[34] Wang Ling, Dani Yogatama, Chris Dyer, and Phil Blunsom. Program induction by rationale generation: Learning to solve and explain algebraic word problems. In *Proceedings of the 55th annual meeting of the association for computational linguistics (volume 1: Long papers)*, pages 158–167, 2017.

[35] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

[36] Wanjun Zhong, Ruixiang Cui, Yiduo Guo, Yaobo Liang, Shuai Lu, Yanlin Wang, Amin Saied, Weizhu Chen, and Nan Duan. Agieval: A human-centric benchmark for evaluating foundation models. In *Findings of the association for computational linguistics: NAACL 2024*, pages 2299–2314, 2024.

[37] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

[38] Swaroop Mishra, Arindam Mitra, Neeraj Varshney, Bhavdeep Sachdeva, Peter Clark, Chitta Baral, and Ashwin Kalyan. Numglue: A suite of fundamental yet challenging mathematical reasoning tasks. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3505–3523, 2022.

[39] Melissa Roemmele, Cosmin Adrian Bejan, and Andrew S Gordon. Choice of plausible alternatives: An evaluation of commonsense causal reasoning. In *AAAI spring symposium: logical formalizations of commonsense reasoning*, pages 90–95, 2011.

[40] Hector J Levesque, Ernest Davis, and Leora Morgenstern. The winograd schema challenge. *KR*, 2012(13th):3, 2012.

[41] Ido Dagan, Oren Glickman, and Bernardo Magnini. The pascal recognising textual entailment challenge. In *Machine learning challenges workshop*, pages 177–190. Springer, 2005.

[42] Luisa Bentivogli, Peter Clark, Ido Dagan, and Danilo Giampiccolo. The fifth pascal recognizing textual entailment challenge. *TAC*, 7(8):1, 2009.

[43] Daniel Khashabi, Snigdha Chaturvedi, Michael Roth, Shyam Upadhyay, and Dan Roth. Looking beyond the surface: A challenge set for reading comprehension over multiple sentences. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 252–262, 2018.

[44] Mohammad Taher Pilehvar and Jose Camacho-Collados. Wic: the word-in-context dataset for evaluating context-sensitive meaning representations. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 1267–1273, 2019.

[45] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[46] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[47] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[48] Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 3: system demonstrations)*, pages 400–410, 2024.

[49] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[50] Stephen J Wright. Coordinate descent algorithms. *Mathematical programming*, 151(1):3–34, 2015.

[51] Alexandre Défossez, Léon Bottou, Francis Bach, and Nicolas Usunier. A simple convergence proof of adam and adagrad. *arXiv preprint arXiv:2003.02395*, 2020.

[52] Haochuan Li, Alexander Rakhlin, and Ali Jadbabaie. Convergence of adam under relaxed assumptions. *Advances in Neural Information Processing Systems*, 36:52166–52196, 2023.

A Benchmarks

To comprehensively evaluate the capabilities of different methods, we conduct experiments on a diverse set of benchmarks covering **mathematical reasoning**, **natural language understanding**, and **MT-Bench** [32]. Specifically, the mathematical reasoning benchmarks assess models' ability to solve arithmetic, algebraic, numerical, and competition-level problems through multi-step reasoning. The natural language understanding benchmarks evaluate core linguistic abilities, including reading comprehension, textual entailment, commonsense reasoning, coreference resolution, word sense disambiguation, and sentiment analysis. In addition, we include MT-Bench to measure open-ended dialogue quality and instruction-following performance in multi-turn conversational scenarios. Together, these benchmarks provide a comprehensive evaluation suite that reflects both task-specific reasoning ability and general assistant-oriented language understanding and generation capability.

Mathematical Reasoning Datasets. We evaluate mathematical reasoning ability on six representative benchmarks: GSM8K [33], AQuA [34], MMLU-Math [35], SAT-Math [36], MATH [37], and NumGLUE [38]. These datasets cover a wide range of mathematical reasoning scenarios, from elementary arithmetic word problems to advanced competition-level problem solving and numerical reasoning in natural language.

- **GSM8K.** GSM8K [33] is a grade-school mathematics benchmark consisting of natural language word problems. Each problem typically requires multiple steps of arithmetic reasoning, such as addition, subtraction, multiplication, division, and proportional reasoning. Since the questions are presented in everyday scenarios, GSM8K evaluates not only the model's calculation ability but also its capacity to understand problem contexts, identify relevant quantities, and generate coherent intermediate reasoning steps.
- **AQuA.** AQuA [34] is an algebraic question answering dataset composed mainly of multiple-choice math problems. The questions involve topics such as algebra, equations, ratios, probability, and numerical computation. Solving AQuA problems often requires translating natural language descriptions into mathematical expressions, applying symbolic manipulation or equation solving, and selecting the correct answer from candidate options. It is commonly used to evaluate mathematical reasoning under a standardized multiple-choice setting.
- **MMLU-Math.** MMLU-Math [35] refers to the mathematics-related subset of the MMLU benchmark. It covers a broad spectrum of mathematical knowledge, including elementary mathematics, high-school mathematics, abstract algebra, and formal logic. Compared with datasets focused mainly on arithmetic computation, MMLU-Math emphasizes mathematical concept understanding and domain knowledge across different levels of difficulty. It is therefore suitable for assessing a model's general mathematical knowledge and reasoning ability.
- **SAT-Math.** SAT-Math [36] contains math questions derived from standardized SAT-style examinations. The problems cover common exam topics such as algebra, geometry, functions, statistics, probability, and data analysis. These questions typically require concise reasoning, accurate application of mathematical rules, and familiarity with standardized test formats. SAT-Math evaluates whether a model can solve structured exam-style problems that resemble real educational assessment scenarios.
- **MATH.** MATH [37] is a challenging mathematical reasoning benchmark consisting of competition-level problems. It spans diverse mathematical areas, including algebra, number theory, geometry, combinatorics, probability, precalculus, and calculus. Unlike simpler arithmetic datasets, MATH often requires advanced symbolic reasoning, multi-step derivations, theorem application, and sometimes proof-like problem solving. As a result, it serves as a strong benchmark for evaluating the upper-level mathematical reasoning capability of large language models.
- **NumGLUE.** NumGLUE [38] is a benchmark designed to evaluate numerical reasoning in natural language understanding tasks. It focuses on the model's ability to recognize and compare quantities, perform arithmetic operations, infer numerical relations, and integrate numerical information with textual semantics. NumGLUE is particularly useful for testing whether models can handle numbers accurately in realistic language contexts, rather than only solving explicitly formatted math problems.

Natural Language Understanding Datasets. We evaluate natural language understanding ability on several representative benchmarks, including BoolQ [27], COPA [39], WSC [40], RTE [41, 42], MultiRC [43], WiC [44], and SST-2 [45]. These datasets cover a broad range of language understanding scenarios, such as question answering, commonsense reasoning, coreference resolution, textual entailment, reading comprehension, word sense disambiguation, and sentiment classification.

- **BoolQ.** BoolQ [27] is a binary question answering dataset in which each example consists of a passage, a naturally occurring question, and a boolean answer. The questions are typically derived from real user information-seeking queries, and solving them requires models to understand the given passage, locate relevant evidence, and determine whether the answer is true or false. BoolQ is commonly used to evaluate reading comprehension and factual reasoning over short passages.
- **COPA.** COPA [39], the Choice of Plausible Alternatives dataset, is designed to evaluate commonsense causal reasoning. Each instance provides a premise and two candidate alternatives, where the model must select the more plausible cause or effect. Since the correct answer often depends on everyday commonsense knowledge rather than explicit lexical overlap, COPA is useful for assessing whether models can reason about causal relations and real-world events.
- **WSC.** WSC [40], the Winograd Schema Challenge, focuses on coreference resolution under challenging linguistic conditions. Each example contains a sentence with an ambiguous pronoun and two possible antecedents, and the model must identify the correct referent. WSC examples are designed so that simple syntactic or statistical cues are insufficient, requiring commonsense reasoning and careful semantic understanding.
- **RTE.** RTE [28], Recognizing Textual Entailment, is a binary classification task that determines whether a hypothesis can be inferred from a given premise. The dataset evaluates a model's ability to understand semantic relations between sentence pairs, including entailment and non-entailment. It is widely used to assess natural language inference ability and logical consistency in language understanding.
- **MultiRC.** MultiRC [43] is a multi-sentence reading comprehension dataset. Each instance contains a paragraph, a question, and multiple candidate answers, where more than one answer may be correct. Solving MultiRC often requires integrating evidence from multiple sentences, performing compositional reasoning, and distinguishing partially correct answers from fully supported ones. This makes it suitable for evaluating complex reading comprehension beyond single-sentence evidence matching.
- **WiC.** WiC [44], Word-in-Context, is a word sense disambiguation benchmark. Each example presents two sentences containing the same target word, and the model must determine whether the word is used with the same meaning in both contexts. WiC evaluates fine-grained lexical semantic understanding, requiring models to distinguish subtle sense differences based on contextual usage.
- **SST-2.** SST-2 [45], the Stanford Sentiment Treebank binary classification task, evaluates sentiment analysis ability. Each example consists of a sentence from movie reviews, and the model must classify it as expressing either positive or negative sentiment. SST-2 tests whether models can recognize affective polarity, handle sentiment-bearing expressions, and perform robust sentence-level classification.

MT-Bench Dataset. MT-Bench [32] is a multi-turn instruction-following benchmark designed to evaluate the conversational ability of large language models in open-ended dialogue scenarios. Each example typically consists of two turns of user instructions, where the second turn often depends on the context established in the first turn. The benchmark covers diverse task categories, including writing, role play, reasoning, mathematics, coding, extraction, humanities, and STEM-related questions, thereby testing models' abilities in instruction following, context retention, logical reasoning, factual understanding, and response generation. Since MT-Bench contains open-ended questions with multiple possible valid answers, it is commonly evaluated using an LLM-as-a-judge protocol, where a strong evaluator model scores responses according to helpfulness, relevance, accuracy, coherence, and instruction adherence. Overall, MT-Bench provides a practical evaluation setting for measuring whether large language models can serve as capable conversational assistants across diverse multi-turn interaction scenarios.

B Backbone Models

We conduct experiments on four representative backbone models, including Llama 3-8B [46], Llama 3-70B [46], Llama 2-7B [47], and RoBERTa-large [31]. These models cover both decoder-only generative language models and encoder-only discriminative language models, enabling evaluation across different model architectures, parameter scales, and pretraining paradigms.

- **Llama 3-8B.** Llama 3-8B [46] is an 8-billion-parameter decoder-only large language model from the Llama 3 family. It is designed for general-purpose language understanding and generation, and provides a strong balance between performance and computational efficiency. Due to its moderate parameter size, Llama 3-8B is suitable for evaluating reasoning, instruction following, and generation ability under relatively practical resource constraints.
- **Llama 3-70B.** Llama 3-70B [46] is a larger model from the Llama 3 family, containing approximately 70 billion parameters. Compared with Llama 3-8B, it has stronger model capacity and typically demonstrates better performance on complex tasks such as mathematical reasoning, commonsense reasoning, and long-form generation. We include Llama 3-70B to examine how the proposed method performs on a large-scale high-capacity language model.
- **Llama 2-7B.** Llama 2-7B [47] is a 7-billion-parameter decoder-only language model from the Llama 2 family. As an earlier-generation open-weight large language model, it serves as a widely used baseline for evaluating language modeling, reasoning, and instruction-following capabilities. Comparing Llama 2-7B with Llama 3-based models allows us to study the effect of model generation and pretraining improvements on downstream benchmark performance.
- **RoBERTa-large.** RoBERTa-large [31] is an encoder-only Transformer model based on the BERT architecture, pretrained with a robustly optimized masked language modeling objective. Unlike the Llama models, RoBERTa-large is primarily designed for natural language understanding tasks rather than autoregressive text generation. It is commonly used for classification, entailment, reading comprehension, and semantic matching tasks. We include RoBERTa-large as a representative discriminative backbone to evaluate the generality of our method beyond decoder-only large language models.

C Baselines

We compare our method with several representative efficient fine-tuning baselines, including LoRA [30], LOMO [4], GaLore [16], BAdam [18], HiFT [19], and APOLLO [17]. These methods cover both parameter-efficient fine-tuning and memory-efficient full-parameter optimization strategies, providing a comprehensive comparison across different training paradigms.

- **LoRA.** LoRA [30], short for Low-Rank Adaptation, is a widely used parameter-efficient fine-tuning method. Instead of updating all parameters of the pre-trained model, LoRA freezes the original model weights and injects trainable low-rank matrices into selected linear layers. During fine-tuning, only these additional low-rank parameters are optimized, which significantly reduces the number of trainable parameters and optimizer states. Since the learned LoRA weights can be merged into the original model weights after training, LoRA introduces little or no additional inference latency.
- **LOMO.** LOMO [4], or LOw-Memory Optimization, is a memory-efficient optimization method designed for full-parameter fine-tuning of large language models. Unlike parameter-efficient methods that update only a small subset of parameters, LOMO aims to update the full model while reducing memory consumption. It fuses gradient computation and parameter update into a single step, thereby avoiding the storage of large intermediate gradients and optimizer states. This makes full-parameter fine-tuning more feasible under limited GPU memory budgets.
- **GaLore.** GaLore [16], short for Gradient Low-Rank Projection, is a memory-efficient training strategy that performs full-parameter learning through low-rank gradient projection. Instead of directly storing and updating full-rank optimizer states, GaLore projects gradients into a low-rank subspace and applies optimization in the compressed space. This design reduces optimizer memory while still allowing all model parameters to be updated, making it a competitive alternative to both standard full fine-tuning and parameter-efficient fine-tuning methods.

- **BAdam.** BAdam [18] is a memory-efficient full-parameter optimization method based on block coordinate descent with Adam-style updates. Rather than updating all parameters simultaneously, BAdam divides the model into blocks and updates only a subset of parameters at each optimization stage. By maintaining optimizer states only for the currently active block, BAdam substantially reduces GPU memory usage while preserving the benefits of adaptive optimization. It is particularly suitable for fine-tuning large language models when full Adam-based optimization is too memory-intensive.
- **HiFT.** HiFT [19], or Hierarchical Full-parameter Fine-Tuning, is an efficient full-parameter fine-tuning strategy that updates only a subset of model parameters at each training step. Across multiple steps, different parameter subsets are selected so that the full model can eventually be adapted. This hierarchical update mechanism reduces the amount of gradients and optimizer states that need to reside in GPU memory at any given time, while avoiding the introduction of additional trainable modules or inference-time overhead.
- **APOLLO.** APOLLO [17], short for Approximated Gradient Scaling for Memory-Efficient LLM Optimization, is a memory-efficient optimizer designed to achieve AdamW-level performance with memory consumption closer to SGD. APOLLO approximates adaptive learning-rate scaling using auxiliary low-rank optimizer states based on random projection, thereby greatly reducing the memory overhead associated with Adam-style second-order statistics. It can be applied to both large-scale pre-training and full-parameter fine-tuning, and serves as a strong baseline for memory-efficient optimization of large language models.

D Implementation Details

For the baseline methods, including GaLore, BAdam, and APOLLO, we implement all experiments using the LLaMA-Factory framework [48]. RoBERTa-large experiments are conducted on a server equipped with NVIDIA RTX 4090 GPUs, while all other methods are evaluated on NVIDIA H800-80GB GPUs. All reported experimental results are averaged over five random seeds: 7, 42, 123, 1234, and 12345.

To ensure a fair comparison of memory efficiency, we report peak GPU memory usage for all methods. The memory comparison is mainly intended to demonstrate the memory advantage of our proposed method. For APOLLO, we only compare the memory usage under the rank-1 setting. For ChunkFT, we empirically recommend setting the hyperparameters K and T to the same value. All experiments are conducted with gradient checkpointing enabled, while DeepSpeed was not used. Peak GPU memory consumption was measured using the torch.cuda.max_memory_allocated() function. Other hyperparameters and training configurations are provided in Tables 10, 11, and 12.

Throughout the algorithm and analysis, K denotes the number of chunks and T the number of consecutive optimizer steps on each active chunk; one complete rotation requires KT optimizer steps. The corrected Llama 3-70B configuration in Table 12 is $K = 200$ and $T = 200$ for CHUNKFT, and $K = 82$ for HiFT. In the Llama 3-70B BoolQ coverage experiment, training used 120,000 optimizer steps, completing three rotations of 40,000 steps each. All 200 chunks were visited, and each was updated for 600 steps. These coverage statistics refer specifically to that BoolQ run, not to the mathematical-reasoning results in Table 8.

E Parameter-Gradient Cost Comparison

This section details the parameter-gradient costs in Table 1 using the accounting convention of Section 3.2. We assume gradient checkpointing is disabled and count only the computation of gradients with respect to trainable parameters. Activation-gradient propagation, forward passes, optimizer operations, and transfers are excluded for every method. Let P be the cost of computing all dense model-parameter gradients once. A coverage cycle selects every trainable parameter for one update, using one optimizer step per partition ($T = 1$) for rotating methods. LoRA's coverage is restricted to its adapter parameters. A coverage cycle does not imply equivalent optimization progress across methods.

Table 10: Hyperparameters of SuperGLUE tasks and instruction tuning.

Model	Hyperparameter	Value
RoBERTa-large	lr	0.00001
	warm up ratio	0.06
	bz	16
	weight decay	0
	epoch	50
	T in ChunkFT	24
	K in ChunkFT	24
Llama2-7B	lr	0.00001 0.000001
	bs	8
	warm up ratio	0.02
	epoch	3
	weight decay	0.01
	K in ChunkFT	34
	T in ChunkFT	34
	T in Badam	100
	Badam_mode	layer
	Galore rank	256
	Galore update interval	256
	Galore scale	0.25
	Galore proj type	SVD
	K in HiFT	34
	Apollo scale	0.25
	Apollo rank	32

Table 11: Hyperparameters of instruction tuning.

Model	Hyperparameter	Value
Llama2-7B	lr	0.00001
	bs	8
	epoch	3
	weight decay	0.01
	K in ChunkFT	34
	T in ChunkFT	34
	T in Badam	100
	Badam_mode	layer
	Galore rank	256
	Galore update interval	256
	Galore scale	0.25
	Galore proj type	SVD
	K in HiFT	34
	Apollo scale	0.25
	Apollo rank	32
Llama3-8B	lr	0.00001
	bs	4 or 8
	epoch	3
	weight decay	0.01
	K in ChunkFT	34
	T in ChunkFT	34
	T in Badam	100
	Badam_mode	layer
	Galore rank	256
	Galore update interval	256
	Galore scale	0.25
	Galore proj type	SVD
	K in HiFT	34
	Apollo scale	0.25
	Apollo rank	32

Adam. Adam computes all model-parameter gradients in a single step. Its normalized parameter-gradient cost is

$$\frac{P_{\text{Adam}}^{\text{cycle}}}{P} = 1. \tag{7}$$

Moment estimation and parameter updates are optimizer operations and are outside this metric.

Table 12: Hyperparameters of math fine-tuning.

Model	Hyperparameter	Value
Llama3-8B	lr	0.0001(LOMO) 0.000001(Adam) 0.00001(others)
	bs	8
	epoch	3
	weight decay	0.01
	K in ChunkFT	34
	T in ChunkFT	34
	T in Badam	100
	Badam_mode	layer
	Galore rank	256
	Galore update interval	256
	Galore scale	0.25
	Galore proj type	SVD
	K in HiFT	34
	Apollo scale	0.25
	Apollo rank	32
Llama3-70B	lr	0.00001
	bs	1 or 2
	grad.accu.	8
	epoch	3
	T in Badam	100
	Badam_mode	layer
	Galore rank	256
	Galore update interval	256
	Galore scale	0.25
	Galore proj type	SVD
	K in HiFT	82
	K in ChunkFT	200
	T in ChunkFT	200
	Apollo scale	0.25
	Apollo rank	8

LOMO. LOMO fuses gradient computation with parameter updates and releases gradients as training proceeds. It still computes gradients for all model parameters, giving

$$\frac{P_{\mathrm{LOMO}}^{\mathrm{cycle}}}{P} \approx 1. \tag{8}$$

Fusion reduces gradient residency but does not eliminate the underlying parameter-gradient computation.

LoRA. For a dense linear weight $W \in \mathbb{R}^{m\times m}$ and n input vectors, computing its parameter gradient costs $\Theta(nm^2)$. LoRA freezes W and trains two rank-r adapter matrices. Computing their parameter gradients costs $\Theta(nmr)$, yielding

$$\frac{P_{\mathrm{LoRA}}^{\mathrm{cycle}}}{P} = \mathcal{O}\left(\frac{r}{m}\right) \tag{9}$$

for comparable square linear layers. This scaling describes adapter parameter gradients, not full-model updates or total backward computation. Activation-gradient propagation through the frozen backbone remains necessary.

BAdam and HiFT. For $j \in \{\mathrm{BAdam}, \mathrm{HiFT}\}$, let K_j be the number of disjoint parameter partitions and $P_{j,k}$ the parameter-gradient cost of the k-th active partition. When these partitions cover all model parameters and inactive parameter gradients are not computed, approximate additivity gives

$$P_j^{\mathrm{cycle}} = \sum_{k=1}^{K_j} P_{j,k} \approx P, \qquad \frac{P_j^{\mathrm{cycle}}}{P} \approx 1. \tag{10}$$

An implementation that first computes inactive parameter gradients and then masks them incurs additional work beyond this model. Such work is not an intrinsic requirement of block-coordinate optimization. The suffix activation-gradient propagation needed to reach an active block is excluded here, just as it is for CHUNKFT; it must not be included only for these baselines.

APOLLO. APOLLO computes dense model-parameter gradients before using projected optimizer statistics. Thus,

$$\frac{P^{\text{cycle}}_{\text{APOLLO}}}{P} = 1. \tag{11}$$

The cost of projection, gradient scaling, and parameter updates belongs to optimizer processing and is excluded consistently with the other methods.

ChunkFT. CHUNKFT partitions the model parameters into disjoint sub-tensor chunks and computes parameter gradients only for the active slices. With P_k denoting the cost for chunk $\mathcal{C}_k$,

$$P^{\text{cycle}}_{\text{CHUNKFT}} = \sum_{k=1}^{K} P_k \approx P, \qquad \frac{P^{\text{cycle}}_{\text{CHUNKFT}}}{P} \approx 1. \tag{12}$$

Its distinction is finer update granularity and avoidance of full-tensor parameter gradients for partially active tensors. This does not make the aggregate parameter-gradient cost lower than that of other full-parameter methods under the same accounting convention. Approximate additivity describes arithmetic work; kernel-launch overheads, tensor shapes, and hardware utilization can affect actual runtime. Byte-balanced memory allocation alone also does not guarantee equal parameter-gradient computation costs across chunks.

Update interval and scope. If a rotating method j performs T_j consecutive optimizer steps per active partition, a complete rotation contains $K_j T_j$ steps. Under fixed per-step costs, the parameter-gradient cost is

$$P^{\text{rotation}}_j \approx T_j \sum_{k=1}^{K_j} P_{j,k} \approx T_j P. \tag{13}$$

Table 1 uses $T_j = 1$ to compare coverage cycles consistently. Its approximately $1\times$ entries for full-parameter methods and $\mathcal{O}(r/m)$ entry for LoRA concern parameter gradients only. All methods still require the activation-gradient propagation dictated by the chain rule, and rotating methods can repeat this work across steps. Consequently, these entries do not establish total-backward or end-to-end runtime ratios; the measured training times are reported separately in Table 3.

F Analysis of Memory Jitter

Besides peak memory, we also analyze memory jitter, which measures the variation of GPU memory across training steps. For a training method with step-wise GPU memory $\mathcal{M}_t$, we define the memory jitter ratio as

$$\text{Jitter} = \frac{\max_t \mathcal{M}_t - \min_t \mathcal{M}_t}{\frac{1}{T}\sum_{t=1}^{T} \mathcal{M}_t}. \tag{14}$$

A smaller value indicates a more stable memory footprint.

For dense methods such as AdamW, LOMO, and LoRA, the training pattern is fixed across steps, so the algorithm-induced memory jitter is small:

$$\text{Jitter}_{\text{Adam/LOMO/LoRA}} \approx 0. \tag{15}$$

For layer-wise rotating methods such as BAdam and HiFT, the active partition is tied to model layers or modules. Let M^{layer}_k denote the number of trainable parameters in the k-th layer-wise partition. The step-wise memory can be approximated as

$$\mathcal{M}^{\text{layer}}_k = 2M + 16M^{\text{layer}}_k + \mathcal{A}_k + \Delta_k, \tag{16}$$

where $\mathcal{A}_k$ denotes activation memory and Δ_k denotes temporary buffers. Since layer-wise partitions can be highly imbalanced, especially for embeddings, MLPs, and output heads, the memory jitter becomes

$$\text{Jitter}_{\text{layer}} = \frac{\max_k \mathcal{M}^{\text{layer}}_k - \min_k \mathcal{M}^{\text{layer}}_k}{\frac{1}{K}\sum_{k=1}^{K} \mathcal{M}^{\text{layer}}_k}. \tag{17}$$

This value can be large when one layer-wise partition contains significantly more parameters or optimizer states than others.

In contrast, CHUNKFT constructs chunks according to byte-level training cost rather than layer identity. For each chunk $\mathcal{C}_k$, we estimate

$$B(\mathcal{C}_k) = B_{\text{param}}(\mathcal{C}_k) + B_{\text{grad}}(\mathcal{C}_k) + B_{\text{master}}(\mathcal{C}_k) + B_{\text{opt}}(\mathcal{C}_k), \tag{18}$$

and partition parameters so that

$$B(\mathcal{C}_k) \approx \frac{1}{K} \sum_{j=0}^{K-1} B(\mathcal{C}_j). \tag{19}$$

The step-wise memory of CHUNKFT is therefore

$$\mathcal{M}_k^{\text{CHUNKFT}} = 2M + B(\mathcal{C}_k) + \mathcal{A}_k + \Delta_k. \tag{20}$$

Ignoring minor variations from activations and temporary buffers, byte-balanced chunking yields

$$\max_k \mathcal{M}_k^{\text{CHUNKFT}} - \min_k \mathcal{M}_k^{\text{CHUNKFT}} \approx 0. \tag{21}$$

More generally, if the chunking imbalance is bounded by

$$\epsilon_B = \max_k \left| B(\mathcal{C}_k) - \frac{1}{K} \sum_{j=0}^{K-1} B(\mathcal{C}_j) \right|, \tag{22}$$

then

$$\text{Jitter}_{\text{CHUNKFT}} \lesssim \frac{2\epsilon_B}{2M + \frac{1}{K}\sum_{j=0}^{K-1} B(\mathcal{C}_j)}. \tag{23}$$

Thus, unlike layer-wise methods whose memory jitter is governed by layer-size imbalance, CHUNKFT explicitly controls memory jitter through byte-balanced chunk construction.

G Convergence Analysis of CHUNKFT

In this section, we provide a convergence analysis of CHUNKFT. The analysis follows the standard non-convex block-coordinate optimization framework and shows that rotating chunk updates converge to a first-order stationary point under smoothness and bounded partial-derivative assumptions.

G.1 Notation

Let the trainable parameters be partitioned into K disjoint chunks:

$$\theta = (\theta_1, \theta_2, \ldots, \theta_K), \tag{24}$$

where θ_i denotes the parameters in the i-th chunk. Let $\mathcal{L}(\theta)$ denote the training objective, which is assumed to be lower bounded by $\mathcal{L}^\star$:

$$\mathcal{L}(\theta) \geq \mathcal{L}^\star, \qquad \forall \theta. \tag{25}$$

CHUNKFT updates chunks in a rotating manner. Here N denotes the total number of chunk epochs (chunk activations), each containing T optimizer steps; thus the run contains NT optimizer steps and $R = \lfloor N/K \rfloor$ completed rotations. The interval T has the same meaning as in Algorithm 1 and the main-text theorem. We use $t = 0, 1, \ldots, N-1$ to index chunk epochs, $i = 1, \ldots, K$ to index chunks, and $h = 0, \ldots, T-1$ to index the inner updates performed on a selected chunk. At chunk epoch t, the active chunk is

$$i_t = (t \bmod K) + 1. \tag{26}$$

For this active chunk, CHUNKFT performs T inner updates before switching to the next chunk. We denote the model parameter after the h-th inner update of chunk epoch t by $\theta^{t,h}$, with

$$\theta^{t,0} \tag{27}$$

being the parameter at the beginning of chunk epoch t. The partial derivative with respect to the active chunk is denoted by

$$g_{i_t}^{t,h} = \nabla_{i_t} \mathcal{L}(\theta^{t,h}). \tag{28}$$

The update rule is

$$\theta_{i_t}^{t,h+1} = \theta_{i_t}^{t,h} - \eta g_{i_t}^{t,h}, \tag{29}$$

and all inactive chunks remain unchanged:

$$\theta_j^{t,h+1} = \theta_j^{t,h}, \qquad j \neq i_t. \tag{30}$$

After T inner updates, we set

$$\theta^{t+1,0} = \theta^{t,T}. \tag{31}$$

For notational simplicity, the proof is written for the block-gradient version of CHUNKFT. The AdamW implementation used in practice can be viewed as a chunk-wise preconditioned variant of this update; we discuss this at the end of the section.

G.2 Assumptions

We make the following two assumptions. Assumption D.1 is standard for analyzing block descent-type methods [50]. Assumption D.2 is commonly used in the analysis of Adam [51]. We adopt this assumption for simplicity of presentation, noting that it can be provably ensured [52].

Assumption D.1. Smoothness. The loss function $\mathcal{L}$ is L-smooth. When restricted to the i-th chunk, it is L_i-smooth. Mathematically, for any θ^1, θ^2,

$$\left\|\nabla \mathcal{L}(\theta^1) - \nabla \mathcal{L}(\theta^2)\right\| \leq L \left\|\theta^1 - \theta^2\right\|, \tag{32}$$

and for each chunk $i = 1, \ldots, K$,

$$\left\|\nabla_i \mathcal{L}(\theta^1) - \nabla_i \mathcal{L}(\theta^2)\right\| \leq L_i \left\|\theta_i^1 - \theta_i^2\right\|. \tag{33}$$

Equivalently, for an update only on chunk i,

$$\mathcal{L}(\theta + U_i d_i) \leq \mathcal{L}(\theta) + \langle \nabla_i \mathcal{L}(\theta), d_i \rangle + \frac{L_i}{2} \|d_i\|^2, \tag{34}$$

where $U_i d_i$ inserts d_i into the i-th chunk and leaves other chunks unchanged.

We define

$$\overline{L} = \max_{i=1,\ldots,K} L_i, \qquad \underline{L} = \min_{i=1,\ldots,K} L_i. \tag{35}$$

Assumption D.2. Bounded partial derivatives. CHUNKFT has bounded partial derivatives along its optimization trajectory. That is, there exists $G > 0$ such that

$$\left\|g_{i_t}^{t,h}\right\| \leq G, \qquad \forall t,\ h. \tag{36}$$

Here

$$g_{i_t}^{t,h} = \nabla_{i_t} \mathcal{L}(\theta^{t,h}) \tag{37}$$

is the partial derivative with respect to the active chunk at chunk epoch t and inner update h.

G.3 One-Step Descent

For one inner update on active chunk i_t, we have

$$\theta^{t,h+1} = \theta^{t,h} - \eta U_{i_t} g_{i_t}^{t,h}. \tag{38}$$

By block-wise smoothness in Assumption D.1,

$$\mathcal{L}(\theta^{t,h+1}) \leq \mathcal{L}(\theta^{t,h}) - \eta \left\|g_{i_t}^{t,h}\right\|^2 + \frac{L_{i_t} \eta^2}{2} \left\|g_{i_t}^{t,h}\right\|^2. \tag{39}$$

Thus,

$$\mathcal{L}(\theta^{t,h+1}) \leq \mathcal{L}(\theta^{t,h}) - \eta \left(1 - \frac{L_{i_t}\eta}{2}\right) \left\| g_{i_t}^{t,h} \right\|^2. \tag{40}$$

If

$$0 < \eta \leq \frac{1}{\overline{L}}, \tag{41}$$

then

$$1 - \frac{L_{i_t}\eta}{2} \geq \frac{1}{2}. \tag{42}$$

Therefore,

$$\mathcal{L}(\theta^{t,h+1}) \leq \mathcal{L}(\theta^{t,h}) - \frac{\eta}{2} \left\| g_{i_t}^{t,h} \right\|^2. \tag{43}$$

Equivalently,

$$\left\| g_{i_t}^{t,h} \right\|^2 \leq \frac{2}{\eta} \left(\mathcal{L}(\theta^{t,h}) - \mathcal{L}(\theta^{t,h+1}) \right). \tag{44}$$

G.4 Average Active-Chunk Stationarity

Summing the one-step descent inequality over all chunk epochs $t = 0, \ldots, N-1$ and inner updates $h = 0, \ldots, T-1$, we obtain

$$\frac{\eta}{2} \sum_{t=0}^{N-1} \sum_{h=0}^{T-1} \left\| g_{i_t}^{t,h} \right\|^2 \leq \mathcal{L}(\theta^{0,0}) - \mathcal{L}(\theta^{N,0}). \tag{45}$$

Since $\mathcal{L}$ is lower bounded by $\mathcal{L}^\star$,

$$\frac{\eta}{2} \sum_{t=0}^{N-1} \sum_{h=0}^{T-1} \left\| g_{i_t}^{t,h} \right\|^2 \leq \mathcal{L}(\theta^{0,0}) - \mathcal{L}^\star. \tag{46}$$

Therefore,

$$\frac{1}{NT} \sum_{t=0}^{N-1} \sum_{h=0}^{T-1} \left\| g_{i_t}^{t,h} \right\|^2 \leq \frac{2\left(\mathcal{L}(\theta^{0,0}) - \mathcal{L}^\star\right)}{\eta NT}. \tag{47}$$

This implies

$$\min_{0 \leq t < N,\ 0 \leq h < T} \left\| g_{i_t}^{t,h} \right\|^2 \leq \frac{2\left(\mathcal{L}(\theta^{0,0}) - \mathcal{L}^\star\right)}{\eta NT}. \tag{48}$$

Hence, as the number of training updates increases, the active-chunk gradient converges to zero in the ergodic sense:

$$\frac{1}{NT} \sum_{t=0}^{N-1} \sum_{h=0}^{T-1} \left\| \nabla_{i_t} \mathcal{L}(\theta^{t,h}) \right\|^2 = \mathcal{O}\left(\frac{1}{NT}\right). \tag{49}$$

G.5 Full-Gradient Stationarity over Completed Rotations

The preceding result controls the gradient of the active chunk at its update time. We now relate it to the full gradient at the beginning of completed rotations.

Let

$$\tau_r = rK \tag{50}$$

denote the beginning chunk epoch of the r-th full rotation, and let

$$R = \left\lfloor \frac{N}{K} \right\rfloor \tag{51}$$

be the number of completed rotations. Within each rotation, every chunk is activated once. For each chunk i, let

$$t_{r,i} \in \{\tau_r, \tau_r + 1, \ldots, \tau_r + K - 1\} \tag{52}$$

be the chunk epoch in which chunk i is activated during rotation r.

By smoothness,

$$\left\|\nabla_i \mathcal{L}(\theta^{\tau_r,0}) - \nabla_i \mathcal{L}(\theta^{t_{r,i},0})\right\| \le L_i \left\|\theta^{\tau_r,0} - \theta^{t_{r,i},0}\right\|. \tag{53}$$

Using

$$\|a\|^2 \le 2\|b\|^2 + 2\|a-b\|^2, \tag{54}$$

we have

$$\left\|\nabla_i \mathcal{L}(\theta^{\tau_r,0})\right\|^2 \le 2\left\|\nabla_i \mathcal{L}(\theta^{t_{r,i},0})\right\|^2 + 2L_i^2 \left\|\theta^{\tau_r,0} - \theta^{t_{r,i},0}\right\|^2. \tag{55}$$

Summing over all chunks gives

$$\left\|\nabla \mathcal{L}(\theta^{\tau_r,0})\right\|^2 \le 2\sum_{i=1}^{K} \left\|\nabla_i \mathcal{L}(\theta^{t_{r,i},0})\right\|^2 + 2\overline{L}^2 \sum_{i=1}^{K} \left\|\theta^{\tau_r,0} - \theta^{t_{r,i},0}\right\|^2. \tag{56}$$

We bound the movement term. During one rotation, at most KT inner updates are performed. By Assumption D.2, each update satisfies

$$\left\|\theta^{t,h+1} - \theta^{t,h}\right\| = \eta \left\|g_{i_t}^{t,h}\right\| \le \eta G. \tag{57}$$

Therefore, for any $t_{r,i}$ in the same rotation,

$$\left\|\theta^{\tau_r,0} - \theta^{t_{r,i},0}\right\| \le KT\eta G, \tag{58}$$

and hence

$$\left\|\theta^{\tau_r,0} - \theta^{t_{r,i},0}\right\|^2 \le K^2T^2\eta^2G^2. \tag{59}$$

Thus,

$$\left\|\nabla \mathcal{L}(\theta^{\tau_r,0})\right\|^2 \le 2\sum_{i=1}^{K} \left\|\nabla_i \mathcal{L}(\theta^{t_{r,i},0})\right\|^2 + 2K^3T^2\overline{L}^2\eta^2G^2. \tag{60}$$

Averaging over R completed rotations, we obtain

$$\frac{1}{R}\sum_{r=0}^{R-1} \left\|\nabla \mathcal{L}(\theta^{\tau_r,0})\right\|^2 \le \frac{2}{R}\sum_{r=0}^{R-1}\sum_{i=1}^{K} \left\|\nabla_i \mathcal{L}(\theta^{t_{r,i},0})\right\|^2 + 2K^3T^2\overline{L}^2\eta^2G^2. \tag{61}$$

Since $R = \lfloor N/K \rfloor$, the first term is controlled by the active-chunk stationarity bound:

$$\frac{1}{R}\sum_{r=0}^{R-1}\sum_{i=1}^{K} \left\|\nabla_i \mathcal{L}(\theta^{t_{r,i},0})\right\|^2 = \mathcal{O}\left(\frac{K}{\eta NT}\right). \tag{62}$$

Therefore,

$$\frac{1}{R}\sum_{r=0}^{R-1} \left\|\nabla \mathcal{L}(\theta^{\tau_r,0})\right\|^2 = \mathcal{O}\left(\frac{K}{\eta NT}\right) + \mathcal{O}\left(K^3T^2\eta^2\right). \tag{63}$$

Choosing

$$\eta = \mathcal{O}\left(\frac{1}{K^{2/3}T\,N^{1/3}}\right) \tag{64}$$

balances the two terms and yields

$$\frac{1}{R}\sum_{r=0}^{R-1} \left\|\nabla \mathcal{L}(\theta^{\tau_r,0})\right\|^2 = \mathcal{O}\left(\frac{K^{5/3}}{N^{2/3}}\right), \tag{65}$$

for fixed T. In particular, for fixed K and T,

$$\liminf_{r\to\infty} \left\|\nabla \mathcal{L}(\theta^{\tau_r,0})\right\|^2 = 0. \tag{66}$$

G.6 Theorem

Theorem 2 (Convergence of CHUNKFT)**.** *Suppose Assumptions D.1–D.2 hold and $\mathcal{L}$ is lower bounded by $\mathcal{L}^\star$. Let* CHUNKFT *update chunks according to the rotating schedule*

$$i_t = (t \bmod K) + 1, \tag{67}$$

and perform T inner updates on each selected chunk. If the learning rate satisfies $0 < \eta \leq 1/\overline{L}$, then

$$\frac{1}{NT} \sum_{t=0}^{N-1} \sum_{h=0}^{T-1} \left\| \nabla_{i_t} \mathcal{L}(\theta^{t,h}) \right\|^2 \leq \frac{2\left(\mathcal{L}(\theta^{0,0}) - \mathcal{L}^\star\right)}{\eta NT}. \tag{68}$$

Furthermore, over completed rotations,

$$\frac{1}{R} \sum_{r=0}^{R-1} \left\| \nabla \mathcal{L}(\theta^{\tau_r,0}) \right\|^2 = \mathcal{O}\left(\frac{K}{\eta NT}\right) + \mathcal{O}\left(K^3 T^2 \eta^2\right), \tag{69}$$

where $\tau_r = rK$ and $R = \lfloor N/K \rfloor$. With an appropriate diminishing learning rate, for fixed K and T, CHUNKFT *converges to a first-order stationary point:*

$$\liminf_{r \to \infty} \left\| \nabla \mathcal{L}(\theta^{\tau_r,0}) \right\|^2 = 0. \tag{70}$$

G.7 Remark on AdamW and CPU Offloading

The proof above is written for block-gradient updates to isolate the effect of rotating chunk optimization. In practice, CHUNKFT uses AdamW with chunk-wise first- and second-moment states. The AdamW update can be written as a preconditioned block update:

$$\theta_{i_t}^{t,h+1} = \theta_{i_t}^{t,h} - \eta P_{t,h}^{(i_t)} g_{i_t}^{t,h} - \eta \lambda \theta_{i_t}^{t,h}, \tag{71}$$

where $P_{t,h}^{(i_t)}$ is a diagonal adaptive preconditioner. If the preconditioner is uniformly bounded and positive, i.e.,

$$0 < p_{\min} I \preceq P_{t,h}^{(i_t)} \preceq p_{\max} I, \tag{72}$$

the same proof applies up to constants depending on $p_{\min}$ and $p_{\max}$.

CPU offloading does not affect the mathematical update. The fp32 master weights and optimizer states of each chunk are stored on CPU and reloaded when the corresponding chunk becomes active. Therefore, offloading changes memory residency and data movement, but not the optimization trajectory defined by the chunk-wise update rule.

G.8 Remark on Stochastic Gradients

For mini-batch training, the same analysis can be extended by replacing the deterministic partial derivative with a stochastic estimator $g_{i_t}^{t,h}$ satisfying

$$\mathbb{E}[g_{i_t}^{t,h} \mid \theta^{t,h}] = \nabla_{i_t} \mathcal{L}(\theta^{t,h}), \qquad \mathbb{E}\|g_{i_t}^{t,h}\|^2 \leq G^2. \tag{73}$$

The resulting convergence bound has the same form in expectation, with the squared gradient norm replaced by its expected value.

H Limitations

CHUNKFT reduces GPU memory by activating and optimizing only one byte-balanced parameter chunk at a time, but this design also introduces several practical considerations. First, CHUNKFT relies on chunk-aware backward implementations for memory-dominant modules such as Embedding, Linear, LayerNorm, and RMSNorm. These operators cover the primary components of modern Transformer architectures, enabling broad applicability in standard LLM training pipelines. Extending CHUNKFT to architectures with highly customized operators may require additional operator-specific implementations.

Second, CHUNKFT keeps optimizer states and fp32 master weights on CPU by default and transfers the active chunk between CPU and GPU during training. Consequently, the end-to-end training efficiency can depend on the underlying system configuration, including host-device interconnect bandwidth, CPU memory bandwidth, and the effectiveness of asynchronous prefetching and offloading. In resource-constrained environments, communication overhead may partially limit the achievable runtime improvement.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction state the main methodological, theoretical, and empirical claims, and these claims are supported by the method description in Section 3, the convergence analysis in Appendix G, and the memory, runtime, and downstream evaluation results in Section 4.1 and the experiment section.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The paper includes Limitations section in Appendix H, discussing the need for chunk-aware backward operators and the dependence on CPU-GPU transfer bandwidth and asynchronous offloading.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.

- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [Yes]

   Justification: The paper states the assumptions for the deterministic convergence result in Appendix G, including smoothness and bounded partial derivatives, and provides the descent argument, stationarity results, theorem, and remarks on AdamW and stochastic gradients.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The algorithm is specified in Algorithm 1, and the experimental setup, benchmarks, baselines, random seeds, hardware, memory measurement protocol, and hyperparameters are described in Section 4.1 and Appendix A–D.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in

the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.

- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: We provide an anonymous code repository.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

Justification: The paper reports the training and evaluation settings, including backbone models, benchmark datasets, baselines, hardware, random seeds, gradient checkpointing, memory measurement with `torch.cuda.max_memory_allocated()`, and hyperparameters in Appendix D.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: Downstream performance tables report variability over five random seeds (Table 4, 6 , 7 and 8).

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

Answer: [Yes]

Justification: The paper reports the main compute resources used for experiments, including RTX 4090-24GB, and H800-80GB GPUs, and provides wall-clock timing for the main Llama 2-7B BoolQ speed comparison in Table 3.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.

- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We adhere to all the ethics guidelines.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [No]

    Justification: The paper focuses on an optimization method and does not include a dedicated broader-impact discussion.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: This paper poses no such risks.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: Existing datasets and code are properly cited.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [N/A]

    Justification: Paper does not exist new assets.

    Guidelines:

- The answer [N/A] means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: This paper does not involve human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: This paper does not involve crowdsourcing or human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [N/A]

Justification: This paper does not involve LLMs in any meaningful capacity.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.