Published in IEEE Computer Architecture Letters, 2025. DOI: 10.1109/LCA.2025.3596616 ©IEEE

Checkflow: Low-Overhead Checkpointing for Deep Learning Training

Hangyu Liu, Shouxi Luo, Ke Li, Huanlai Xing, Bo Peng

***Abstract*—During the time-consuming training of deep neural network (DNN) models, the worker has to periodically create checkpoints for tensors like the model parameters and optimizer state to support fast failover. However, due to the high overhead of checkpointing, existing schemes generally create checkpoints at a very low frequency, making recovery inefficient since the unsaved training progress would get lost. In this paper, we propose Checkflow, a low-overhead checkpointing scheme, which enables per-iteration checkpointing for DNN training with minimal or even zero cost of training slowdown. The power of Checkflow stems from the design of *i*) decoupling a tensor's checkpoint operation into snapshot-then-offload, and *ii*) scheduling these operations appropriately, following the results of the math models. Our experimental results imply that, when the GPU-CPU connection has sufficient bandwidth for the training workload, Checkflow can theoretically overlap all the checkpoint operations for each round of training with the training computation, with trivial or no overhead in peak GPU memory occupancy.**

***Index Terms*—Checkpointing; deep learning training; GPU memory optimization; scheduling.**

I. Introduction

As is known, the training of modern deep neural networks (DNN) is generally time-consuming, during which various errors might occur, making the entire job fail [1]. To reduce wasted GPU hours and lost progress caused by failure, the training workers have to create checkpoints, i.e., writing crucial model states like parameter weights and optimizer state to persistent storage, periodically. Such a mechanism enables workers to resume a failed training job from its last saved checkpoint rather than restarting from scratch; thus it is widely used in practice. Traditional coarse-grained checkpointing conserves GPU resources but leaves many iterations unprotected against failures. More aggressive, high-frequency checkpointing offers maximal resilience but incurs prohibitive overhead. Indeed, high-frequency checkpointing mechanisms generally face a dichotomy: they either minimize checkpointing latency at the cost of high peak GPU memory usage, e.g., Checkfreq [1], or conserve GPU memory but incur long stalls, e.g., stalling the training to create checkpoints [2].

Distinguished from these solutions, in this work, we introduce Checkflow, a novel framework designed to enable fine-grained, per-iteration checkpointing without causing interference to the training. Figure 1 outlines the full checkpointing process. Checkflow focuses on optimizing the offload step from GPU to CPU to reduce in-training overhead. Once a snapshot is offloaded to CPU memory, it can be safely released from GPU memory. The subsequent transfer from CPU to disk can be handled asynchronously by the CPU without stalling training. The key idea behind Checkflow is to unify checkpointing with tensor scheduling by modeling checkpoint parameters as memory-managed tensors and jointly optimizing their lifecycles within the operator schedule. Not only does Checkflow successfully hide the checkpointing latency, but it also introduces negligible memory overhead.

This work was supported by NSFSC under Grant 2025ZNSFSC0489. *(Corresponding author: Shouxi Luo.)*

The authors are with Southwest Jiaotong University, China.

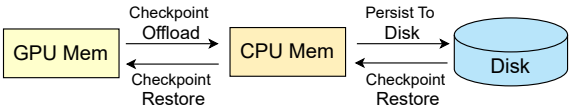


Fig. 1: Overview of a typical checkpointing workflow.

In summary, our contributions are twofold:

- **ILP-based checkpointing scheduling**: Similar to Checkfreq [1], Checkflow splits each non-activation tensor's checkpointing into two operations: *snapshot* (i.e., duplicate it in the GPU memory) and *offload* (i.e., transfer the duplication to persistent storage). Then, by encoding the scheduling of crucial tensors' checkpoint creation (snapshot), preservation, and offloading as integer linear programming (ILP) to solve, Checkflow can overlap these checkpoint operations with training, without significantly increasing the peak GPU memory occupancy.
- **Experimental verification**: Our case studies of the training of eight representative DNNs imply that, the per-iteration checkpointing design of Checkflow is low-overhead, having negligible impacts on both the overall training time and peak GPU memory usage.

In the following, we first discuss the limitations of the related work in Section II, then present the designs of Checkflow in Section III. After that, we report the evaluation results in Section IV and finally conclude the article in Section V.

II. Related Solutions and Their Limitations

As DNNs continue to grow in size and complexity, efficient checkpointing has become critical to support long-duration training and ensure fault tolerance [1]. Prior studies have proposed various checkpointing strategies, each offering distinct trade-offs in stall time, memory overhead, and applicability across models. We summarize three representative approaches.

- **Synchronous checkpointing** ensures consistency by pausing training to write the complete model state to persistent storage. This method is simple, broadly applicable, and introduces no additional memory overhead, contributing to its widespread use. However, it incurs significant stall time during checkpointing, leading to underutilization of GPU resources.
- **Asynchronous checkpointing** improves performance by overlapping checkpoint I/O with computation. For exam-

ple, Checkfreq [1] employs a two-stage pipelining strategy to effectively hide I/O latency, resulting in minimal runtime overhead under favorable conditions. However, it requires sufficient GPU memory to accommodate both model parameters and checkpoint snapshot simultaneously, which may not be feasible for large models.

- **Incremental and application-specific techniques**, such as Check-N-Run [3], reduce checkpoint size by logging parameter updates and applying quantization. These methods significantly reduce storage overhead and I/O bandwidth for models with sparse parameter updates (e.g., DLRM). However, they are tightly tailored to specific architectures and are not easily generalized to dense models or vision tasks.

In summary, existing schemes suffer from three limitations:

- **High memory overhead**: Pipelined approaches often require storing full or partial checkpoints in GPU memory, increasing peak memory usage and elevating the risk of out-of-memory (OOM) errors.
- **Limited flexibility**: Many methods treat the entire model as a single checkpointing unit, lacking support for dynamic fragmentation or adaptive scheduling based on memory and I/O availability.
- **Inefficient scheduling**: Checkpoint operations are often scheduled statically or conservatively, failing to exploit low-memory or idle GPU periods during training.

III. Checkflow

To address these limitations, we propose Checkflow, a low-overhead checkpointing scheme for deep learning training. The design of Checkflow is partially inspired by Checkfreq [1], for decoupling each checkpoint operation into snapshot-then-offload, and MODeL [4], for peak memory efficient fine-grained operator scheduling. Our core innovation lies in establishing an integrated memory constraint model that treats checkpoint parameters as specialized tensors subject to full-lifecycle management. Specifically, Checkflow:

- Encodes checkpointing related actions (i.e., snapshot creation, preservation, and offload) as first-class constraints within the optimization space to schedule;
- Guarantees that each target tensor completes snapshot before being updated while optimizing the global peak memory usage;
- Achieves efficient pipelining of computation executions and checkpoint operations.

Leveraging the predictable structure of the training task's computation graph, Checkflow formulates the scheduling of checkpointing related operations as an ILP problem. By integrating checkpoint constraints directly into operator scheduling, Checkflow could find memory-feasible execution plans that minimize the additional memory footprint of checkpointing. Currently, Checkflow focuses optimizing the offload of checkpoint from GPU to CPU. For the asynchronously persistent to disks like SSDs, the CPU has a whole training iteration time to do so. As our analysis in Section IV-B will show, with the model and/or batch size increasing, the requirements on the write throughput of the disk drops rapidly. Even if the throughput becomes the new bottleneck, selective checkpoint persist is a promising solution, which is left for future work.

Now, we explain Checkflow's designs in detail.

A. Problem Formulation

1) Preliminary: Given a worker's DNN training task, its involved computation can be represented as a directed acyclic graph (DAG), a.k.a., *computational graph*, or *dataflow graph*, $G = (V, E)$, where its vertices V are computational operators like *convolution*, *matrix multiplication*, *activation functions*, and directed edges E are multidimensional tensors denoting their data dependencies [5]. Here, E includes all tensors (activations and model weights); later we define $\hat{E} \subset E$ as the set of tensors to be checkpointed. Similar to the work of [4], we assume that the training device (e.g., GPU) can execute only one computational operator at any given time. Accordingly, the execution of all operators V can be discretized into $|V|$ steps, denoting their execution order. Currently, Checkflow is tailored to the case where operators' execution steps are already determined, e.g., by tools like MODeL [4]. Accordingly, the steps at which activation tensors would be created and discarded, and the parameter tensors would be updated, are already known in advance. Here, we let $T = \{1, \cdots, |V|\}$ be the set of all steps, and use the binary constant $l_{e,t}$ to denote whether tensor e is available at step t.

To be fault-tolerant, Checkflow creates checkpoints for tensors like the parameter weights and optimizer states, which are denoted by $\hat{E}$. Following the design of Checkfreq [1], for each $e \in \hat{E}$, Checkflow decouples its checkpoint operation into two steps: *i*) duplicate e in the GPU memory via `snapshot`, resulting in a tensor namely $\texttt{sn}(e)$, then *ii*) offload $\texttt{sn}(e)$ to the CPU for the following persistent storage. For each tensor e, Checkflow uses the binary variables $C_{\texttt{sn}(e),t}$, $P_{\texttt{sn}(e),t}$, and $S_{\texttt{sn}(e),t}$ to indicate whether e's snapshot is *created*, *preserved*, *offloaded* at step t, respectively.

Let $\bar{E} = \{\texttt{sn}(e) : e \in \hat{E}\}$, then we have

$$C_{e,t}, P_{e,t}, S_{e,t} \text{ are binary}, \quad \forall t \in T, \forall e \in \bar{E} \tag{1}$$

2) Checkpointing Scheduling: For each snapshot tensor $e \in \bar{E}$, only one action is permitted at step t.

$$C_{e,t} + P_{e,t} + S_{e,t} \leq 1, \quad \forall t \in T, \forall e \in \bar{E} \tag{2}$$

A snapshot tensor e can be preserved in the GPU memory only if it was created or preserved in the previous step.

$$P_{e,t} \leq C_{e,t-1} + P_{e,t-1}, \quad \forall t \in T \setminus \{1\}, \forall e \in \bar{E} \tag{3}$$

And it must be created and offloaded exactly once.

$$\sum_{t \in T} S_{e,t} = \sum_{t \in T} C_{e,t} = 1, \quad \forall e \in \bar{E} \tag{4}$$

At the beginning, no snapshot tensors are held by the GPU.

$$P_{e,1} = 0, \quad \forall e \in \bar{E} \tag{5}$$

The snapshot of e can be performed at step t, if and only if this tensor is available in the GPU memory at that time.

$$C_{\texttt{sn}(e),t} \leq l_{e,t}, \quad \forall t \in T, \forall e \in \hat{E} \tag{6}$$

And e's offloading can be performed if and only if it was created or resides in the GPU memory at the previous step.

$$S_{e,t} \leq C_{e,t-1} + P_{e,t-1}, \quad \forall t \in T \setminus \{1\}, \forall e \in \bar{E} \tag{7}$$

Finally, to ensure the consistency of checkpoints, creation must happen before weight update. Letting $u_{e,t}$ indicate whether weight tensor e is updated at step t, we have

$$\sum_{t'=1}^{t-1} C_{\mathtt{sn}(e),t'} \geq u_{e,t}, \quad \forall t \in T \setminus \{1\}, \forall e \in \hat{E} \tag{8}$$

3) Limited Bandwidth: Recall that the execution order of the computational operators has been scheduled. We further use τ_t to indicate the actual execution time of the operator running in step t. Let B_S be the bandwidth of the high-speed links like PCIe from the GPU memory to the CPU memory for *offloading*, and $\eta(e)$ be the size of tensor e. Then, the total volume of tensors moved in each step is limited by the product of the link's capacity and the time duration, i.e.,

$$\sum_{e \in \bar{E}} S_{e,t} \cdot \eta(e) \leq B_S \cdot \tau_t, \quad \forall t \in T \tag{9}$$

Whether checkpointing can be fully pipelined with training depends on the interplay between the workload compute time, checkpoint size, and available GPU-CPU bandwidth. Specifically, Eq. (9) implies that GPUs with higher compute power require higher GPU-CPU bandwidth for pipelining.

4) Peak GPU Memory Minimization: To minimize the peak GPU memory usage, we introduce the variable `peak_mem` and enforce the following constraint:

$$peak\_mem \geq \sum_{e \in \bar{E}} \alpha_{e,t} \cdot \eta(e) + \sum_{e \in E} l_{e,t} \cdot \eta(e), \quad \forall t \in T \tag{10}$$

where

$$\alpha_{e,t} \geq \max(C_{e,t}, P_{e,t}, S_{e,t}), \quad \forall t \in T, \forall e \in \bar{E} \tag{11}$$

Here, $\alpha_{e,t}$ indicates whether a snapshot tensor e occupies the GPU memory at step t or not. Regarding the optimization goal, Checkflow aims to minimize the $peak\_mem$ as follows.

$$\mathtt{Minimize}\ peak\_mem \tag{12}$$

B. Scheduler Designs

Algorithm 1 takes as input the computation graph G, memory limit M, the set of weight tensor snapshots S, and chunk size limit `per_size` for splitting checkpoints, and outputs a schedule satisfying all memory and scheduling constraints, or reports infeasibility if no valid solution exists.

Checkflow formulates the joint scheduling of training and checkpointing as an ILP problem, integrating memory usage, execution order, and data movement constraints (see Algorithm 1). By solving this ILP with Gurobi, Checkflow generates a schedule that overlaps checkpointing operations with training computations whenever possible, minimizing peak GPU memory usage while effectively hiding the time cost of checkpointing.

In practice, the ILP model may not always yield a feasible solution due to stringent constraints—such as those in Eq. (9)—that cause scheduling conflicts. To handle this, Checkflow employs an iterative refinement strategy as outlined in Algorithm 1. When infeasibility arises from data movement constraints (e.g., limited bandwidth or overlapping communication windows), Checkflow attempts to split large checkpoints into smaller fragments constrained by a predefined maximum size. These fragments can then be offloaded at different time steps, enabling more flexible scheduling that satisfies movement constraints.

If initial splitting still fails to meet bandwidth constraints, Checkflow further refines the split granularity until each offload fits within its compute window. Since checkpoint transfer times are generally much shorter than training computation times and the bandwidth is typically provisioned to be sufficiently large, a feasible split can always be found. If infeasibility results from exceeding the global memory limit, Checkflow incrementally relaxes the memory budget and resolves until a feasible solution is obtained.

Algorithm 1 Checkpointing Optimization with Checkflow

```
function CHECKFLOW(G, M, per_size, 𝕊)
    G' ← COPY(G)
    for S_i ∈ 𝕊 do
        G'.add(S_i)
    end for
    while true do
        (t, s, mem) ← ILPSCHED(G', M)
        if t = 0 then return (TRUE, s, mem)
        else if t = 1 then
            SPLITCHECKPOINTS(G', per_size)
        else return (FALSE, nil, nil)
        end if
    end while
end function
function ILPSCHED(G', M)
    m ←build a ILP model following §III-A
    solve m to obtain its schedule s and peak_mem pm
    if pm is optimal then return (0, s, pm)
    else if Eq. (9) is violated then return (1, null, null)
    else return (2, null, null)
    end if
end function
function SPLITCHECKPOINTS(G', per_size)
    for each checkpoint S_i in G' do
        if S_i.size > per_size then
            divide S_i into ⌈S_i.size/per_size⌉ parts
            replace S_i with divided parts in G'
        end if
    end for
end function
```

IV. Performance Evaluation

A. Methodology

We evaluate Checkflow on a diverse set of DNNs to demonstrate its effectiveness and generality. Our evaluation includes eight widely adopted architectures, AlexNet, BERT, MNASNet, ResNet, ResNet3D, Transformer, VGG, and ViT, with configurations same to these used in [4]. These models cover a wide range of application domains, including image classification, natural language processing, and video understanding. They also span both compact and large-scale architectures. All experiments are conducted using a single NVIDIA GeForce RTX 3090 GPU and an Intel Xeon Silver

4214R CPU. Checkflow is implemented in Python 3.11 and leverages Gurobi 11.0.1 to solve the ILP formulation.

We assume that the CPU-GPU PCIe bandwidth (B_S) is 63GB/s. Because of the limited 24GB memory of RTX 3090 GPU, we are unable to train models like BERT with batch size (`bs`) larger than 32. To perform consistent performance evaluation for all models, following the settings used in MODeL [4], we use `bs` = 1 and `bs` = 32 as representative settings. Here, `bs` = 1 reflects memory-constrained or latency-sensitive scenarios, while `bs` = 32 serves as a unified larger batch configuration for comparative analysis. Indeed, larger batch size settings (e.g., `bs` = 128, 256) lead to longer per-iteration computation time, which would widen the time window for overlapping and thus make per-iteration checkpointing easier to realize. For each setting, the execution order of all training operators is determined by MODeL [4]. Then, Checkflow generates a checkpointing scheduling, aiming to reduce peak memory usage while hiding the runtime cost of checkpointing operations. By default, Checkfreq [1] is used as the baseline.

B. Results

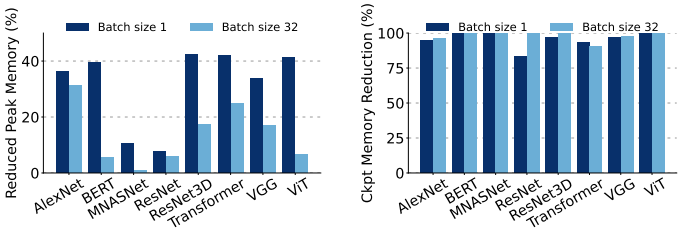
Generally, we find that Checkflow is able to overlap all checkpointing operations with the training in our tests. Regarding the peak GPU memory occupancy, as Figure 2a illustrates, compared to Checkfreq [1], Checkflow reduces the peak GPU memory by between 1% and 42% (12.8% on average for `bs` = 32). This is mainly due to its ability to overlap checkpoint memory with training memory. Models with larger checkpoints, such as BERT and Transformer, show relatively greater reductions (over 35% at `bs` = 32), while models like MNASNet and ResNet benefit less due to their smaller checkpoint sizes relative to their peak memory usage. These results demonstrate that Checkflow effectively reduces peak memory by hiding checkpoint overhead through optimized scheduling. In addition to reducing peak memory occupy, Figure 2b shows that Checkflow effectively conceals the memory overhead introduced by checkpointing through scheduling. When `bs` = 1, most models achieve no additional peak memory usage, with BERT, MNASNet, and ViT attaining this effect completely. A similar trend is observed for `bs` = 32, where five out of eight models also show no additional peak memory usage, and the remaining models still reduce over 90% of checkpoint-induced peak memory overhead.

To further analyze the impacts of B_S and `bs` on the feasibility of totally overlapping checkpointing with training, we vary the values of B_S and `bs` to reconduct tests. We observe consistent results, but only report the representative results of AlexNet and BERT in Table I due to the limitation of space. As is shown, while BERT (`bs` = 32) remains feasible even at 20 GB/s due to its longer compute time per iteration, AlexNet is feasible at 63 GB/s but becomes infeasible at 40 GB/s and below when `bs` = 32. However, increasing AlexNet's batch size to 1024 restores feasibility across all bandwidth settings. This confirms our approach is more suitable for large models or large batch size, which provide wider scheduling windows for checkpointing.

Now, we further theoretically analyze the requirements on the write throughput of disk to support per-iteration checkpoint

TABLE I: Feasibility of Checkflow across various scenarios.

Model	batch size	63 GB/s	40 GB/s	20 GB/s
AlexNet	32	✓	✗	✗
AlexNet	1024	✓	✓	✓
BERT	32	✓	✓	✓



(a) Reduced peak GPU memory (b) Reduced memory overhead

Fig. 2: Experimental results.

persist. We observe that it only becomes a limiting factor when a small model is trained with a small batch size, e.g., AlexNet (`bs` = 1) requires a write throughput larger than 37 GB/s in theory. As the model and/or batch size increase, the required throughput drops rapidly and would be smaller than 5 GB/s.

Lastly, although Checkflow relies on solving ILP models using Gurobi, it can obtain the scheduling plan within tens of seconds. As Checkflow works at the compilation stage, this solving time does not affect the actual training.

V. Conclusion and Future Work

In this paper, we propose Checkflow, a low-overhead per-iteration checkpointing solution for DNN training. By taking advantage of decomputing and scheduling, Checkflow is able to achieve substantial peak-memory savings and incurs negligible preprocessing overhead, enabling efficient, fine-grained checkpointing without impacting training throughput.

Limitations and future work. While the current version of Checkflow is tailored to the case where the training job only involves a single GPU, its core designs are compatible with mixed-precision training and can be extended to multi-GPU or multi-node scenarios by formulating inter-device collective communication as specific schedulable operations. However, extending it to support tensor and model parallelism would introduce more complex dependencies and heterogeneous communication patterns, which pose both system-level and algorithmic challenges. In addition, to make Checkflow practical, full integration with training frameworks like PyTorch is also needed to validate Checkflow's practicality and address possible interference with training operators. We leave all these as important directions for future work.

References

[1] J. Mohan, A. Phanishayee, and V. Chidambaram, "CheckFreq: Frequent, fine-grained dnn checkpointing," in *FAST*, 2021, pp. 203–216.
[2] E. Rojas *et al.*, "A study of checkpointing in large scale training of deep neural networks," *arXiv preprint arXiv:2012.00825*, 2021.
[3] A. Eisenman *et al.*, "Check-N-Run: a checkpointing system for training deep learning recommendation models," in *NSDI*, 2022, pp. 929–943.
[4] B. Steiner *et al.*, "Model: memory optimizations for deep learning," in *ICML*, 2023, pp. 32 618–32 632.
[5] L. Zheng *et al.*, "Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning," in *OSDI*, Jul. 2022, pp. 559–578.