GoCkpt: Gradient-Assisted Multi-Step overlapped Checkpointing for Efficient LLM Training

Keyao Zhang
Zhejiang University & Alibaba Group
Hangzhou, China
zhangkeyao@zju.edu.cn

Yiquan Chen
Alibaba Group
Hangzhou, China
zy.zhengyi@alibaba-inc.com

Zhuo Hu
Zhejiang University
Hangzhou, China
zjullk@zju.edu.cn

Wenhai Lin
Zhejiang University
Hangzhou, China
linwh@zju.edu.cn

Jiexiong Xu
Zhejiang University
Hangzhou, China
jasonxu@zju.edu.cn

Wenzhi Chen*
Zhejiang University
Hangzhou, China
chenwz@zju.edu.cn

Abstract

The accuracy of large language models (LLMs) improves with increasing model size, but increasing model complexity also poses significant challenges to training stability. Periodic checkpointing is a key mechanism for fault recovery and is widely used in LLM training. However, traditional checkpointing strategies often pause or delay GPU computation during checkpoint saving for checkpoint GPU-CPU transfer, resulting in significant training interruptions and reduced training throughput.

To address this issue, we propose GoCkpt, a method to overlap checkpoint saving with multiple training steps and restore the final checkpoint on the CPU. We transfer the checkpoint across multiple steps, each step transfers part of the checkpoint state, and we transfer some of the gradient data used for parameter updates. After the transfer is complete, each partial checkpoint state is updated to a consistent version on the CPU, thus avoiding the checkpoint state inconsistency problem caused by transferring checkpoints across multiple steps. Furthermore, we introduce a transfer optimization strategy to maximize GPU-CPU bandwidth utilization and SSD persistence throughput. This dual optimization—overlapping saves across steps and maximizing I/O efficiency—significantly reduces invalid training time. Experimental results show that GoCkpt can increase training throughput by up to 38.4% compared to traditional asynchronous checkpoint solutions in the industry. We also find that GoCkpt can reduce training interruption time by 86.7% compared to the state-of-the-art checkpoint transfer methods, which results in a 4.8% throughput improvement.

CCS Concepts: • **Computing methodologies** → **Machine learning**; • **Computer systems organization** → **Dependable and fault-tolerant systems and networks**.

Keywords: Machine Learning, LLM, Checkpoint, Gradient, Mixed precision training

*Wenzhi Chen is the corresponding author.

1 Introduction

Large language models (LLMs) have undergone rapid adoption across diverse domains in data centers [16, 19], driven by two key trends: the exponential growth of model parameters and training token volumes. These large autoregressive AI models have demonstrated exceptional performance across a broad range of tasks [5]. This trajectory is reinforced by scaling laws, which have propelled model sizes from hundreds of millions of parameters in early models, such as BERT [9], to over 500 billion in systems like Megatron-Turing NLG [36]. In this context, mixed-precision training has emerged as the de facto standard for scaling large models. Supported by a robust technological ecosystem—including hardware adaptations (e.g., Tensor Cores), algorithmic innovations (e.g., Automatic Mixed Precision (AMP) [14, 44] with gradient scaling), and framework integration (e.g., PyTorch, TensorFlow)—it delivers measurable efficiency gains (e.g., 1.2-2x training speedups) and resource optimizations (e.g., halving GPU memory footprints for model parameters). By reducing forward propagation precision from FP32 to FP16 or BF16, mixed-precision training significantly lowers training memory demands.

However, the rapid growth in model size has led to drastically extended training durations. Prominent case studies illustrate this trend: the BLOOM model [35] required 3.5 months (1 million training hours) to train; LLaMA [39] took 54 days; and OPT-175B [46] (175B parameters) trained for 33 days across 992 NVIDIA A100 GPUs (80GB each) using Fully Sharded Data Parallelism (FSDP) [47] and 8-way Megatron-LM tensor parallelism to process 180B tokens. Critically, this training endured 105+ restarts due to frequent hardware failures (e.g., overheating, power outages) and software issues (e.g., MPI errors, checkpoint corruption), with the longest stable training interval lasting just 2.8 days. Similarly, GLM-130B [45] (130B parameters) required 60 days of training across 96 DGX-A100 nodes (768 GPUs total).

Large model training faces exceptionally high interruption risks. For example, LLaMA3 experienced interruptions averaging every 3 hours over its 54-day training cycle. Reports

from Alibaba's Unicron system indicate a 43.4% failure rate for resource-intensive LLM training jobs, with 37% attributed to hardware failures and 73% of total failures remediable via restarts [17]. Meta's research reveals that 50% of machine learning training job runtime is squandered due to inefficiencies [18], while Microsoft observes an average failure interval of just 45 minutes in multi-tenant GPU clusters [23]. Even under nominal operation, training trajectories often exhibit anomalies—slow convergence, stalls, persistent incorrect feature learning, or severe loss fluctuations—that degrade efficiency. Such pathologies, documented in PaLM [8] and GLM-130B, and observed recurrently in mainstream models (e.g., BLOOM-175B, OPT-175B), are unpredictable and lack effective prevention strategies. Current mitigations primarily involve rolling back to the latest valid checkpoint and applying corrective actions, such as skipping problematic data batches, adjusting parameter precision, or modifying architectural components.

To mitigate resource waste from interruptions, periodic checkpointing has become a cornerstone technique in large-scale training: PaLM employed a multi-layered strategy (10-minute memory snapshots with full parameter saves at specific intervals), while GLM-130B introduced dynamic interval adjustment. These designs reduced resource loss from hardware failures by 20-30%, enhancing overall utilization. Indeed, periodic checkpointing is explicitly cited as critical for training continuity in logs of BLOOM-175B and OPT-175B, validating its efficacy in distributed scenarios. In contrast, aperiodic mechanisms (e.g., Just-In-Time (JIT) [15]) reduce overhead via node-level data-parallel replicas but lack generality across diverse distributed setups because some research suggests only using data-parallel inter-node [11, 36].

However, existing checkpointing mechanisms still face significant efficiency problems. First, computation interrupts remain a critical issue: traditional checkpointing halts or slows down GPU computation during state transfer (especially model/optimizer parameters from GPU to CPU), resulting in low GPU utilization. For instance, saving full model states can take seconds to minutes, leaving GPU compute units idle and directly reducing throughput. Second, synchronization overhead fragments parallelism: existing frameworks [28, 30] rely on device-level synchronization (e.g., pausing all devices to coordinate writes), breaking the training pipeline and underutilizing GPU-CPU bandwidth. These challenges are exacerbated with cost-effective preemptible instances (60-90% cheaper than on-demand), where interruptions are frequent—for example, a 64-spot instance cluster experienced 127 training failures within 24 hours [3].

These inefficiencies require a rethinking of the checkpoint architecture. To address computation interrupts and synchronization overhead, we propose GoCkpt, a system that overlaps checkpoint transfers with multi-step training, improving training efficiency through three key innovations.

• Cross-Step Checkpoint Transfer. We propose a scheme that allows checkpoint snapshots to span multiple training steps. This allows training to continue during a checkpoint snapshot, allowing inconsistent checkpoint versions to be transferred to the CPU, reducing training interruptions caused by the checkpoint system and improving training throughput.

• Reconstructing a consistent checkpoint on the CPU. GoCkpt additionally transfers low-precision gradients generated during mixed-precision training steps and updates inconsistent checkpoint versions to the latest consistent version on the CPU.

• IO bandwidth optimization. GoCkpt introduces I/O optimization techniques to enhance GPU-CPU transmission bandwidth and accelerate data persistence throughput, improving computational resource utilization during training.

We implemented GoCkpt and assessed its performance in comparison to existing systems. The experimental results indicate that this approach surpasses conventional asynchronous checkpointing schemes by 38.4%. Additionally, compared to state-of-the-art checkpoint transfer methods, GoCkpt reduces training interruption time by up to 86.7% and enhances throughput by 4.8%.

The remainder of this paper is organized as follows: section 2 provides the necessary background; section 3 details our motivation analysis; section 4 describes the design and implementation of GoCkpt; section 5 presents evaluation results; section 6 discusses related work; and section 7 concludes.

2 Background

2.1 Process of LLM Training

The LLM training process is inherently iterative, involving cyclical interactions between forward propagation, backward propagation, and parameter updates, as schematically depicted in Figure 1.

At the first part of this workflow lies the Forward pass, where input data traverses the neural network using current model parameters (M^N) and optimizer states (O^N), producing activations that capture hierarchical semantic representations. Subsequently, the Backward pass computes gradients (G^N) of the loss function concerning these parameters, backpropagating errors through the network to quantify sensitivity to input perturbations. Finally, the Update phase leverages these gradients to adjust model weights ($M^N \rightarrow M^{N+1}$) and refine optimizer states ($O^N \rightarrow O^{N+1}$), incorporating techniques such as momentum, gradient clipping, or learning rate scheduling to stabilize training.

A key aspect of this process is versioned state management, where each training step (N) is associated with a different snapshot ($M^N + O^N$) of the model and optimizer parameters. This versioning ensures consistency between iterations, enabling incremental progress while reducing the risk of

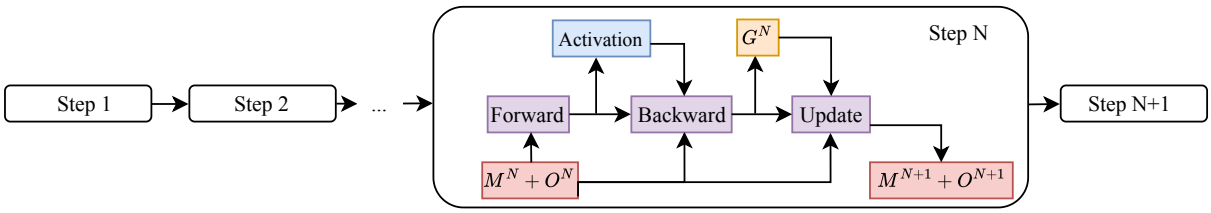


Figure 1. Process of LLM Training (M+O-Model and Optimizer parameters, G-Gradients, N-Model and Optimizer version in step N)

divergence or catastrophic forgetting. Typically, when we checkpoint, we save M^N and O^N as the GPU checkpoint state for the Nth step.

2.2 Mixed-Precision Training

Training billion-parameter models demands not only efficient parallelization but also optimized numerical computation to address memory and speed constraints. Conventional full-precision (FP32) training stores parameters, gradients, and optimizer states (e.g., AdamW's momentum and variance terms) in 32-bit floating-point format, which ensures numerical stability but incurs significant memory overhead—often exceeding GPU/TPU capacity for large models. To address this, mixed-precision training (MPT) has become a cornerstone optimization [14, 25, 44], combining 16-bit (FP16 or Bfloat16) and 32-bit arithmetic to reduce memory usage while preserving model accuracy.

At its core, MPT uses 16-bit arithmetic for computations that tolerate lower precision (e.g., matrix multiplications, forward/backward activation passes) and retains FP32 for operations requiring stability (e.g., gradient updates, optimizer steps). Critically, while optimizer states (e.g., AdamW's m and v) remain in FP32 to avoid instability, a FP32 replica of the model parameters is also kept. The parameters and gradients are stored and computed in 16-bit format. This reduces the memory footprint of parameters and gradients by 50% compared to FP32-only training (since FP16 and Bfloat16 use 2 bytes per value vs. 4 bytes for FP32).

Modern frameworks (e.g., PyTorch's torch.cuda.amp, TensorFlow's tf.keras.mixed_precision) further optimize this by dynamically managing conversions between 16-bit and 32-bit arithmetic during training, ensuring numerical stability (via techniques like loss scaling to prevent gradient underflow) while minimizing overhead. Empirically, MPT has enabled training trillion-parameter models (e.g., GPT-3, PaLM) on thousands of GPUs, where FP32-only approaches would be infeasible due to memory constraints.

2.3 CPU Assisted Parameters Update

Some frameworks, such as DeepSpeed Offload++ and Deep Optimizer States [29], use the CPU to assist the GPU in updating parameters and offloading memory. Similar principles can be effectively applied to optimize checkpointing systems, particularly for state information transmission and storage. By leveraging the CPU, we can pass the latest gradients to older model parameter versions, while the optimizer is consistently updated with the latest GPU-computed values.

3 Motivation

Large-scale model training demands extraordinary computational resources, where checkpoint systems play a pivotal role in mitigating the cost of unplanned interruptions. However, existing checkpointing techniques face persistent challenges in GPU utilization, state transfer overhead, and adaptability to dynamic training scenarios. This section first models the time consumption of checkpoint systems to quantify inefficiencies, then analyzes limitations of current solutions, and finally demonstrates how our approach (GoCkpt) leverages mixed-precision training and relaxed consistency constraints to achieve near-zero overhead checkpointing.

3.1 Quantifying Inefficiencies in Checkpoint Systems

To systematically evaluate checkpoint overhead, we model the training process as a sequence of steps with periodic checkpointing. Let T_{total} denote the total effective training time (sum of all step times T_{step}), N the checkpoint interval (number of steps between saves), p the system failure rate (failures per second), T_{ckpt} the time to save a complete checkpoint, and T_{load} the time to restore from a checkpoint.

Training interruptions incur three types of wasted time:

1. Checkpoint save overhead: Over T_{total}, the total time spent saving checkpoints is $T_{\text{save}} = \frac{T_{\text{total}}}{N \cdot T_{\text{step}}} \cdot T_{\text{ckpt}}$ (since one checkpoint is saved every N steps).

2. Checkpoint restore induced idle time: Failures can occur uniformly between checkpoints, with an average gap of $\frac{NT_{\text{step}}}{2}$ (assuming exponential distribution for failure times and p is small, a common model for hardware/software errors [6]). The expected idle time before recovery is $\frac{1}{2}pNT_{\text{total}}T_{\text{step}}$ (by memoryless property of exponential distribution), where p^{-1} is the mean time between failures (MTBF).

3. Checkpoint load overhead: Each failure requires restoring the latest checkpoint, contributing $T_{\text{restore}} = pT_{\text{total}}T_{\text{load}}$ over T_{total}.

Total wasted time T_{waste} combines these components:

$$T_{\text{waste}} = T_{\text{save}} + \frac{1}{2}pNT_{\text{total}}T_{\text{step}} + pT_{\text{total}}T_{\text{load}}$$

(Note: We use p^{-1} instead of pT_{full} for clarity, aligning with standard reliability theory.)

The ratio of wasted time to effective training time, P quantifies checkpoint inefficiency:

$$P = \frac{T_{\text{waste}}}{T_{\text{total}}} = \frac{T_{\text{ckpt}}}{NT_{\text{step}}} + \frac{pNT_{\text{step}}}{2} + pT_{\text{load}}$$

A critical observation is that P is minimized when the derivative with respect to N is zero. Taking $\frac{dP}{dN} = -\frac{T_{\text{ckpt}}}{N^2T_{\text{step}}} + \frac{pT_{\text{step}}}{2} = 0$. we find the optimal checkpoint interval N^*:

$$N^* = \sqrt{\frac{2T_{\text{ckpt}}}{pT_{\text{step}}^2}}$$

So that we can get $P^* = \sqrt{2pT_{\text{ckpt}}} + pT_{\text{load}}$ is the minimized overhead of checkpoint system when $N = N^*$. The corresponding GPU utilization overhead can be calculated by $\frac{P^*}{P^*+1}$.

This confirms our initial intuition: the optimal checkpoint frequency balances save overhead against failure-induced losses. However, our analysis reveals a deeper insight: the fundamental bottleneck of checkpoint systems lies not in the interval N, but in the per-checkpoint overheads T_{ckpt} and T_{load}.

3.2 Limitations of Existing Checkpoint Schemes

Current checkpointing techniques struggle to minimize T_{ckpt} and T_{load} for large-scale models. We categorize their limitations as follows:

Parallelism-constrained schemes: Periodic checkpointers like CheckFreq [30] and VeloC [31] attempt to overlap checkpointing with training steps. However, due to strict consistency requirements (e.g., ensuring all GPUs finish the same step before saving), they can only parallelize within a single step's forward/backward pass, leaving most GPU cycles idle during checkpoint saves. Our experimental validation shows that existing optimizations still offer nearly 5% throughput improvement at the optimal checkpoint frequency. (section 5)

Storage-bottleneck solutions: Persistence-focused approaches such as PCCheck [37] and GPM [32] use byte-addressable persistent memory (PMEM) or block transfers to accelerate storage. However, these approaches fail to address the GPU-CPU bandwidth bottleneck: transferring checkpoints (e.g., a 16GB checkpoint for a billion-parameter model) over PCIe Gen3 (which can achieve a maximum bandwidth of approximately 12 GB/s) still takes over a second per checkpoint, and even using PMEM cannot reduce this time further.

3.3 Feasibility of GoCkpt

To overcome the aforementioned difficulty in reducing GPU interruption duration, a natural approach is to parallelize checkpoint transmission with more training steps, thereby hiding the checkpoint operation under more training computations. However, this presents a serious problem: the checkpoint state changes between training steps. Directly parallelizing checkpoint transmission with training will destroy the consistency of the checkpoint states, resulting in different checkpoint versions for each part and affecting checkpoint accuracy. The characteristics of mixed-precision training and CPU parameter update techniques in subsection 2.2 and subsection 2.3 inspire us to take advantage of the fact that the gradient space in mixed-precision training is much smaller than the model parameter and optimizer parameter space. In addition to the checkpoint data, we can also transmit a portion of the gradient data generated by backpropagation and use the CPU to update the checkpoint, ultimately developing a complete and consistent checkpoint version on the CPU.

4 Design and Implementation

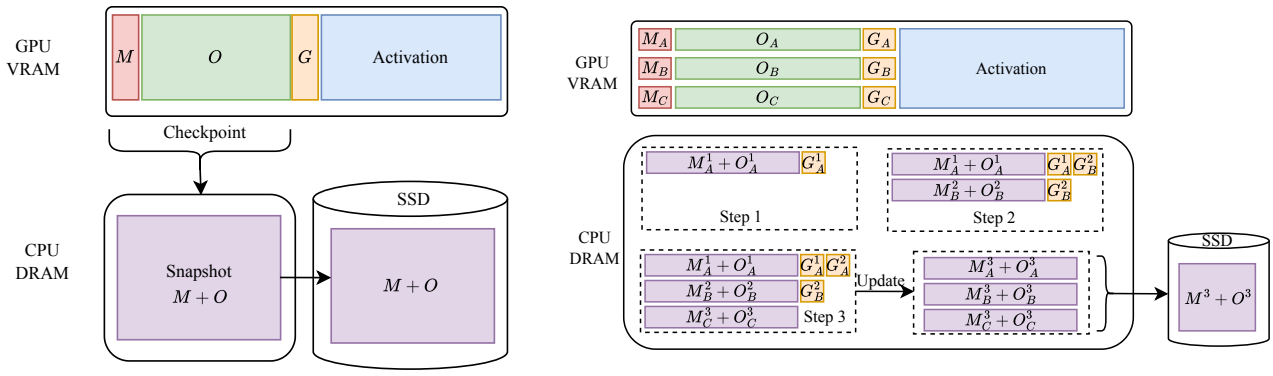
As discussed in section 3, existing checkpointing methods share a common limitation: they are unable to mitigate the training latency caused by transferring checkpoint data from the GPU to the CPU.

To minimize this latency, we propose GoCkpt. Our design aims to achieve the following goals:

- Overlap the checkpoint snapshot transfer process with multiple training steps.
- Reconstruct a consistent checkpoint state on the CPU.
- Optimize I/O to fully utilize the PCIe bandwidth between the GPU and CPU and avoid disrupting training.

4.1 Overview of GoCkpt

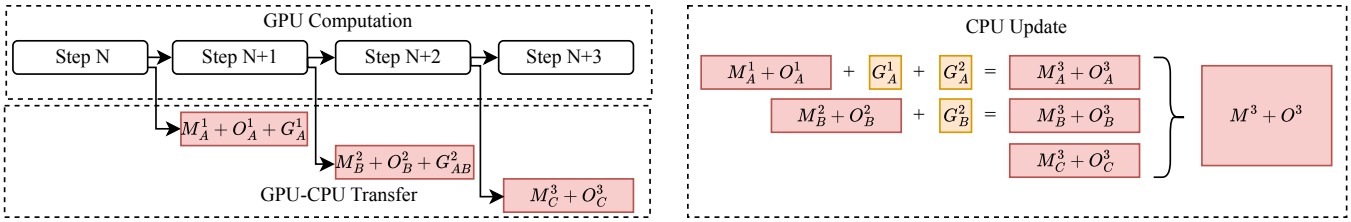
Figure 2 illustrates the overall framework of GoCkpt's design, highlighting the differences between existing single-step checkpoints and our proposed multi-step checkpoints. As Figure 2a shows, traditional checkpointing schemes typically consist of two phases: snapshot and persist. The snapshot phase transfers checkpoint data (primarily model and optimizer states) to the CPU, interrupting training. After the snapshot phase, a background thread is launched to perform the persist phase, persisting the complete checkpoint state on the CPU to a medium such as an SSD. In contrast, as Figure 2b, our GoCkpt approach consists of three phases: the checkpoint overlap transfer phase, the checkpoint CPU reconstruction phase, and the checkpoint persistence phase. The checkpoint overlap transfer phase also requires transferring checkpoint data to the CPU. Still, by splitting the entire checkpoint data into multiple parts, we can overlap each part with a training step and additionally transfer the low-precision gradient data from mixed-precision training



(a) Snapshot in single step

(b) Snapshot in multiple steps

Figure 2. Traditional single-step snapshot (a) and GoCkpt multi-step snapshot (b) overview



(a) Parallelize GPU-CPU transfer with GPU computation

(b) Update the checkpoint state to a consistent version by CPU

Figure 3. Compute transfer overlap and CPU-assisted updates

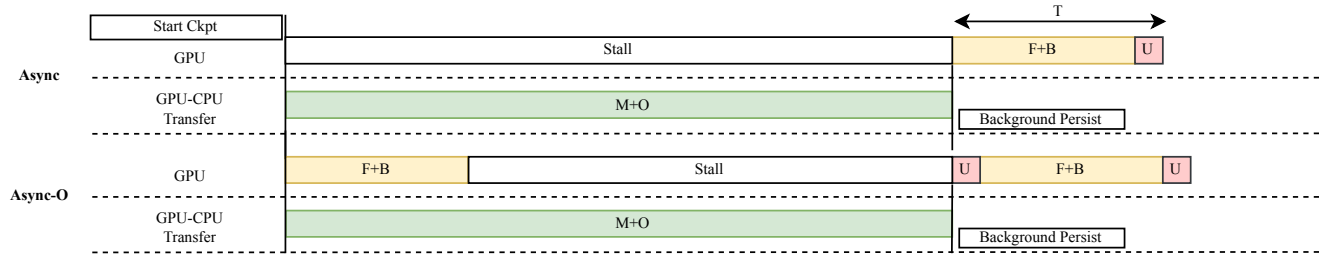
for subsequent phases. The checkpoint CPU reconstruction phase leverages the CPU data to update the checkpoint state to the latest consistent version through CPU computation. The final persistence phase is consistent with traditional checkpointing schemes.

4.2 Parallelize GPU-CPU transfer with GPU computation

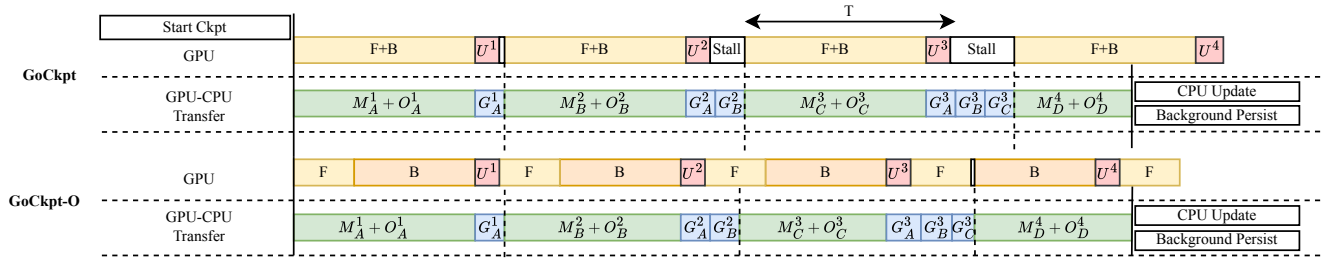
4.2.1 transfer contens. GoCkpt hides the visible interruption duration by overlapping the GPU-to-CPU checkpoint transfer with GPU computation. The diagram is shown in Figure 3a, where we begin the checkpoint operation after Step N. Instead of transferring all data at once, we split the entire checkpoint into multiple parts, transferring a portion at each step. In the diagram, the checkpoint state, including model and optimizer parameters, is divided into three parts: A, B, and C. These parts overlap with training in Step N+1, N+2, and N+3, with one part transferred in each step. Since each step in large model training updates the model and optimizer parameters based on the gradients generated in that step, the three parts of the checkpoints transferred to the CPU are the checkpoint states corresponding to Steps N+1 through N+3. In addition to the checkpoint state, we also need to transfer the gradients corresponding to the existing checkpoints to the CPU at each step (G_A^1 and G_{AB}^2 in the diagram). These gradients are retained only within the corresponding step and therefore need to be transferred to the corresponding step.

4.2.2 Transmission order and priority management. Because model parameters and optimizer parameters often reside in separate memory spaces, we organize them into blocks. This ensures that after each block of model parameters is transferred, the corresponding optimizer parameters are immediately transferred, achieving block-level alignment between the two. At the end of each step, we adaptively detect the data blocks that have been assigned and submit the transmission tasks for the gradients corresponding to these data blocks. We use a prioritized queue to manage transmission tasks, with gradient transmission tasks assigned a higher priority. When gradient transmission conflicts with model/optimizer state transmission tasks, gradient transmission takes precedence.

4.2.3 Checkpoint Stall Analysis. In this section, we analyze the duration of training interruptions caused by GoCkpt and compare it with traditional asynchronous checkpoints (Async) and the most advanced checkpoint transmission scheme that overlaps with single-step training (Async-O) to demonstrate the theoretical advantages of GoCkpt's transmission scheme.



(a) traditional async checkpoint schemes: Async and Async-O



(b) multi-step overlapped checkpoint schemes: GoCkpt and GoCkpt-O

Figure 4. Computation and data transfer flow of various checkpointing schemes. F-Forward, B-Backward, U-Update, T-Time of one single step, M+O-When GPU transfers Model and Optimizer parameters to CPU, G-When GPU transfers gradients to CPU

In the actual training of large models, single-step training times can vary slightly due to computational and communication delays. However, this variation is relatively small relative to the total training time and is consistent over time, meaning that the training times for consecutive steps are typically very close. Therefore, we can assume that the single-step training time is always the same and use the average single-step training time of the first few steps during the actual checkpointing process as a reference. First, we analyze the idea of the GoCkpt algorithm and conduct a detailed analysis of its time cost and the acceleration effect on checkpoints.

As shown in Figure 4a, the typical checkpoint schemes nowadays usually run in parallel with the first GPU-CPU transmission stage after the checkpoint saving process is initiated. The effectiveness of this method depends on the single-step duration: if the single-step duration exceeds the checkpoint transmission time, pauses become inevitable [30].

As shown in Figure 4b, our checkpointing system requires transferring the contents of large model checkpoints and the gradient information needed in sections while training steps continue. For the GoCkpt solution, we hide entirely the checkpoint transfer process within the training process, so the only visible overhead to the user is the gradient transfer. In mixed-precision training scenarios, only one-sixth of the gradients need to be transferred for each checkpoint part at each step.

Assuming we split the checkpoint transfer into N steps, we can calculate the checkpoint outage duration for Async-O and GoCkpt separately:

$$T_{Async-O} = (N-1)T_{step}$$
$$T_{GoCkpt} = \sum_{i=1}^{N-1} \frac{i}{7} T_{step} = \frac{N(N-1)}{14} T_{step}$$
$$\Delta T = T_{GoCkpt} - T_{Async-O} = \frac{-N^2+15N-14}{14} T_{step}$$

It can be seen from the formula that, in the optimal case, GoCkpt can overlap the entire checkpoint transmission process into 7 or 8 steps, and can bring a $4T_{step}$ reduction in checkpoint outage duration. Therefore, we propose an optimal checkpoint transmission strategy: the checkpoint process is overlapped into seven steps. Suppose the model and optimizer state transmission still cannot be completed after seven steps. In that case, the remaining checkpoints are transmitted by blocking, thereby avoiding an increase in gradient transmission overhead that would offset the optimization of the GoCkpt method.

4.2.4 Further hide the overhead of gradient transmission.

Based on the previous scheme, we propose GoCkpt-O, a more optimized scheme to reduce the overhead of gradient transmission further. It is based on some existing implementations—such as DeepSpeed's BF16Optimizer and the Zero-Stage3 implementation that retain gradients from the forward pass of the current step for use in the forward pass of the next step. They do not overwrite the original gradient space with newly computed gradients until the next back-propagation step. This behavior allows us to overlap the transmission of model gradients with the forward propagation of the next step, using gradients that remain unchanged during the parameter update phase of the current step.

By leveraging this mechanism, we can further reduce the communication overhead associated with gradient transfer in the overall training process.

The data transfer flow can be visualized as Figure 4b GoCkpt-O: when the gradient transmission time does not exceed the combined duration of the parameter update and the forward propagation of the next step, the gradient transfer can be effectively overlapped into the forward pass, thereby reducing perceived latency.

4.3 Update the checkpoint state to a consistent version by CPU

4.3.1 Parameter Update Module. After transferring the model parameters, optimizer states, and gradients, we update parameters on the CPU using the AdamW optimization strategy. As illustrated in Figure 3b, consider overlapping checkpoint transfers across three consecutive steps: For checkpoint version 1 of part A transferred at Step N+1, gradients of part A computed in Steps N+1 and N+2 are used to update checkpoint version 3. Similarly, checkpoint version 2 of part B transferred at Step N+2 is updated with part B's gradients computed at Step N+2, also yielding checkpoint version 3. Once these updates are complete, the CPU retains the full checkpoint version 3—equivalent to directly transferring the checkpoint from Step N+3 to CPU memory.

During the parameter update process, we also use a multi-threading mechanism to update the parameters in parallel, thereby minimizing the overhead of parameter updates. In fact, from the experiments in section 5, we can see that the time to update the parameters is relatively short with an appropriate number of threads, much shorter than the interval between checkpoint saves.

4.3.2 Checkpoint loading. Since optimizing checkpoint loading and optimizing checkpoint interruption have orthogonal effects on GPU utilization, we use the same synchronization strategy for loading checkpoints as traditional checkpointing schemes. When loading a checkpoint, it is first read from the SSD into CPU memory and then transferred to GPU memory. Once the model and optimizer parameters are ready on the GPU, training is resumed at the step after the checkpoint is consistent.

4.4 IO Bandwidth Optimization

4.4.1 Multi-threaded Design. GoCkpt introduces multi-threaded design optimizations in both the snapshot and persistence phases of data transfer. During the multi-step snapshot phase, background threads manage independent CUDA streams and thread synchronization to manage GPU-CPU checkpoint transfers, minimizing disruption to training. During the persistence phase, multiple threads concurrently write to the SSD to maximize NVMe SSD bandwidth utilization.

4.4.2 GPU-CPU PCIe Transfer Optimization. To optimize data transfer between the GPU and CPU, we pre-register the CPU memory used as Pinned Memory using the PyTorch interface. This technique locks the memory pages to be used, avoiding swapping in and out and improving data transfer efficiency. We also split the model and optimizer parameters into 4MB chunks, transferring the corresponding model and optimizer parameter chunks at a time. This maximizes PCIe bandwidth while ensuring traceability of transfer status.

4.4.3 Data persistence module. After the parameter update is complete, the data persistence module uses multiple threads in parallel to save the model parameters to disk in the background, fully utilizing the SSD's write bandwidth. After the model tensors are persisted, a callback function is used to save the pre-prepared checkpoint metadata, such as the current step count, historical throughput, and historical training time. Saving this metadata marks the completion of the latest checkpoint, allowing it to be used for recovery. If the previous checkpoint metadata has not yet been written to disk when the next checkpoint is saved, GoCkpt will wait for the last checkpoint to complete before starting the new checkpoint process.

4.5 Multi-card Environment Design

In a multi-GPU environment, training frameworks like DeepSpeed launch a training process per GPU, so the checkpointing framework only needs to handle inter-GPU synchronization (with Rank 0 monitoring completion by other Ranks and coordinating synchronization).

For checkpointing strategies in multi-GPU setups, we define rules based on actual parallelism (tensor parallelism, pipeline parallelism). Each GPU stores only its shards involved in parallel training, focusing on data parallelism efforts.

Under data parallelism, we determine how data parallel group members collectively save exact model copies. With ZeRO-1/2 parallelism, each GPU in the data parallel group saves its optimizer state shard; during loading, these shards are fetched via intra-group communication, avoiding redundant duplication [34].

When the model is trainable, data parallelism reduces per-GPU memory pressure. Thus, our approach combines tensor parallelism (TP) on individual GPUs with ZeRO-1 data parallelism across multi-GPU servers to distribute workload.

4.6 GoCkpt Implementation

We implemented our GoCkpt and GoCkpt-O solutions in Python and C++, totaling approximately 2,000 lines of code, and integrated them into the Deepspeed training framework. We used Pybind11 to provide easy-to-use interfaces, including three API interfaces: save_checkpoint, backward_begin, and update_begin, which are inserted before the checkpoint

save, backpropagation, and parameter update phases of each step. We used PyTorch for CPU-bound memory allocation and management, C++ for multithreading, and independent CUDA stream management for fine-grained, high-performance data replication. Finally, we encapsulated Deepspeed's native CPU AdamW updater to implement parameter updates on the CPU, ensuring consistency and accuracy with those on the GPU.

5 Evaluation

5.1 Setup

The experimental hardware environment was configured with two distinct setups: a single-GPU platform and a multi-GPU server cluster. For the single-GPU configuration, we deployed a system equipped with a Tesla V100S GPU, paired with dual 48-core CPUs (96 cores total), 128 GB of DDR4 RAM, and 3.8 TB of high-speed NVMe SSD storage dedicated to data I/O. In the multi-GPU setup, an Alibaba Cloud server was utilized, featuring eight H100-80GB GPUs interconnected via NVLink for low-latency communication. Each GPU was assigned a dedicated network interface card (NIC), and the GPUs were logically grouped into NUMA nodes with four GPUs per node. The server was powered by a 224-core Intel Platinum 8480C CPU, supported by 2 TB of ECC RAM (composed of 32 × 64 GB modules) and four 3.5 TB NVMe SSDs (aggregating to 14 TB of raw storage capacity). To mitigate potential SSD bandwidth bottlenecks during checkpointing, we implemented targeted GPU optimizations: four GPUs were strategically distributed across 2 NUMA nodes, with each process allocated 28 CPU cores to enforce strong CPU-GPU affinity within the same NUMA domain. Additionally, each GPU independently saved checkpoint data to its associated NVMe SSD, enabling parallel I/O operations and reducing storage access latency.

5.2 Benchmark and Baseline

The experimental evaluation focuses on the Wikidata dataset, primarily quantifying model throughput (samples processed per second). Following large-scale model training standards (per LLaMA documentation), raw data was preprocessed via fixed-sequence padding to ensure uniform input dimensions across runs, minimizing computational/memory load variability for consistent checkpointing scheme comparisons.

Diverse models—LLaMA3.1-1B, Qwen3-0.6B, OPT-350M were selected to assess performance across computational scales to smaller configurations. LLaMA3-8B in multi-GPU settings further enabled exploration of scaling behavior, particularly critical distributed communication/synchronization overheads.

Three checkpointing categories were evaluated: synchronous schemes (e.g., Deepspeed, DCP) pausing training to serialize model states (parameters, optimizers, gradients) for strong consistency/recoverability but significant interruption latency; asynchronous schemes (e.g., Torch-Snapshot, DCP-Async) offloading persisting to background processes for concurrent training; and single-step overlapping schemes (DLRover-Flash, Datastates-LLM) interleaving checkpointing with single-step forward/backward computation to reduce overhead.

We replicated two GPU-CPU transfer-optimized schemes: Async (asynchronous with I/O optimizations) and Async-O (combining transfers with single-step overlap). Proposed schemes—the GoCkpt (explicit gradient transfer waits) and the GoCkpt-O (implicit gradient transfer)—were also evaluated, alongside an ideal zero-overhead scenario (no storage delays/computation interruptions) as a theoretical upper bound.

All experiments used consistent batch sizes to isolate checkpointing impacts: one sample/device for single-GPU runs and four samples/device for multi-GPU setups. This scaling ensured full parallel resource utilization while maintaining cross-configuration comparability, enabling evaluation of each scheme under realistic training conditions.

5.3 Performance Evaluation

In this part, we wish to investigate the role of the GoCkpt checkpointing system in reducing checkpoint downtime and improving the training throughput of large models.

We show the throughput effect of three different models with two different frequencies of checkpoint intervals, once every 50 steps and once every 200 steps.

In Figure 5, we can see that under different models and checkpoint frequencies, the gap between GoCkpt and Ideal is between 0.2% and 1.8%, and the gap between GoCkpt-O and Ideal is less than 1.2%. Compared with traditional asynchronous checkpointing schemes, GoCkpt and GoCkpt-O achieve 6.8% to 38.4% and 6.7% to 40.1% performance improvement, respectively. GoCkpt and GoCkpt-O achieve at most 1.7% to 3.0% and 2.9% to 4.3% performance improvement, respectively, compared with the two replication schemes Async and Async-O with the same data transfer optimization.

Figure 6 shows the comparison of checkpointing time cost between several checkpointing schemes with IO optimization. In the test of LLaMA3.2-1B model, the actual checkpointing time caused by GoCkpt and GoCkpt-O schemes is only 17.7% to 31.0% and 0.5% to 10.0% of asynchronous checkpointing. Compared with the Async-O scheme, GoCkpt and GoCkpt-O schemes can reduce the checkpointing time by 57.7% to 70.1% and 86.4% to 99.2%, respectively. For relatively small models such as Qwen3-0.6B and OPT-350M, the GoCkpt-O schemes reduce the time consumption of the checkpointing scheme to 0.003 to 0.004 s, resulting in nearly zero overhead.

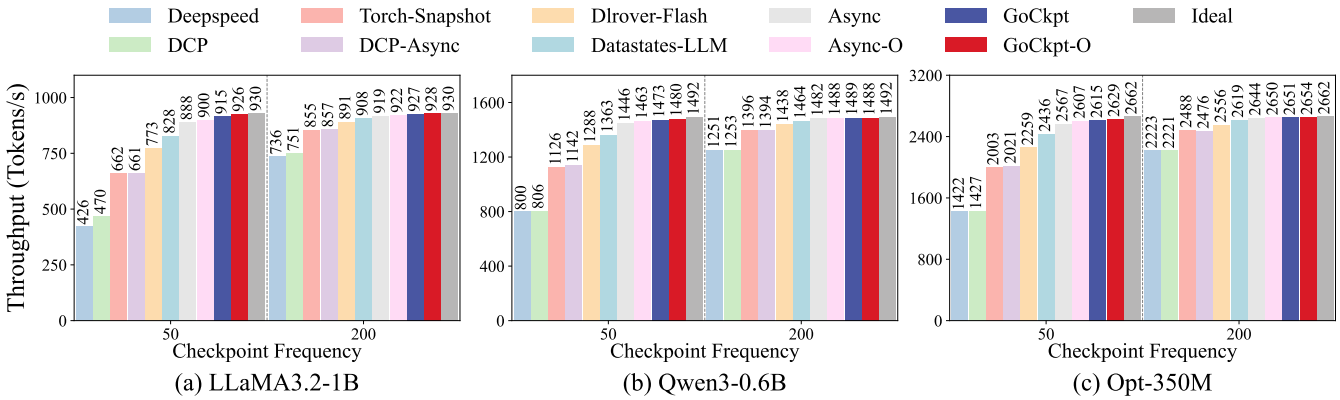


Figure 5. Checkpoint throughput and stall time for different checkpoint frequency settings (higher is better)

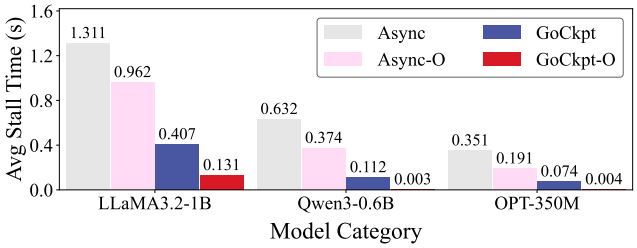


Figure 6. Average stall time of different checkpointing schemes (lower is better)

Scheme	Max T_{ckpt}(s)	N_{best}	Throuput(Tokens/s)
Deepspeed	36.79	472	411.9
DCP-Async	12.226	272	697.8
Async	1.313	89	758.0
Async-O	0.988	77	776.3
GoCkpt	0.435	51	786.4
GoCkpt-O	0.175	32	794.1

Table 1. The performance of different solutions on the crash expression task, using Llama3.2-1B model, N_{best} is calculated by formula $\sqrt{\frac{2T_{ckpt}}{pT_{step}^2}}$, the system crashes every 600s

5.4 Checkpoint & Restore Expriments

We simulate the process of crashing the checkpoint system and reloading the latest checkpoint under the simulated failure frequency (crash per 600s), which is used to verify the complete overhead change caused by the checkpoint system.

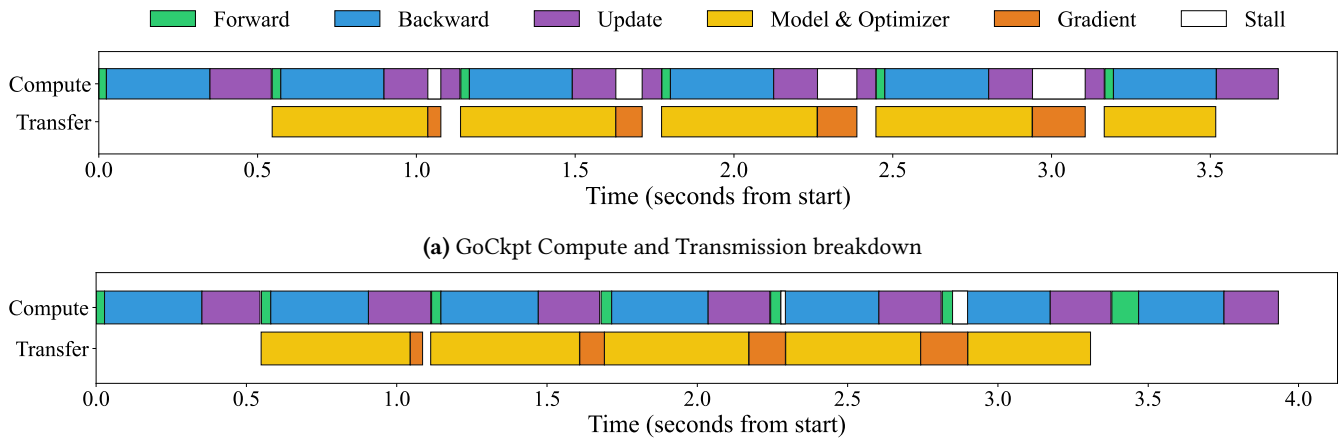
For the traditional synchronous checkpoint, asynchronous checkpoint, Async, Async-O, and our schemes GoCkpt and GoCkpt-O, we determined the optimal checkpoint frequency according to their checkpoint interruption time, respectively, as shown in Table 1. For each scheme, we picked the checkpoint frequency that is very close to the optimal frequency.

At this frequency, the overhead of checkpoint saving and the total overhead of checkpoint recovery of each kind of checkpoint are close to the lowest in theory. In this case, we verify the optimal throughput of various checkpoint schemes through experiments. From the experimental results, we can see that GoCkpt-O and GoCkpt schemes can effectively improve the optimal checkpoint frequency and obtain higher throughput because they can reduce the time of checkpoint transmission itself. Compared with the existing schemes, GoCkpt-O achieves 192.8% and 113.8% compared with the synchronous checkpointing scheme Deepspeed and the asynchronous checkpointing scheme torch DCP Async, respectively. Even compared with the transport optimized Async and Async-O schemes, GoCkpt-O can obtain 2.3%-4.8% throughput improvement respectively.

5.5 Breakdown analysis of GoCkpt system

In this section, we conduct an experimental analysis of the actual performance of the GoCkpt system under the setting of LLaMA3.2-1B model. We analyze the breakdown of the computation and transmission process during the checkpoint saving process in Figure 7a and Figure 7b.

As shown in Figure 7a, we overlap the entire checkpoint transfer process into multiple training steps, and we can see that the actual stall process occurs during the gradient transfer. The figure illustrates the time spent on gradient computation, parameter update, and checkpoint transmission during the checkpoint saving process. The x-axis represents the time in seconds, while the y-axis shows the different phases of the process. The GoCkpt scheme stops training in the process of transferring gradients. Although the training is still paused, the gradient size in the mixed-precision training is much smaller than the size of the model checkpoint, so the majority of the training time can be overlapped with the checkpoint transmission time. In the figure, we can also observe that the update phase is divided into two parts: the part that does not interrupt the checkpoint transmission



(a) GoCkpt Compute and Transmission breakdown

(b) GoCkpt-O Compute and Transmission breakdown

Figure 7. Breakdown analysis of GoCkpt and GoCkpt-O

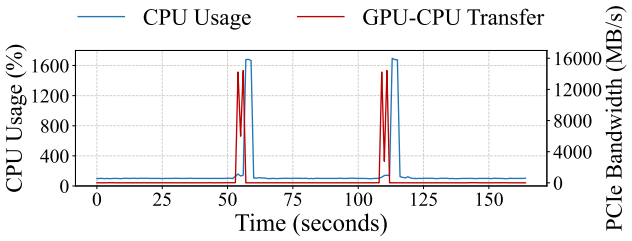


Figure 8. GoCkpt CPU Usage (Left y-axis) and GPU-CPU transfer bandwidth (Right y-axis) curve

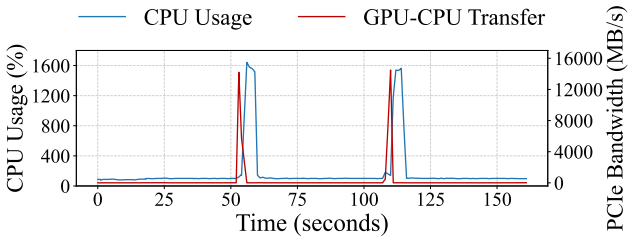


Figure 9. GoCkpt-O CPU Usage (Left y-axis) and GPU-CPU transfer bandwidth (Right y-axis) curve

and the part that does. We analyze that this part of the time without interrupting the checkpoint transmission is an asynchronous operation that is not entirely completed during the backpropagation phase. After the gradient transfer, the parameters are updated on the GPU.

Similarly, we conduct breakdown analysis for the GoCkpt-O scheme in Figure 7b. In this scheme, we assume that the gradient is retained until the end of the forward propagation process, to make full use of the parameter update phase and the overhead of the forward propagation phase to hide the gradient transmission. It can be seen in the real experiments that by applying this optimization, the overhead of gradient transfer is completely hidden in the first few training steps.

5.6 System Indicator Monitoring

As can be seen from Figure 8 and Figure 9, each checkpoint save operation results in a period of increased GPU-CPU transfer bandwidth and a brief increase in CPU utilization. The actual peak CPU utilization depends on the number of background threads, which is set to 16 in our implementation.

For both GoCkpt and GoCkpt-O, we schedule parameter updates and background persistence after the GPU-CPU transfer. As can be seen in the figure, each GPU-CPU transfer peak is followed by a CPU utilization peak. For both schemes, GoCkpt waits and stalls at each step of the data transfer, reducing the overall bandwidth utilization during the checkpoint transfer. This is reflected in the double peaks in each GPU-CPU transfer in the figure. This means that the maximum bandwidth cannot be achieved throughout the transfer. In GoCkpt-O, by introducing hidden and overlapping gradient transfers, a more balanced single-GPU-CPU transfer is achieved; that is, the data can be viewed as a continuous transfer from the GPU to the CPU. After the transmission is completed, the CPU performs background parameter updates and persistence work to obtain complete checkpoint data from the data transmitted in multiple steps.

5.7 Scalability Verification Experiments

We conducted experiments on a multi-card checkpointing system (4xH100). We found that while multi-card communication occurs via NVLink, the GPU-CPU transmission path in our configuration is undertaken via a PCIe switch, with

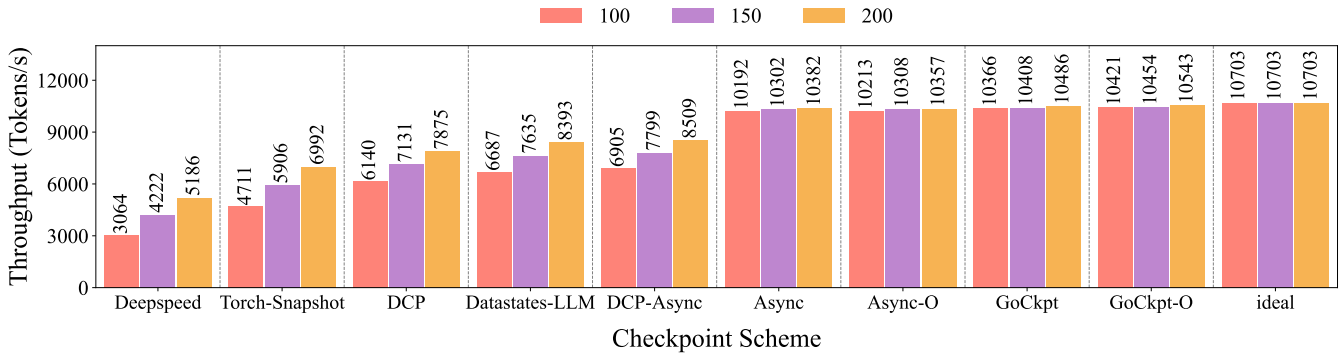


Figure 10. Throughput of LLaMA3-8B with varying checkpoint frequencies (higher is better)

each card connected to a separate PCIe root domain. This means each card is connected to a separate PCIe switch entity and has its own GPU-CPU transmission path, which does not conflict with the others.

As shown in Figure 10, experiments show that when training the LLaMA8B model with four cards, GoCkpt and GoCkpt-O achieve the highest training throughput, trailing Ideal by only 2.0% to 3.1% and 1.5% to 2.6%, respectively. This represents a 23.2% to 50.9% improvement over the asynchronous checkpointing solution DCP-Async, and 1.0% to 2.2% throughput improvement over our replicated Async solution.

6 Related Work

Periodic checkpointing As a cornerstone technology for large-scale model training, periodic checkpointing has undergone substantial optimization in four core dimensions: *frequency*, *Snapshot*, *Persistence*, and *Loading*. For frequency optimization, studies derive optimal intervals through mathematical modeling (e.g., the Young/Dali formula [6]) or graph-based fault correlation analysis [13], while JIT [15] and CPR [27] further align with these analyses. Regarding the Snapshot stage, early methods (e.g., CheckFreq [30] and VeloC [31]) introduced asynchronous schemes to overlap snapshots and training, while later methods (e.g., DataStates-LLM [28] and ByteCheckpoint [40]) focused on utilizing the GPU-CPU PCIe bandwidth. Innovations in the persistence phase aim to minimize storage latency: PCCheck [37] overlaps GPU-CPU data transfers with SSD persistence through block-based pipelining; GEMINI [41] stores state replicas in network nodes to bypass slow node persistence; GPM [32] explores the feasibility of PMEM as an alternative persistence medium. During the loading phase, ServerlessLLM [12] reduces recovery time through in-memory preloading, although this approach relies on error-free local checkpoints, which is a challenge in large-scale training. It is worth noting that some periodic methods relax consistency requirements during persistence (e.g., CPR [27]), but are still limited to specific workloads (e.g., recommendation models) and suffer from convergence risks in large language models [33].

Aperiodic Checkpointing JIT [15] proposes a fault-triggered checkpointing to avoid periodic overhead, yet still incurs GPU-CPU transfer and persistence costs, making it unsuitable for frequently preempted instances. Swift [48] reconstructs states via pipeline communication logs. FlowCheck[21] maintains CPU parameters during training but faces state inconsistency over time, requiring periodic validation. What's more, Phoenixos [43] integrates checkpointing at the OS level but lacks application-specific optimizations for large models.

Checkpoint Compression Techniques aim to reduce checkpoint volume: Check-N-Run[10], SSDC [42] saves incremental parameters for recommendation models; ExCP [26] uses compression/pruning; Inshrinkerator[1] supports lossy or lossless quantization-aware differential compression; and APR [2] omits recomputable data;Delta-DNN [20], SCAR [33], LC-Checkpoint [7], QD-Checkpoint [24], explore lossy compression based on state-difference.

Fault Tolerance via Replication Some works, such as Varuna [4], Oobleck [22], and Bamboo [38] improve robustness using data parallel replicas, where failed node states are reconstructed from peers. This complements (but is orthogonal to) checkpointing-based strategies.

7 Conclusion

GoCkpt is a method for training large language models (LLMs) that minimizes training interruptions caused by the checkpointing system by overlapping checkpointing with multiple single-step training processes. For large model training, the bottleneck of checkpointing is that the single-step checkpointing interruption time cannot be further reduced, which limits the frequency of checkpointing and leads to significant GPU utilization overhead. Existing checkpointing schemes inevitably incur the overhead of transferring model and optimizer state from the GPU to the CPU. GoCkpt addresses this bottleneck by allowing checkpointing to be

overlapped across multiple training steps and restoring a consistent checkpoint version on the CPU. In our experiments, our checkpointing system improves training throughput by 38.4% compared to traditional asynchronous checkpointing schemes. We also show that GoCkpt can reduce training interruption time by 86.7% while improving throughput by 4.8% compared to state-of-the-art checkpointing schemes.

References

[1] Amey Agrawal, Sameer Reddy, Satwik Bhattamishra, Venkata Prabhakara Sarath Nookala, Vidushi Vashishth, Kexin Rong, and Alexey Tumanov. 2024. Inshrinkerator: Compressing Deep Learning Training Checkpoints via Dynamic Quantization. In *Proceedings of the 2024 ACM Symposium on Cloud Computing* (Redmond, WA, USA) *(SoCC '24)*. Association for Computing Machinery, New York, NY, USA, 1012–1031. https://doi.org/10.1145/3698038.3698553

[2] Ismail Akturk and Ulya R. Karpuzcu. 2020. ACR: Amnesic Checkpointing and Recovery. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 30–43. https://doi.org/10.1109/HPCA47549.2020.00013

[3] Joel André, Foteini Strati, and Ana Klimovic. 2022. Exploring learning rate scaling rules for distributed ML training on transient resources. In *Proceedings of the 3rd International Workshop on Distributed Machine Learning* (Rome, Italy) *(DistributedML '22)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3565010.3569067

[4] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. 2022. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 472–487. https://doi.org/10.1145/3492321.3519584

[5] Jehyeon Bang, Yujeong Choi, Myeongwoo Kim, Yongdeok Kim, and Minsoo Rhu. 2024. vTrain: A Simulation Framework for Evaluating Cost-Effective and Compute-Optimal Large Language Model Training. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 153–167. https://doi.org/10.1109/MICRO61859.2024.00021

[6] Anne Benoit, Yishu Du, Thomas Herault, Loris Marchal, Guillaume Pallez, Lucas Perotin, Yves Robert, Hongyang Sun, and Frederic Vivien. 2022. Checkpointing à la Young/Daly: An Overview. In *Proceedings of the 2022 Fourteenth International Conference on Contemporary Computing* (Noida, India) *(IC3-2022)*. Association for Computing Machinery, New York, NY, USA, 701–710. https://doi.org/10.1145/3549206.3549328

[7] Yu Chen, Zhenming Liu, Bin Ren, and Xin Jin. 2020. On efficient constructions of checkpoints. In *Proceedings of the 37th International Conference on Machine Learning (ICML'20)*. JMLR.org, Article 152, 10 pages.

[8] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2023. PaLM: Scaling Language Modeling with Pathways. *J. Mach. Learn. Res.* 24 (2023), 240:1–240:113. https://jmlr.org/papers/v24/22-1144.html

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. 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, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/V1/N19-1423

[10] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. 2022. Check-N-Run: a Checkpointing System for Training Deep Learning Recommendation Models. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, Amar Phanishayee and Vyas Sekar (Eds.). USENIX Association, 929–943. https://www.usenix.org/conference/nsdi22/presentation/eisenman

[11] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. 2021. DAPPLE: a pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 431–445. https://doi.org/10.1145/3437801.3441593

[12] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Low-Latency Serverless Inference for Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 135–153. https://www.usenix.org/conference/osdi24/presentation/fu

[13] Masoud Gholami Estahbanati and Florian Schintke. 2019. Multilevel Checkpoint/Restart for Large Computational Jobs on Distributed Computing Resources. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. 143–14309. https://doi.org/10.1109/SRDS47363.2019.00025

[14] Ziyi Guan, Hantao Huang, Yupeng Su, Hong Huang, Ngai Wong, and Hao Yu. 2024. APTQ: Attention-aware Post-Training Mixed-Precision Quantization for Large Language Models. In *Proceedings of the 61st ACM/IEEE Design Automation Conference, DAC 2024, San Francisco, CA, USA, June 23-27, 2024*, Vivek De (Ed.). ACM, 107:1–107:6. https://doi.org/10.1145/3649329.3658498

[15] Tanmaey Gupta, Sanjeev Krishnan, Rituraj Kumar, Abhishek Vijeev, Bhargav Gulavani, Nipun Kwatra, Ramachandran Ramjee, and Muthian Sivathanu. 2024. Just-In-Time Checkpointing: Low Cost Error Recovery from Deep Learning Training Failures. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 1110–1125. https://doi.org/10.1145/3627703.3650085

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 770–778. https://doi.org/10.1109/CVPR.2016.90

[17] Tao He, Xue Li, Zhibin Wang, Kun Qian, Jingbo Xu, Wenyuan Yu, and Jingren Zhou. 2024. Unicron: Economizing Self-Healing LLM Training at Scale. *CoRR* abs/2401.00134 (2024). https://doi.org/10.48550/ARXIV.2401.00134 arXiv:2401.00134

[18] Samuel Hsia, Udit Gupta, Mark Wilkening, Carole-Jean Wu, Gu-Yeon Wei, and David Brooks. 2020. Cross-Stack Workload Characterization of Deep Recommendation Systems. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*. 157–168. https://doi.org/10.1109/IISWC50251.2020.00024

[19] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, Yonggang Wen, and Tianwei Zhang. 2024. Characterization of Large Language Model Development in the Datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, April 15-17, 2024*, Laurent Vanbever and Irene Zhang (Eds.). USENIX Association, 709–729. https://www.usenix.org/conference/nsdi24/presentation/hu

[20] Zhenbo Hu, Xiangyu Zou, Wen Xia, Sian Jin, Dingwen Tao, Yang Liu, Weizhe Zhang, and Zheng Zhang. 2020. Delta-DNN: Efficiently Compressing Deep Neural Networks via Exploiting Floats Similarity. In *Proceedings of the 49th International Conference on Parallel Processing* (Edmonton, AB, Canada) *(ICPP '20)*. Association for Computing Machinery, New York, NY, USA, Article 40, 12 pages. https://doi.org/10.1145/3404397.3404408

[21] Zimeng Huang, Hao Nie, Haonan Jia, Bo Jiang, Junchen Guo, Jianyuan Lu, Rong Wen, Biao Lyu, Shunmin Zhu, and Xinbing Wang. 2025. FlowCheck: Decoupling Checkpointing and Training of Large-Scale Models. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 1334–1349. https://doi.org/10.1145/3689031.3696088

[22] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. 2023. Oobleck: Resilient Distributed Training of Large Models Using Pipeline Templates. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 382–395. https://doi.org/10.1145/3600006.3613152

[23] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 947–960. https://www.usenix.org/conference/atc19/presentation/jeon

[24] Haoyu Jin, Donglei Wu, Shuyu Zhang, Xiangyu Zou, Sian Jin, Dingwen Tao, Qing Liao, and Wen Xia. 2023. Design of a Quantization-Based DNN Delta Compression Framework for Model Snapshots and Federated Learning. *IEEE Transactions on Parallel and Distributed Systems* 34, 3 (2023), 923–937. https://doi.org/10.1109/TPDS.2022.3230840

[25] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proc. VLDB Endow.* 13, 12 (2020), 3005–3018. https://doi.org/10.14778/3415478.3415530

[26] Wenshuo Li, Xinghao Chen, Han Shu, Yehui Tang, and Yunhe Wang. 2024. ExCP: Extreme LLM Checkpoint Compression via Weight-Momentum Joint Shrinking. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net. https://openreview.net/forum?id=hlvKd7Vdxm

[27] Kiwan Maeng, Shivam Bharuka, Isabel Gao, Mark C. Jeffrey, Vikram Saraph, Bor-Yiing Su, Caroline Trippel, Jiyan Yang, Mike Rabbat, Brandon Lucia, and Carole-Jean Wu. 2020. CPR: Understanding and Improving Failure Tolerant Training for Deep Learning Recommendation with Partial Recovery. *CoRR* abs/2011.02999 (2020). arXiv:2011.02999 https://arxiv.org/abs/2011.02999

[28] Avinash Maurya, Robert Underwood, M. Mustafa Rafique, Franck Cappello, and Bogdan Nicolae. 2024. DataStates-LLM: Lazy Asynchronous Checkpointing for Large Language Models. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing* (Pisa, Italy) *(HPDC '24)*. Association for Computing Machinery, New York, NY, USA, 227–239. https://doi.org/10.1145/3625549.3658685

[29] Avinash Maurya, Jie Ye, M. Mustafa Rafique, Franck Cappello, and Bogdan Nicolae. 2024. Deep Optimizer States: Towards Scalable Training of Transformer Models using Interleaved Offloading. In *Proceedings of the 25th International Middleware Conference* (Hong Kong, Hong Kong) *(Middleware '24)*. Association for Computing Machinery, New York, NY, USA, 404–416. https://doi.org/10.1145/3652892.3700781

[30] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, Marcos K. Aguilera and Gala Yadgar (Eds.). USENIX Association, 203–216. https://www.usenix.org/conference/fast21/presentation/mohan

[31] Bogdan Nicolae, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, and Franck Cappello. 2019. VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 911–920. https://doi.org/10.1109/IPDPS.2019.00099

[32] Shweta Pandey, Aditya K Kamath, and Arkaprava Basu. 2022. GPM: leveraging persistent memory from a GPU. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 142–156. https://doi.org/10.1145/3503222.3507758

[33] Aurick Qiao, Bryon Aragam, Bingjing Zhang, and Eric P. Xing. 2019. Fault Tolerance in Iterative-Convergent Machine Learning. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 5220–5230. http://proceedings.mlr.press/v97/qiao19a.html

[34] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory optimizations Toward Training Trillion Parameter Models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–16. https://doi.org/10.1109/SC41405.2020.00024

[35] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilic, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, Jonathan Tow, Alexander M. Rush, Stella Biderman, Albert Webson, Pawan Sasanka Ammanamanchi, Thomas Wang, Benoît Sagot, Niklas Muennighoff, Albert Villanova del Moral, Olatunji Ruwase, Rachel Bawden, Stas Bekman, Angelina McMillan-Major, Iz Beltagy, Huu Nguyen, Lucile Saulnier, Samson Tan, Pedro Ortiz Suarez, Victor Sanh, Hugo Laurençon, Yacine Jernite, Julien Launay, Margaret Mitchell, Colin Raffel, Aaron Gokaslan, Adi Simhi, Aitor Soroa, Alham Fikri Aji, Amit Alfassy, Anna Rogers, Ariel Kreisberg Nitzav, Canwen Xu, Chenghao Mou, Chris Emezue, Christopher Klamm, Colin Leong, Daniel van Strien, David Ifeoluwa Adelani, and et al. 2022. BLOOM: A 176B-Parameter Open-Access Multilingual Language Model. *CoRR* abs/2211.05100 (2022). https://doi.org/10.48550/ARXIV.2211.05100 arXiv:2211.05100

[36] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *CoRR* abs/1909.08053 (2019). arXiv:1909.08053 http://arxiv.org/abs/1909.08053

[37] Foteini Strati, Michal Friedman, and Ana Klimovic. 2025. PCcheck: Persistent Concurrent Checkpointing for ML. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 811–827. https://doi.org/10.1145/3669940.3707255

[38] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. 2023. Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, Mahesh Balakrishnan and Manya Ghobadi (Eds.). USENIX Association, 497–513. https://www.usenix.org/conference/nsdi23/presentation/thorpe

[39] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023). https://doi.org/10.48550/ARXIV.2307.09288 arXiv:2307.09288

[40] Borui Wan, Mingji Han, Yiyao Sheng, Yanghua Peng, Haibin Lin, Mofan Zhang, Zhichao Lai, Menghan Yu, Junda Zhang, Zuquan Song, Xin Liu, and Chuan Wu. 2025. ByteCheckpoint: A Unified Checkpointing System for Large Foundation Model Development. In *22nd USENIX Symposium on Networked Systems Design and Implementation, NSDI 2025, Philadelphia, PA, USA, April 28-30, 2025*, Theophilus A. Benson and Radhika Niranjan Mysore (Eds.). USENIX Association, 559–578. https://www.usenix.org/conference/nsdi25/presentation/wan-borui

[41] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. 2023. GEMINI: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 364–381. https://doi.org/10.1145/3600006.3613145

[42] Lingrui Xiang, Xiaofen Lu, Rui Zhang, and Zheng Hu. 2024. SSDC: A Scalable Sparse Differential Checkpoint for Large-scale Deep Recommendation Models. In *IEEE International Symposium on Circuits and Systems, ISCAS 2024, Singapore, May 19-22, 2024*. IEEE, 1–5. https://doi.org/10.1109/ISCAS58744.2024.10557880

[43] Tianle Sun Yingyi Hao Rong Chen Mingcong Han Jinyu Gu Haibo Chen Xingda Wei, Zhuobin Huang. 2025. PhoenixOS: Concurrent OS-level GPU Checkpoint and Restore with Validated Speculation. In *Proceedings of the ACM SIGOPS 31th Symposium on Operating Systems Principles*.

[44] Jinchen Xu, Guanghui Song, Bei Zhou, Fei Li, Jiangwei Hao, and Jie Zhao. 2024. A Holistic Approach to Automatic Mixed-Precision Code Generation and Tuning for Affine Programs. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2024, Edinburgh, United Kingdom, March 2-6, 2024*, Michel Steuwer, I-Ting Angelina Lee, and Milind Chabbi (Eds.). ACM, 55–67. https://doi.org/10.1145/3627535.3638484

[45] Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, Weng Lam Tam, Zixuan Ma, Yufei Xue, Jidong Zhai, Wenguang Chen, Zhiyuan Liu, Peng Zhang, Yuxiao Dong, and Jie Tang. 2023. GLM-130B: An Open Bilingual Pre-trained Model. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/forum?id=-Aw0rrrPUF

[46] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona T. Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: Open Pre-trained Transformer Language Models. *CoRR* abs/2205.01068 (2022). https://doi.org/10.48550/ARXIV.2205.01068 arXiv:2205.01068

[47] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. 2023. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proc. VLDB Endow.* 16, 12 (2023), 3848–3860. https://doi.org/10.14778/3611540.3611569

[48] Yuchen Zhong, Guangming Sheng, Juncheng Liu, Jinhui Yuan, and Chuan Wu. 2024. Swift: Expedited Failure Recovery for Large-Scale DNN Training. *IEEE Trans. Parallel Distrib. Syst.* 35, 9 (Sept. 2024), 1644–1656. https://doi.org/10.1109/TPDS.2024.3429625