PCcheck: Persistent Concurrent Checkpointing for ML

Foteini Strati*
ETH Zurich
Switzerland

Michal Friedman*
ETH Zurich
Switzerland

Ana Klimovic
ETH Zurich
Switzerland

Abstract

Training large-scale machine learning (ML) models is expensive and time-intensive, consuming many hardware accelerators for days or weeks. As the scale of hardware deployments and training time continue to grow, the probability of failures also increases. The desire to use cheaper cloud resources, such as spot VMs, to lower costs also dramatically increases the frequency of failures. The standard approach to deal with failures is to periodically pause training and checkpoint model parameters to persistent storage. Unfortunately, today's checkpointing mechanisms introduce high overhead when applied at high frequencies, yet frequent checkpointing is necessary to avoid long recovery times.

We present a concurrent checkpointing mechanism, PCcheck, that allows frequent checkpointing with minimal overhead. Our framework supports persisting checkpoints to SSD and persistent main memory (PMEM) for both single-machine and distributed settings. PCcheck enables checkpointing as frequently as every 10 iterations for detailed monitoring and fast recovery times in case of failures, while maintaining minimal (3%) overhead on training throughput.

ACM Reference Format:
Foteini Strati, Michal Friedman, and Ana Klimovic. 2025. PCcheck: Persistent Concurrent Checkpointing for ML. In *Thirtieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '25), March 30 – April 3, 2025, Rotterdam, The Netherlands.* ACM, New York, NY, USA, 17 pages.

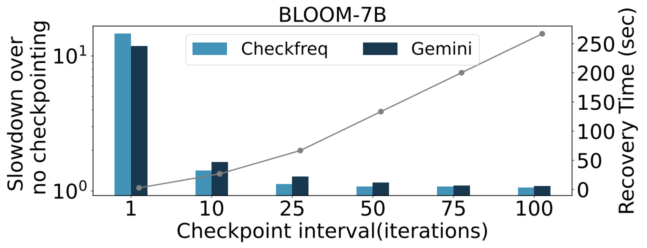


Figure 1. Performance impact of CheckFreq and Gemini checkpointing for BLOOM-7B training on an A100-40GB GPU cluster, without failures. The secondary y-axis and the grey line show the recovery time when a failure occurs.

1 Introduction

Deep neural networks (DNNs) are the foundation for many modern artificial intelligence (AI) applications [34, 37, 43, 73]. While DNNs offer high accuracy, these models are often computationally intensive, time-consuming, and expensive to train [72]. Large-scale DNNs can take several days or even weeks to train across tens to thousands of hardware accelerators, such as GPUs and TPUs [76].

The longer a training job executes and the more hardware accelerators it uses, the higher the probability of the job encountering failures. Failures include hardware, network, and power failures, as well as software bugs and out-of-memory issues, which commonly occur in production DNN clusters [27, 40]. A study from Meta shows that 50% of ML training jobs encounter a failure within less than 16 minutes of execution [28]. Microsoft reports a 45 minute mean time between job failures in a multi-tenant GPU cluster [39]. Another common type of failure that a DNN job may experience is preemption. DNN cluster managers often schedule lower priority jobs on spare resources and preempt jobs as the load varies [31, 35, 58]. Similarly, for model training on public clouds, it is increasingly common to use preemptible (i.e., "spot") virtual machines as they cost 60-90% less than on-demand VMs [17, 66]. However, cloud providers can preempt spot VMs on short notice, resulting in significantly higher job failure rates [16, 53, 75]. For example, Thorpe et al. [66] found that a GPU cluster of 64 spot VMs in AWS EC2 experienced 127 distinct preemption events in 24 hours.

The standard approach for dealing with failures or preemptions in DNN jobs is to periodically checkpoint model weights and optimizer state to persistent storage. After a failure occurs, the job resumes training from the state captured in the most recent persistent checkpoint. To avoid long retraining times after a failure, a training job should frequently checkpoint its state. High frequency checkpointing is particularly crucial when training on unreliable resources, such as spot VMs. Although cloud providers may give a preemption warning (e.g. 30 sec in Google Cloud and Azure [33, 49] or 2 min in AWS [18]), this grace period is often not enough time to persist large models. For example, the OPT-1.3B model has 16 GB of model and optimizer state, which takes 37 seconds to persist to a 1TB-SSD on a a2-highgpu-1g VM in

*Equal Contribution

Google Cloud (using `torch.save` and `flush` as CheckFreq proposes [50]), exceeding the grace period.

Besides enabling failure recovery, frequent checkpoints are also commonly used for debugging model training dynamics, such as accuracy divergence [48]. Training accuracy may deviate from the anticipated trend due to unexpected hardware or software behavior, which requires debugging and re-training the model [38, 61]. Checkpoints help avoid training from scratch, which is often prohibitively expensive.

Unfortunately, current periodic checkpointing mechanisms introduce high overhead in DNN training jobs when applied at high frequency. Figure 1 shows training throughput slowdown when using state-of-the-art mechanisms, CheckFreq [50] and Gemini [68], while training the BLOOM-7B model [71] compared to training the same model with no checkpointing. CheckFreq optimizes checkpointing by overlapping the "snapshot" phase (copying weights to DRAM) and the "persist" phase (flushing the checkpoint to durable storage) with the training itself. Gemini optimizes checkpointing by leveraging remote CPU memory instead of persistent storage. However, both mechanisms still have more than 10% overhead when checkpointing every 50 training iterations or less. We find that a key reason for this overhead is that these mechanisms handle *only one checkpoint at a time*, so a checkpoint cannot be copied to DRAM (local or remote) until the previous checkpoint has successfully been persisted. This becomes a critical bottleneck when checkpointing frequently.

Figure 2 shows training goodput[1] for the BLOOM-7B model assuming GPU resource availability from a GPU spot VM trace collected by André et al. [16]. The trace captures the preemptions experienced over a 16-hour time window when requesting a cluster of 64 A100 GPUs on spot VMs in Google Cloud. The training goodput depends on the checkpointing overhead and the frequency of failures. We measure how the goodput varies with different checkpoint intervals compared to an ideal system that does not stall training while saving checkpoints. The larger the checkpoint interval, the less often we checkpoint and the more retraining needs to be done to reconstruct the state after a failure, reducing goodput. Due to CheckFreq's and Gemini's inability to handle multiple concurrent checkpoints, they have high overhead when checkpointing more frequently than every 50 iterations as the next checkpoint must wait for the previous one to finish. Overall, CheckFreq and Gemini achieve only up to 66% and 58% of the ideal peak goodput for this trace of GPU spot VM resource availability, respectively.

We propose PCcheck[2], a framework for DNN training that supports multiple *concurrent* checkpoints in parallel. Concurrent checkpoints can help reduce idle GPU time and

[1]*Goodput* is useful throughput, i.e., batches per second discounting batches that are recomputations of previously executed batches after a failure.
[2]PCcheck is available at

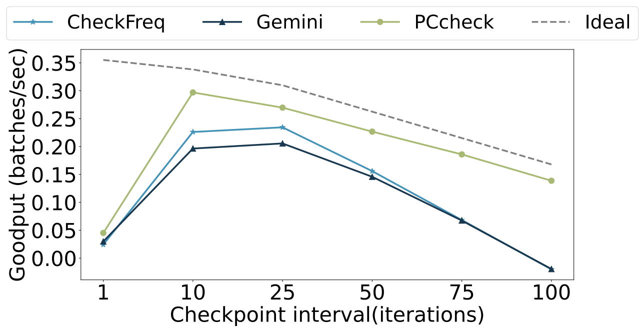


Figure 2. Goodput as a function of checkpoint interval for the BLOOM-7B model training on a spot GPU VM cluster on Google Cloud.

increase checkpoint frequency, since training does not have to wait for the previous checkpoint to finish, before initiating a new one. However, naively issuing concurrent checkpoints can increase CPU memory and storage overheads, as well as PCIe and storage bandwidth contention, which could degrade training throughput. Hence, we design PCcheck to carefully select the number of concurrent checkpoints and minimize the time per checkpoint by optimizing the copying mechanism to persistent storage, pipelining GPU-CPU snapshotting and persisting, and using multiple threads to persist each checkpoint. This allows PCcheck to support far more frequent checkpointing than prior systems, leading to faster recovery times as fewer training iterations need to be recomputed after a failure. We evaluate PCcheck on a variety of models from vision and natural language domains, using SSD and PMEM as two different storage medias. We show that PCcheck allows for frequent checkpointing (e.g. every 10 iterations) with minimal (3%) overheads on training throughput. Moreover, PCcheck achieves up to 2.86× higher goodput than state-of-the-art checkpointing systems and close to ideal (see Figure 2) for DNN training on the GPU spot VM trace, due to its ability to minimize stalls while checkpointing frequently.

2 Background

2.1 Why Checkpoint?

Fault Tolerance. As DNN jobs commonly run on large hardware deployments, several studies have shown that these jobs are highly susceptible to failures and preemptions [27, 28, 39, 40]. Job failures are even more common when training DNN models on slack resources (e.g., using spot VMs in public clouds or running low-priority jobs in shared clusters). Thorpe et al. [66] show that a GPU cluster of 64 spot VMs in AWS EC2 experienced 127 distinct preemption events over 24 hours. André et al. [16] monitor a cluster of 64 A100 GPUs in GCP, and observe 26 preemptions over 3.5 hours. Checkpointing during training is essential to minimize retraining time [16, 17, 39, 53, 67]. As Figure 2 shows,

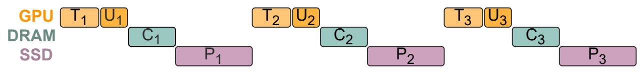


Figure 3. Traditional checkpointing in PyTorch [56] / TensorFlow [14]. T_x and U_x are the model training and update steps for iteration x. C_x is the time to copy checkpoint state from GPU to DRAM. P_x is the time to persist state to storage.

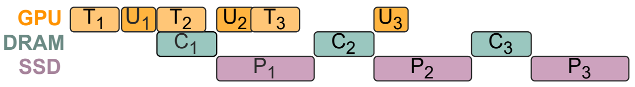


Figure 4. CheckFreq [50] overlaps training (T_{x+1}) with copying (C_x). Next checkpoint is copied when previous persisted.

checkpointing frequently (e.g. ≤ 25 iterations in this case) is vital to ensure high training throughput in environments with high failure/preemption rates.

Monitoring and Debugging ML Models. Checkpoints are also useful for monitoring and debugging DNN training. DNN accuracy can "derail" during training due to data outliers, floating point overflows, exploding or vanishing gradients [61], and logic hardware failures [38]. Hence, ML practitioners need tools to efficiently and accurately monitor and debug model state. Therefore, ML monitoring, visualization, and debugging tools [15, 61, 63, 70] are widely used.

Most of these tools capture features such as model parameters, gradients, and evaluation metrics (loss, accuracy, etc.) throughout training, persist them to storage, visualize them, and enable various queries for debugging. Amazon SageMaker Debugger [61] allows users to specify which tensors to snapshot, and how often. Cockpit [63] captures model parameters and gradients and visualizes them as histograms for debugging. Pythia [20] provides frequent checkpoints of large language models, for better observability and interpretability. As these tools checkpoint large training states from GPUs to persistent storage, they can significantly degrade training throughput [61]. The checkpoint mechanism should be as lightweight as possible to avoid high overheads when using these monitoring tools.

2.2 Checkpointing Mechanisms for DNN Training

Traditional checkpointing, as implemented in PyTorch [56], TensorFlow [14] and MxNet [22] typically persists model weights as shown in Figure 3. Training (T) involves fetching input data and computing forward and backward passes. The update phase (U) updates the model weights. The weights are then copied to the DRAM (C) and persisted to the persistent storage (P). All these steps occur sequentially, meaning that training stalls until the weights are persisted.

To reduce performance overheads, CheckFreq [50] pipelines checkpointing to some extent, enabling training to continue concurrently with the checkpoint itself, but allowing only one checkpoint at a time. Once weights are copied to DRAM, training resumes, until the next checkpoint. If the former checkpoint has not persisted yet, the next one will wait. Figure 4 shows this process, assuming we checkpoint every iteration. The second iteration's copying waits until the previous checkpoint is persisted, leaving the GPU idle. According to Figure 1, the slowdown of CheckFreq on BLOOM-7B spans of ranges from 15× to 1.05× when the checkpointing occurs every 1 to 100 iterations, respectively.

GPM [55] targets persistent memory (PMEM) and is not specific to ML training. GPM leverages NVIDIA's unified virtual memory (UVM) [11] between the GPU's main memory and DRAM to copy data using GPU kernels rather than the GPU's dedicated copy engines. As we show in § 5, GPM has significant overheads in DNN jobs as it stalls training while persisting state (similar to Figure 3 but without the intermediate DRAM copy).

DeepFreeze [54] reduces checkpoint overheads in distributed data parallel training, but focuses on CPU-only settings, where training computations are much slower than on GPUs. Hence, the system does not address the checkpoint overheads that arise in modern DNN hardware environments.

Gemini [68] proposed an alternative to checkpointing for distributed training, by snapshotting the training state of one machine to the CPU's memory of other machines. Gemini pipelines training state copies to a remote machine's GPU first and then its CPU during the forward/backward pass. This process is interleaved with activation and gradient exchange. Gemini assumes that the network will have low latency and high bandwidth, thus the GPU-GPU transfer will be fast. However this is not always the case. For example, on GCP a2-highgpu-1g, the highest achievable network bandwidth is 1.88 GB/sec [2]. As shown in § 5, although Gemini avoids the slow disk bandwidth, it still performs poorly over low-bandwidth networks.

Gupta et al. [36] propose just-in-time checkpointing. In contrast to the previous related works based on periodic checkpointing, this method transfers the training state from GPU to persistent storage only upon detecting a failure. In particular, healthy workers participating in collective operations detect a worker has failed, and then checkpoint their GPU state in a persistent file. This technique assumes that the contents of the failed worker are replicated (since large-scale distributed training is commonly done with a combination of data and pipeline parallelism [17, 66]), ensuring that the model state is not lost upon a failure. The authors assert that simultaneous multi-node failures are extremely rare, making this a valid assumption. However, this might not be true when training over preemptible resources, where *bulky* VM preemptions are very common [16, 66]. Since our work targets environments where multi-node failures or preemptions are common, we focus on periodic checkpointing in the remainder of the paper, which ensures that training can recover at every scenario.

Takeaway: *Current checkpointing mechanisms have high overhead when applied at high frequency in DNN training on high-throughput accelerators, like GPUs. We identify that the main overhead is due to checkpoints stalling until a previous checkpoint persists. To support high-frequency checkpoints with minimum stalls during training, checkpointing mechanisms need (but currently lack) support for efficiently managing multiple concurrent checkpoints.*

2.3 Hardware Architecture Factors

Checkpointing performance depends on several hardware factors, such as the data copy engine and the storage media.

Data Copy Engines. GPUs consist of execution engines (with multiple Streaming Multiprocessors (SM)), DMA copy engines, and memory. Data is typically transferred using the copy engines, which can potentially copy data in parallel with kernel execution on the SMs [5]. Data can also be copied with GPU kernels, as in GPM [55].

Storage Media. We consider both persistent non-volatile memories (PMEM) and SSDs. Compared to SSD, PMEM is byte–addressable, and has comparable access speeds to DRAM. Upon a crash, only the PMEM's content survives. Moreover, the order in which data is written to the cache may differ from the order in which the content reaches PMEM, leading to inconsistent states upon a failure. To address this problem, a process can write-back a particular value, or issue non-temporal stores, followed by a fence [74]. PMEM and SSD can be accessed via memory mapping, and SSDs require an explicit function call to ensure persistence (e.g., `msync`). Although Intel recently discontinued its PMEM manufacturing [13], PMEM technology can still be part of the memory hierarchy through CXL [23]. Compute Express Link (CXL) is a cache-coherent interconnect open interface based on the PCIe channel. It is also byte-addressable and allows accesses in a cache line granularity, while maintaining coherence and consistency. Hence, our support for PMEM can also be applicable for the upcoming CXL technology.

3 PCcheck Design

As discussed in § 2.2, systems like CheckFreq [50] and Gemini [68] have high overhead when checkpointing frequently because they stall training until the previous checkpoint persists. Yet, checkpointing frequently is important for model training quality monitoring and handling preemptions and failures that often arise at scale. To avoid these stalls, we propose PCcheck, a framework that orchestrates multiple *concurrent* checkpoints with low overhead, enabling efficient fault-tolerance and debugging. While concurrent checkpoints can help reduce idle GPU time and increase checkpoint frequency, they also increase CPU memory and storage demands and contention, leading to potential performance issues. PCcheck optimizes the number of concurrent checkpoints and minimizes time per checkpoint using pipelining techniques and multiple threads. These techniques enable PCcheck to achieve faster and more frequent checkpointing than systems like CheckFreq [50], GPM [55], and Gemini [68].

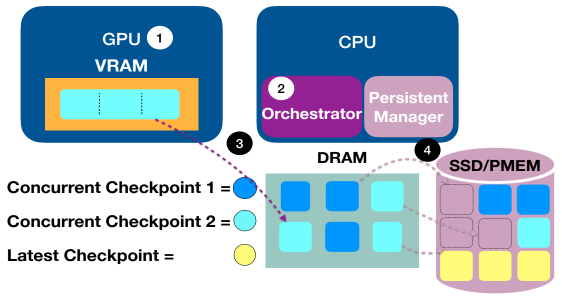


Figure 5. System architecture

§ 3.1 describes PCcheck's system architecture. § 3.2 describes the throughput-memory tradeoff introduced with PCcheck. § 3.3 describes the hardware mechanisms we use to optimize the data path from GPU to persistent storage. Finally, in § 3.4 we show how we can tune PCcheck's configuration parameters.

3.1 Overview

Figure 5 shows the PCcheck system architecture and the checkpointing steps during DNN training. PCcheck's orchestrator coordinates ongoing checkpoints. PCcheck's persistent manager keeps track of the mmaped checkpoint addresses and uses multiple threads to copy checkpoint data from DRAM to SSD or PMEM.

Life of a Checkpoint. In Step ① in Figure 5, the GPU executes training iterations until it reaches the next checkpoint. In Step ②, the orchestrator initiates checkpointing by first finding a free memory region for the checkpoint in DRAM using a lock-free queue [52] that stores free addresses. In Step ③, the orchestrator triggers the GPU copy engines to copy checkpoint state from GPU memory to DRAM in chunks. In Step ④, the orchestrator triggers the persistent manager to persist the checkpoint chunks from DRAM to storage using multiple threads. While chunks from a particular checkpoint may be scattered in DRAM, the orchestrator ensures that all the checkpoint's chunks are ordered and written to consecutive addresses on persistent storage. The persistent manager also keeps track of the latest fully persisted checkpoint, which is used to recover from failures. Users can tune configuration parameters (see Table 2). PCcheck provides a tool to optimize configuration parameters for a given workload and hardware setup (see § 3.4).

Minimizing Checkpoint Stalls. Figure 6 depicts PCcheck's concurrent checkpointing over time, assuming checkpointing occurs every iteration. As soon as the GPU updates

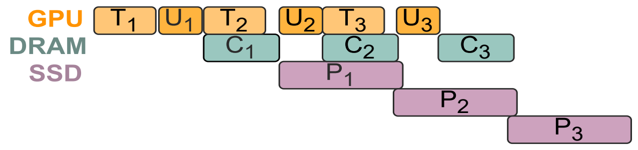


Figure 6. PCcheck algorithm over time, checkpointing every iteration. T_x and U_x are the model training and model update steps for iteration x, respectively. C_x is the time to copy state from GPU to DRAM. P_x is the time to persist state to storage.

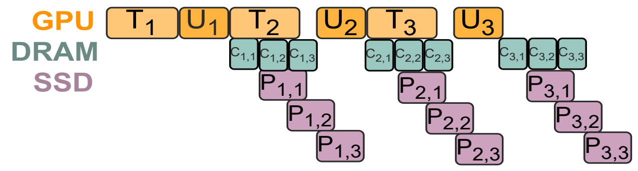


Figure 7. PCcheck with pipelining. Checkpoints are divided into 3 chunks. PCcheck overlaps copying with persisting.

the model weights (U_1), the orchestrator initiates a copy of the checkpointed state to DRAM (C_1), while the GPU continues training the model for the next iteration (T_2). When it is time to copy the next update (C_2), PCcheck does not stall the copy even though the first checkpoint has not been persisted yet (P_1). PCcheck's support for multiple concurrent checkpoints is a key feature compared to CheckFreq (see Figure 4) and Gemini [68]. PCcheck also reduces the time it takes to persist a checkpoint (e.g., P_1) compared to other state-of-the-art techniques by parallelizing writes from DRAM to persistent storage with multiple threads. The main bottleneck now becomes the model weight update. The stall between T_2 and U_2, could be eliminated by keeping an extra copy of model weights in GPU memory. One copy of the model weights can then be updated, such that the next training iteration can proceed (T_3), while the other copy of model weights (from U_1) remains intact as it is copied to DRAM. However, since GPU memory is limited and expensive, we find it is not worth consuming extra memory on the GPU to avoid such stalls.

To further reduce stalls, PCcheck supports pipelining, as shown in Figure 7. Instead of copying the entire updated weights (e.g., all of C_1 in Figure 6) and only then start persisting the data, PCcheck can split the data into multiple chunks (e.g., 3 chunks in Figure 7). By using chunks, it is also possible to start overwriting the already-persisted chunks in DRAM with the new weights on the GPU side.

Checkpointing for Distributed Training. In multi-node training environments, PCcheck runs one orchestrator per node. For pipeline parallel training, where the model is split across nodes, each node checkpoints its model partition independently. However, as multiple checkpoints take place in parallel, PCcheck must ensure that the latest checkpointed model partition in each persistent device corresponds to the same iteration for all workers. This requires an extra coordination step between the orchestrators across all nodes, which has negligible overhead compared to the actual training. The same approach can also be followed with Fully Sharded Data Parallelism [59], where each worker has its own model shard. When a combination of data and pipeline parallelism is used, the checkpoint state of each pipeline stage is partitioned among the data parallel replicas of this stage, reducing the overall checkpointing overhead. The different replicas also coordinate to ensure that the latest persisted checkpoint corresponds to the same iteration.

Algorithm	GPU Mem	DRAM	Storage
CheckFreq	m	m	$2 \times m$
GPM	m	0	$2 \times m$
Gemini	m + buffer	m	0
PCcheck	m	m to $2 \times m$	$(N+1) \times m$

Table 1. Memory footprint, where m is checkpoint size and N is the number of concurrent checkpoints.

3.2 Throughput-Memory Tradeoff

Table 1 compares PCcheck's memory and storage requirements to other systems. GPM, CheckFreq, and Gemini allow only one checkpoint at a time, so they do not require space larger than one checkpoint size in DRAM. PCcheck can leverage extra memory capacity to support concurrent checkpoints more efficiently. In the pipelined version, the orchestrator manages the checkpoint in chunks. Once a chunk is persisted, it is freed and available for following chunk copies from the GPU. However, when all CPU memory chunks are occupied (not yet persisted), upcoming checkpoints need to wait for free chunks in DRAM. Users can set the number of DRAM chunks that PCcheck is free to use. In general, as we will show in § 5.4.3, a larger number of DRAM chunks allows PCcheck to minimize stalls. However, even with strict memory constraints, PCcheck leads to significant improvements over the baselines.

PCcheck also consumes more space in the persistent storage. PCcheck requires $(N+1) \cdot m$ to allow N concurrent checkpoints and guarantees at least one valid checkpoint at any time. Since the state-of-the-art techniques allow only one checkpoint at a time, $N = 1$ and therefore, they require $2 \cdot m$ persistent storage. Gemini does not consume any space in persistent storage but rather an extra buffer (32 MB) on a GPU since it copies the data to a remote GPU first.

3.3 Data Copy Mechanisms

We now describe the hardware mechanisms PCcheck uses to optimize data copying.

GPU to DRAM. To optimize the data copy bandwidth from GPU to DRAM, we considered both GPU copy engines and copy kernels, as used by GPM [55]). We also found that the bandwidth depends on memory pinning and Direct Data I/O (DDIO) configurations. Using copy engines with pinned memory and DDIO enabled (such that I/O ends up in the last level cache instead of DRAM) yields the highest performance. Though copy engines have some initialization overhead, this is negligible for GB-size checkpoints. Furthermore, copy engines do not consume precious compute resources on the GPU. When pinned memory[3] is used, copy engines leverage direct memory access (DMA) to copy data directly to DRAM, without involving the CPU or GPU's compute resources.

We considered using GPU copy engines to copy data directly to PMEM or using peer-to-peer (P2P) PCIe technologies such as GPUDirect Storage [4] for direct GPU-to-SSD transfers. However, since PMEM and SSD bandwidth is lower than DRAM, PCcheck achieves higher overall throughput by overlapping fast GPU-to-DRAM copies with slower persistent writes to storage. Furthermore, GPUDirect is not widely available in the cloud or our on-premise machines.

DRAM to PMEM/SSD. PCcheck supports both SSD and PMEM. The choice of storage influences the persistence mechanisms, the optimal configuration of parallel writers, and the number of concurrent checkpoints. We compare two mechanisms for writing to PMEM: using non-temporal store instructions (bypassing the cache) and using a `clwb` instruction, which writes back data from the cache to PMEM. Both instructions require a `fence` for persistence. Since PCcheck only writes data once without reading it back, bypassing the cache with a non-temporal store instruction followed by an `sfence` achieves higher bandwidth (4.01GB/sec on our machine, whose specs are described in § 5.1) compared to the `clwb` instruction approach (2.46GB/sec). This is consistent with insights from prior work [74]. For SSD, PCcheck writes to an mmapped memory region and persists using `msync()` after every checkpointing write.

3.4 Configuring PCcheck

Table 2 lists PCcheck's configuration, system parameters, and user constraints. The user can choose configuration parameters, while the system parameters depend on the platform. Given user constraints, such as memory consumption and acceptable checkpointing overhead, PCcheck optimizes the configuration parameters for a given DNN training workload. Users can also override parameter values. § 5.4 shows a sensitivity study for all the parameters.

The user defines the maximum DRAM (M) and storage size (S) dedicated for checkpoints ($M \leq S$), and a maximum slowdown q due to checkpointing ($q \geq 1$). Our goal is to find the minimum checkpoint interval f (in terms of training iterations) so that the checkpoint overhead remains smaller than q, while the recovery time in case of a failure is minimized. To saturate the GPU-CPU bandwidth, the buffer size (b) should be large enough [55] and is chosen by the tool based on T_G. In our setup, we empirically set it to 100-500 MB. Similarly, the number of writer threads per checkpoint (p) is ideally 2 to 4. The number of chunks is $c = \frac{M}{b}$, and the number of concurrent checkpoints is $N \leq \frac{S}{m} - 1$ (note that we might end up using less storage).

Assuming we checkpoint every f iterations , we have up to N concurrent checkpoints, and the time to write a checkpoint (i.e. from the moment it started copying from GPU until is persisted) is, at worst case (i.e. when all N checkpoints are ongoing and there is maximum contention), T_w. Furthermore, the total number of training iterations is A. For simplicity, we assume that f divides A. When there are no checkpoints, the runtime is: $runtime_0 = A \cdot t$. However, when there is a checkpoint every f iterations, assuming $N = 1$, the runtime changes to:

$$runtime_1 = f \cdot t + \max\left(Tw, f \cdot t\right) \cdot \left(\frac{A}{f} - 1\right) + Tw$$

where $\frac{A}{f} - 1$ represents the checkpoint interval between the first and last checkpoint, and is multiplied by the maximum time between running the next interval, $f \cdot t$, and the checkpointing time, Tw. We take the maximum value since we have only one concurrent checkpoint and it is written in parallel with training. We add $f \cdot t$ and T_w to represent the first interval (before the first checkpoint has even started) and the last interval that does not run with checkpointing in parallel.

If $N > 1$, we can extend the $runtime_1$ to the following (again, for simplicity, we assume $N \cdot f$ divides A):

$$runtime_2 = f \cdot t + \max\left(Tw, N \cdot f \cdot t\right) \cdot \left(\frac{A}{f \cdot N} - 1\right) + Tw$$

The interesting case is when $Tw > N \cdot f \cdot t$ (meaning that training has to stall). In that case, we have:

$$runtime_2 = f \cdot t + Tw \cdot \frac{A}{f \cdot N}$$

Given a specific overhead q, we want $runtime_2 \leq q \cdot runtime_0$, thus:

$$f \cdot t + Tw \cdot \frac{A}{f \cdot N} \leq q \cdot A \cdot t \qquad (1)$$

Since usually, A is very large (i.e. training can span days), we can ignore the $f \cdot t$ term in 1. Then, we have:

$$Tw \cdot \frac{A}{f \cdot N} \leq q \cdot A \cdot t \Rightarrow f \geq \frac{Tw}{N \cdot q \cdot t} \qquad (2)$$

We note that a larger N would theoretically lead to a shorter waiting time of training. However, larger N means more concurrent checkpoints contending on the storage device bandwidth, thus increasing T_w. Our goal is to find the minimum $\frac{T_w}{N}$, based on the fact that $N \leq \frac{S}{m} - 1$. $\frac{T_w}{N}$ depends on the checkpoint size m, iteration time t and the storage device bandwidth T_s. For example, if $N = 1$, $T_w = \frac{m}{T_s}$. Since m and t are given, our tool empirically finds the best N to minimize

[3]memory that has been registered with *cudaHostRegister()*

Configuration Parameters	System/Model Parameters	User Constraints
# concurrent checkpoints (N)	GPU-CPU PCIe bandwidth (T_G)	Total DRAM size (M)
# parallel writer threads (p)	Storage bandwidth (T_S)	Total storage size (S)
DRAM buffer size (b)	Iteration time (t)	Checkpoint overhead (q)
# chunks (c) in DRAM	Checkpoint size (m)	# iterations (A)
Checkpoint interval (in iterations) (f)		

Table 2. Configuration and system parameters.

$\frac{T_w}{N}$. It initiates a checkpoint every t seconds. It then varies N in $[1, \frac{S}{m} - 1]$, measures T_w for each checkpoint, and finds the optimal N. In practice, we need to check only a few values of N, so this profiling round has negligible overhead to the overall training. Once N^* is fixed, the minimum checkpoint interval f^* is:

$$f^* = \left\lceil \frac{T_w}{N^* \cdot q \cdot t} \right\rceil \quad (3)$$

The optimal checkpoint frequency might vary throughout training due to contention for shared resources, such as GPU-CPU PCIe bus, or disk bandwidth. For example, vision model training is input-bound [51], and LLM training commonly offloads activations to CPU memory and disk [44]. This behavior might necessitate adapting the checkpoint frequency during training. We plan to extend PCcheck by monitoring training throughput and traffic between GPU, CPU, and storage, and adapt (3) accordingly.

4 PCcheck Algorithm

We now describe the concurrent checkpointing algorithm in detail, including the recovery procedure and its guarantees.

4.1 Persisting Checkpoints

The persistent manager tracks memory-mapped addresses for writing checkpoints using an algorithm based on a global counter and three classes - *Check_meta*, *Data*, and *Queue*. The global counter advances with each new checkpoint request, ensuring the order among checkpoint requests. The *Check_meta* class represents a single checkpoint's metadata, holding its counter and a pointer to its data. The *Data* class contains the recorded checkpoint's data. *Queue* is a lock-free queue based on [52], holding available slots for storing checkpoints, apart from the latest valid checkpoint. When the orchestrator requests a new checkpoint slot, it dequeues an element from the queue, indicating where the new data can be recorded. Every time a checkpoint's recording becomes irrelevant, (i.e. a thread persists a more recent checkpoint), PCcheck adds the address of the old checkpoint to the end of the queue. The persistent manager keeps track of a pointer to the latest persistent checkpoint.

PCcheck maintains the invariant that there is at least one persistent checkpoint, which is the latest among the last $N+1$ checkpoints, and cannot be overwritten. If there are only N threads that update new checkpoints, the algorithm is lock-free. Furthermore, we always guarantee that old checkpoints will not interfere with newer ones by implementing it with proper use of compare-and-swap (CAS) instructions and comparing appropriate checkpoint counters.

Listing 1. Checkpoint Operation

```
void checkpoint (data) {
    // check the last updated checkpoint
    check_meta* last_check = *CHECK_ADDR;
    // get a new counter for the current checkpoint
    long curr_counter = atomic_add(&g_counter, 1);
    // find free space to update the new checkpoint
    int data_location = 0;
    while (true) {
        data_location = free_space.deq();
        if (data_location != EMPTY) break;
    }
    // prepare thread parameters & update memory
    threads[i] = new thread(&persist, &data);
    for (int j = 0; j < num_threads; j++)
        threads[j]->join();
    check_meta check = {data_location, curr_counter};
    memcpy(curr_check, &check, sizeof(check_meta));
    BARRIER(curr_check);
    while (true) {
        bool res = CAS(CHECK_ADDR,
                      last_check, curr_check);
        check_meta* check = *CHECK_ADDR;
        if (res) { // success. Persist and enq old location
            BARRIER(CHECK_ADDR);
            int free = last_check.data_location;
            free_space.enq(free);
        } else if (check->counter < curr_counter) { // retry
            last_check = check;
            continue;
        } else { // more updated checkpoint was registered
            BARRIER(CHECK_ADDR);
            free_space.enq(data_location);
        }
        break;
    }
    return;
}
```

Listing 1 contains the pseudo-code for the checkpoint operation. For readability, we describe PCcheck's simplified non-pipelined version, depicted in Figure 6. The checkpoint operation starts by finding the last persistent checkpoint located in CHECK_ADDR (Line 3). Then, it gets a new counter by calling atomic_add with the global counter in Line 5. The global counter helps in ordering all checkpoints.

In Lines 6–11, PCcheck finds space for persisting the current ongoing snapshot, relying on the guarantee that the last persistent checkpoint is not in the queue. Then, every thread saves its part of the checkpoint in the persistent memory in the *persist* function (Lines 12–13). In the PMEM version, every thread must also call a *fence()* within the *persist* function. The *fence()* is internal to each CPU, meaning that the main thread, which spawned the threads (in the checkpoint operation), cannot call a fence to cover all data that was written to PMEM. In the SSD version, however, the main thread can call a single *msync()* with the checkpoint address and persist the data, improving performance.

In Lines 16– 18, we persist the start address of the checkpoint. We emphasize that the actual training state is persisted only *once* per checkpoint, and the following procedure refers to persisting the *pointer* to the latest checkpoint. BARRIER refers to writing back the data to the persistent storage with an adequate fence/sync instruction. For correctness, it is important to persist the entire data and the checkpoint that points to this data *before* CHECK_ADDR is updated. When the data is persisted, the current thread tries to update CHECK_ADDR using a CAS instruction. Note that since the current checkpoint is sampled (Line 3) *before* getting the counter for the new checkpoint (Line 5), the CASing trial is guaranteed to be legal. A trial is legal if the new checkpoint which is trying to CAS has a bigger counter than the current checkpoint. If the trial is successful, we persist the checkpoint address and return the location of the older checkpoint to the queue, as it is free to be reused. If it failed, there are two potential reasons. First, another concurrent checkpoint was concurrently recorded. If the value of the other concurrent checkpoint counter was higher, then we persist the checkpoint's address, return our location to the free queue and exit (Lines 30–32). Otherwise, the other checkpoint is outdated and should be updated. In this case, we retry to record the newer checkpoint address and persist upon success. BARRIER guarantees persistence, and we call it both after updating the new checkpoint address or failing in this update as a more updated checkpoint has been registered. Once the operation is over, the most updated checkpoint is registered and persisted.

Pipelining and Using Chunks. There are only a few modifications to the described pseudocode when employing pipelining and splitting the checkpoint into chunks. Specifically, when a new checkpoint is initialized, it gets an address in the persistent device for writing, as shown in Lines 3-11. Then, each chunk is copied to a free CPU buffer (from a CPU buffer pool described in Section 3.1) and persisted to disk at the appropriate offset from that checkpoint's start address. Once the last batch is copied and persisted, the thread responsible for this batch will execute Lines 16-34 to update the pointer to the latest checkpoint. While a checkpoint might be chunked in DRAM, it is saved in contiguous space in the persistent storage.

Checkpointing in Distributed Training. As described in Section 3.1, when distributed checkpointing is enabled, we have to make sure all peers have a globally consistent last checkpoint. To achieve this, each peer maintains a peer_check variable, representing the last globally consistent checkpoint. After a successful CAS (Line 24), each peer sends its checkpoint ID (curr_counter from Listing 1) to the peer with rank 0 and waits. Once rank 0 has received the checkpoint IDs from all peers, it notifies them to continue. Each peer updates its local peer_check to the new value and continues with lines 25 and 26. In our experiments, all peers had the same ordering of checkpoint IDs sent to rank 0. We are working towards improving the robustness and efficiency of this coordination mechanism for large-scale deployments.

4.2 Recovering with Checkpoints

CHECK_ADDR points to the last consistent checkpoint. To recover, PCcheck loads the checkpoint that corresponds to CHECK_ADDR from persistent storage into GPU memory with the help of a persistent iterator, which logs data read locations. The DNN job then resumes training.

We estimate the recovery time of PCcheck as follows. We assume each iteration takes time t, the checkpoint frequency is f iterations, and we allow up to N concurrent checkpoints, written in parallel to training. The time to write a checkpoint is T_w (taking $\frac{T_w}{t}$ iterations), and the time to load the checkpoint to GPU is l. PCcheck's recovery time is:

$$0 \le recovery \le l + f \cdot t + t \cdot min(N \cdot f, \frac{T_w}{t}). \qquad (4)$$

if $T_w > N \cdot f \cdot t$, training will stall at iterations $k \cdot (N+1) \cdot f$, i.e. $\le (N+1) \cdot f$ iterations need to be redone upon a failure. If $T_w < N$, the number of lost iterations is bound by T_w, meaning that in the worst case, $f + \frac{T_w}{t}$ iterations need to be repeated upon a failure.

For reference, the recovery time of CheckFreq and Gemini (which allow one checkpoint at a time to be persisted asynchronously with training), is given by $0 \le recovery \le l + 2 \cdot f \cdot t$ [50]. GPM [55] stalls training in order to persist each checkpoint, thus $0 \le recovery \le l + f \cdot t$

5 Evaluation

We implemented PCcheck on top of PyTorch [56] and Deepspeed [60].

5.1 Setup

Platform. We evaluate PCcheck on a2-highgpu-1g VMs on Google Cloud [32]. The VMs are equipped with an NVIDIA A100-40GB GPU, with PCIe3x16. Each VM has an Intel Xeon

Model	Dataset	Batch size in A100	Batch size in RTX	Checkpoint size (GB)
VGG16	ImageNet	32	32	1.1
BERT	SQuAD	3	3	4
TransformerXL	WikiText	64	32	2.7
OPT-1.3B	WikiText	1	-	16.2
OPT-2.7B	WikiText	1	-	45
Bloom-7B	WikiText	1	-	108

Table 3. Evaluated models. The checkpoint includes model and optimizer state.

2.2GHz processor with 1 NUMA node and 12 vCPUs, and 85 GB of DRAM. We attach a 1 TB SSD (`pd-ssd`) to each VM. The contents of `pd-ssd` disks remain even after a VM is preempted or crushed, and the disk can be reattached to another VM [3]. We use Ubuntu 20.04, CUDA 12.1, and PyTorch-2.1. We also evaluate our system on an Intel Optane persistent main memory (PMEM) device with an NVIDIA Titan RTX-24GB GPU, connected via PCIe3x8. For PMEM experiments, we used a machine with an Intel Xeon Gold 6248R 3GHz processor with 2 NUMA nodes, each with 48 cores. It has 128 GB of DRAM in total and 2 TB SSD. The machine has 4 NVDIMMs with 512 GB in total. We run PMEM in an AppDirect mode. The contents of the PMEM device also remain intact after a GPU failure. We use Ubuntu 20.04 and CUDA 11.6.

Models. We use models from the computer vision and NLP domain. We chose batch sizes following prior work [8, 69]. Table 3 lists the models, their checkpoint size, and microbatch size on each machine type. For OPT-2.7B and BLOOM-7B, we employ pipeline-parallel distributed training with 2 and 6 VMs respectively. The rest models are evaluated in single-GPU setups. Similar to related work [17, 50], each checkpoint includes model and optimizer state.

Baselines. We compare with CheckFreq [50], GPM [55] and Gemini [68]. Gemini does not have an open-source implementation, so we implemented it based on the paper description. Since Gemini relies on distributed training to replicate the training state, we evaluate it only in distributed setups. As GPM was designed to run on PMEM only, we adjust it to run with an SSD as well. GPM uses CUDA kernels to copy the checkpoint from GPU (instead of `cudaMemcpy`), which we keep in the SSD version. We `mmap` the file for checkpointing and use `cudaHostRegister` to allow access from GPU kernels. To persist a checkpoint, we call `cudaDeviceSynchronize` (to make sure copying has finished) and `msync` the mmaped file. We extend CheckFreq and GPM (initially proposed for single-GPU training) to work on distributed setups. We also compare to an ideal baseline, which saves checkpoints with zero overhead. We run experiments 3 times and report the average results. The standard deviation was consistently less than 0.2.

5.2 Performance Results

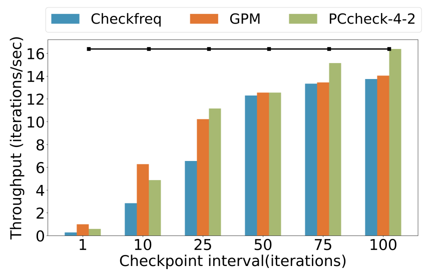
5.2.1 PCcheck with SSD. Figure 8 shows the training throughput when varying the checkpointing frequency. The legend denotes the PCcheck configuration as N-p, where N is the number of concurrent checkpoints and p is the number of parallel threads per checkpoint. We use a DRAM size of $2m$, where m is the checkpoint size. We evaluate PCcheck with the best configuration (determined by the profiling tool) and show a sensitivity study in § 5.4.

In single-GPU settings, at high checkpoint frequencies, CheckFreq has the highest overhead. For example, for the VGG16 workload (Figure 8a), CheckFreq leads up to 57× slowdown when checkpointing every iteration, and between 5.74×-1.19× slowdown when checkpointing every 10-100 iterations. Even though CheckFreq overlaps checkpointing and training, it can only support one checkpoint at a time. When the GPU tries to checkpoint before the previous checkpoint finishes, it stalls the entire pipeline. For less frequent checkpoints, and larger checkpoint sizes (i.e. BERT, OPT-1.3B), GPM performs much worse since it stops the GPU entirely for every checkpoint. Hence, GPM's overhead becomes more substantial than CheckFreq at a lower checkpointing frequency.

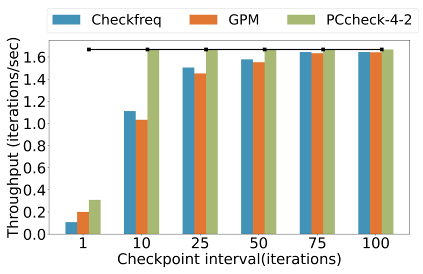
PCcheck outperforms both CheckFreq and GPM since it persists checkpoints with parallel workers and allows multiple concurrent checkpoints to minimize stalls during training. Moreover, it considers all system bottlenecks to optimize the data transfer. For example, in OPT-1.3 (Figure 8d), when checkpointing every 50 iterations, PCcheck's overhead is 1.02×, compared to 1.9× overhead for GPM and 1.17× for CheckFreq. In a few cases, GPM outperforms CheckFreq, when checkpointing every iteration (Figures 8a, 8d - 8f). In such scenarios of very frequent checkpointing, the concurrent checkpointing mechanism introduced by PCcheck offers limited benefits. Additionally, GPM directly copies to a memory-mapped file, which in some cases, leads to improved per-checkpoint-time, as will be discussed in 5.3. However, the training throughput of all baselines when checkpointing every iteration is very far from ideal, deeming these scenarios quite unrealistic. As checkpoint frequency decreases, GPM struggles to match PCcheck, since it does not parallelize checkpointing with training, and only keeps one active checkpoint at a time. As a result, while PCcheck can checkpoint every 10-25 iterations with minimal overheads, GPM's overheads remain significant at these frequencies.

We also experiment with a higher-end machine for OPT-1.3B, using a `Standard_NC40ads_H100_v5` VM from Azure [7] with an H100 GPU, and a 3.5TB NVMe SSD. We observe similar patterns for PCcheck and the baselines, since the iteration time was halved, and the disk bandwidth doubled.

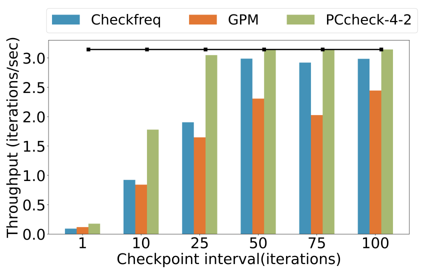
In multi-GPU settings, we observe that Gemini significantly degrades training throughput. Gemini transfers the training state over the inter-machine network, avoiding the



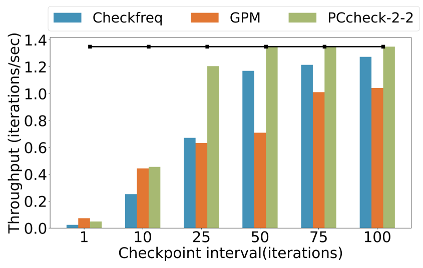
(a) VGG16



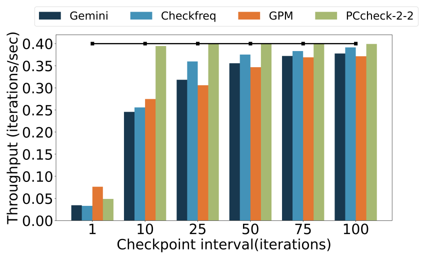
(b) Transformer-XL



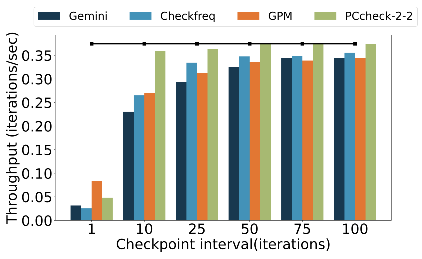
(c) BERT



(d) OPT-1.3B



(e) OPT-2.7B



(f) BLOOM-7B

Figure 8. Training throughput with checkpointing on SSD, A100, without failures. The horizontal black line represents the training throughput without checkpoints.

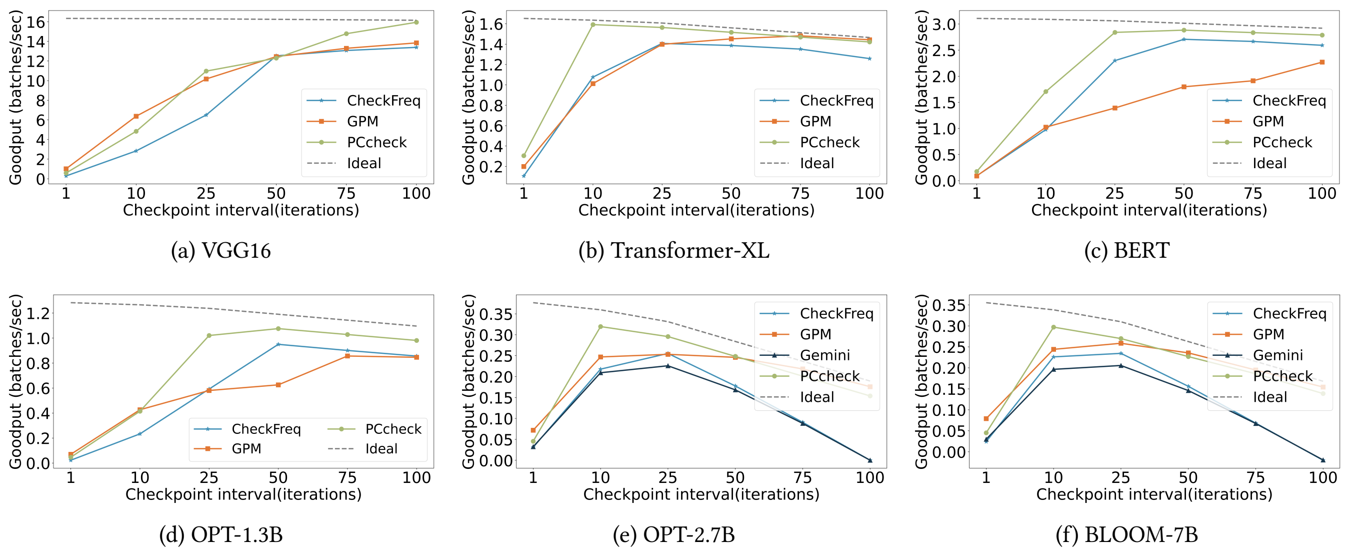
slower persistent storage bandwidth. However, it uses only one checkpoint at a time, meaning that the next checkpoint cannot start until the previous one has finished, which causes stalls when checkpointing frequently. Moreover, the inter-machine network in our setup is not fast enough to hide the checkpoint transfer time. The measured network bandwidth in our `a2-highgpu-1g` VMs is 15 Gbps, which is commonly seen in public cloud VMs [2]. Overall, Gemini leads to 1.62-1.06× and 1.65-1.08× lower throughput for OPT-2.7B and BLOOM-7B models respectively when checkpointing every 10-100 iterations. On the contrary, the slowdown of PCcheck at these frequencies is < 1.05× and < 1.02× respectively.

5.2.2 Recovery Times. At low checkpoint frequency, recovery time increases, reaching 80 seconds for OPT-1.3B when checkpointing every 100 iterations (Figure 8d). In this setting, CheckFreq has 5% overhead. With the same training throughput, PCcheck checkpoints every 50 iterations and recovers in 50 seconds. For BLOOM-7B (Figure 8f), PCcheck can recover in 26 seconds with 5% overhead, while CheckFreq and Gemini incur similar overhead and recover in 250 seconds.

5.2.3 Goodput with Spot Instances. The ultimate metric of interest is training goodput (useful throughput) in a real DNN cluster prone to failures or preemptions. In elastic distributed DNN training frameworks, such as Varuna [17], whenever any worker fails or gets preempted, all workers resume from the latest checkpoint. We compare goodput for PCcheck, CheckFreq, GPM, and Gemini assuming GPU availability based on the resource preemption trace collected by André et al. [16] in a Google Cloud A100 GPU spot instance cluster. We replay the resource preemption trace as follows: whenever a change in the allocated resources occurs (i.e. a VM is preempted or retrieved), the system stops and rolls back to the previous checkpoint. Thus, given the total training time T, the number of failures r, the average checkpoint interval f, and the average iteration time t (which includes checkpoint overheads), we compute how much time is spent doing useful work (goodput) and recovering. We use the average recovery time from 4.2 for each baseline, excluding the time needed to reattach a `pd-ssd` disk to a VM (which is around 5.5 sec, and similar for all baselines except Gemini that relies on DRAM copies.) The time spent making progress is $prog = T - rec$. During that time, the model has made progress on $seenBatches = \frac{prog}{t}$ batches, thus $goodput = \frac{seenBatches}{T}$.

Figure 9 shows goodput for the above resource availability simulation. Due to high resource preemptions in spot instance clusters, it is optimal to checkpoint every 10 – 25 iterations to avoid long recovery times. When checkpointing every iteration, most time is spent checkpointing, resulting in high overhead. Conversely, when checkpointing infrequently, most time is spent redoing lost work. Thus, frequent checkpoints are necessary, and PCcheck enables such fine-grained checkpointing.

For a fixed checkpoint frequency, the goodput depends on two key factors: 1) the training throughput in the absence of failures, and 2) the amount of lost work per failure. PCcheck



(a) VGG16 (b) Transformer-XL (c) BERT

(d) OPT-1.3B (e) OPT-2.7B (f) BLOOM-7B

Figure 9. Goodput when replaying a Google Cloud A100 GPU resource preemption trace [16]. Higher is better.

improves training throughput by introducing concurrent checkpointing. However, when multiple checkpoints are in progress simultaneously, the training may need to roll back more iterations (impacting factor 2). Thus, the overall goodput depends on the balance between training throughput and lost work. For instance, in Figure 9d, PCcheck achieves 1.77× higher goodput than CheckFreq, when checkpointing every 10 iterations. Without failures, (Figure 8d), the throughput of PCcheck and CheckFreq, is 0.5 iters/sec, and 0.256 iters/sec respectively, meaning that for the whole duration of the trace, PCcheck would have completed 6137 iterations, while CheckFreq would have completed 3404 iterations. When taking failures and average rollback iterations into account, PCcheck and CheckFreq will need to redo 550 and 257 iterations, respectively. Thus, the effective number of iterations (which reflects goodput) is 5587 and 3147 iterations for PCcheck and CheckFreq respectively.

This can be generalized when comparing PCcheck with other baselines that do not support concurrent checkpointing. In all cases, PCcheck greatly outperforms the other baselines, approaching the ideal upper bound. VGG16 (Figure 9a) has the smallest iteration time (60 ms) and its checkpoint size is 1.1 GB, meaning that the checkpoint overhead is quite high and the recovery time is low. This explains why all baselines have low goodput at frequent checkpointing. Nevertheless, PCcheck still performs similarly to the ideal case, starting from a checkpoint frequency of 100. In all the other cases, we notice that as the size of the checkpoint grows, and the iteration time decreases, the gap between PCcheck and the other baselines increases. Looking at each checkpoint frequency individually, and taking the ratio between PCcheck's goodput and the baselines' goodput, PCcheck gets up to 1.75×, 2.86×, and 2.75× higher goodput than GPM, CheckFreq, and Gemini, respectively. Alternatively, if, for each model, we consider the peak goodput of each baseline across all checkpoint frequencies, and compare it with PCcheck's peak goodput for that specific model, PCcheck has up to 1.27×, 1.25×, and 1.44× higher goodput than GPM, CheckFreq, and Gemini, respectively.

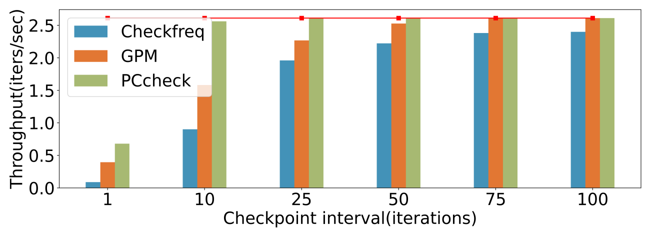


Figure 10. Checkpointing overhead for the BERT model when using an Intel Optane PMEM device.

We also experimented with higher numbers of concurrent checkpoints and observed slight decreases in goodput, since more concurrent checkpoints do not always translate to higher throughput, due to storage bandwidth limitations (see Sections 4 and 5.4). PCcheck picks a modest number of concurrent checkpoints (2-4) that achieves high throughput and goodput while accounting for scalability limitations.

5.2.4 PCcheck with PMEM.

In Figure 10, we evaluate PCcheck's impact on training throughput when using PMEM for checkpointing BERT (the largest model fit in this machine). Since the GPU on this machine has lower compute capability than the A100 GPU, the training throughput is decreased. Furthermore, the PMEM bandwidth is higher than the SSD's, thus the checkpointing overhead is lower. CheckFreq and GPM also perform better than in the SSD setup. Nevertheless,

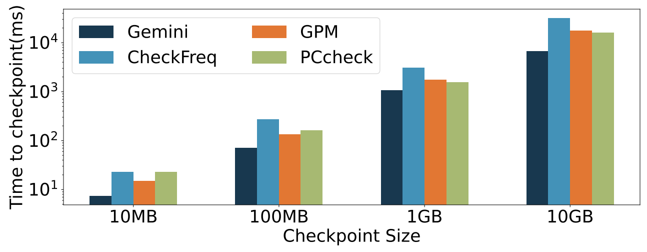


Figure 11. Time to persist 1 checkpoint, with varying sizes, SSD, A100. y-axis is log scale.

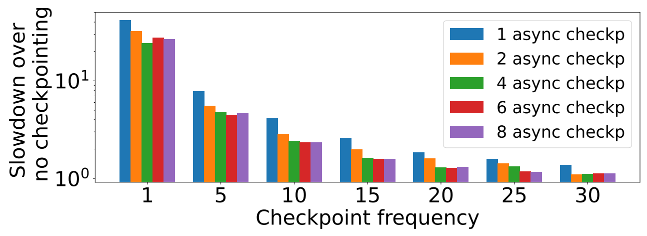


Figure 12. Slowdown over no checkpointing, VGG-16, varying frequencies and the number of concurrent checkpoints.

PCcheck still outperforms both in all checkpoint frequencies. By checkpointing every 10 iterations instead of 100 iterations, as CheckFreq, the recovery time decreases by 10×, while keeping the same overhead.

5.3 Microbenchmarks

Figure 11 shows the time to write a checkpoint of various sizes using SSD, or over the network in Gemini's case. Gemini has the lowest time per checkpoint since it does not write to storage. However, as only one checkpoint is allowed at a time, training performance is significantly degraded (§ 5.2). PCcheck optimizes the entire copying path, outperforming CheckFreq and GPM by up to 1.9×. PCcheck uses GPU copy engines to move data to DRAM, employs multiple threads to copy the data, and persists in a pipelined manner, fully saturating the persistent storage device.

5.4 Sensitivity Study

As Table 2 shows, the key factors that affect PCcheck's performance are 1) the number of concurrent checkpoints, 2) the number of parallel threads per checkpoint, 3) the chunk size, and 4) the amount of DRAM used for checkpointing. We now examine the impact of these factors in PCcheck's performance. We show results for the SSD setup, but results with PMEM show similar trends.

5.4.1 Concurrent Checkpoints. Figure 12 demonstrates the slowdown compared to no checkpointing with varying frequencies and number of concurrent checkpoints for VGG-16. Even though VGG-16 has relatively small checkpointing size, using more than one checkpoint is consistently better.

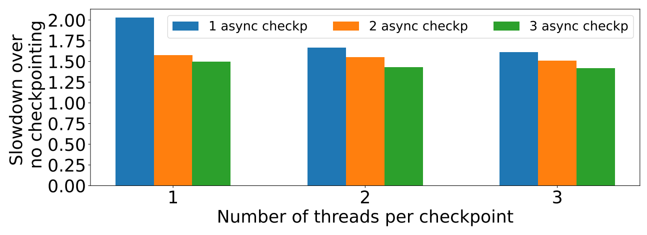


Figure 13. Slowdown over no checkpointing, OPT-350M, with a fixed checkpoint frequency of 10 iterations, varying the number of parallel threads per checkpoint.

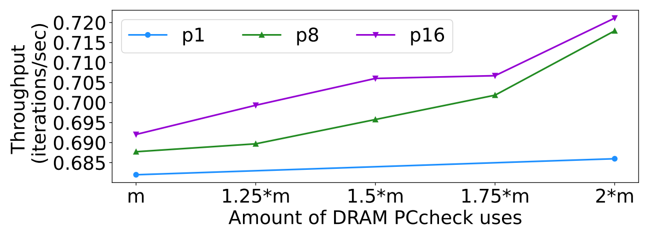


Figure 14. PCcheck throughput of OPT-1.3B, checkpointing every 15 iterations. X-axis varies the DRAM size for checkpointing. m is the checkpoint size. p_x denotes the pipelined version of PCcheck, splitting the checkpoint into x chunks.

We generally do not need more than 4 checkpoints at a time, as they already saturate the SSD bandwidth, and more threads start competing for resources.

5.4.2 Parallel threads per checkpoint. Figure 13 demonstrates the slowdown compared to no checkpointing for OPT-350M at a fixed checkpoint frequency of 10 iterations, while varying the number of parallel threads per checkpoint. Using 3 threads (instead of 1) leads to 1.36×, 1.16×, 1.13× improvement when having 1, 2, 3 concurrent checkpoints respectively. Thus, the benefit of having multiple parallel checkpoints is significant, but decreases as the number of concurrent checkpoints increases, due to extra contention for storage bandwidth. This is taken into account when PCcheck configures the parameters to use (Section 3.4). We observe similar effects for both SSD and PMEM, since both devices can employ low-to-moderate levels of parallelism [74].

5.4.3 Chunk size and DRAM buffer size. Figure 14 shows how the training throughput of the OPT-1.3B model changes for varying sizes of the DRAM buffer and pipeline chunks. We observe that pipelining leads to slightly higher throughput compared to the non-pipelined case, although the differences are quite small. Moreover, although we use a DRAM buffer size of $2 \cdot m$ for our experiments, reducing this size does not greatly affect performance. Using a DRAM buffer size of m adds only up to 7% overhead compared to using a buffer of $2 \cdot m$, thus PCcheck can safely be used under tight memory constraints.

6 Related Work

Checkpointing in HPC and Other Domains. Checkpointing is a well-known technique in various domains such as distributed systems [21, 29, 42, 65], HPC [26, 30, 41], and databases [46, 62, 77]. These works focus on recovering efficiently and reducing checkpoint latency and bandwidth. PCcheck provides an efficient checkpointing mechanism tailored for DNN training supporting both PMEMs and SSDs.

Checkpointing Recommendation Models. Several systems focus on checkpointing large recommendation models. Check-N-Run [28] proposes differential checkpointing (i.e., leveraging the fact that only a small part of the recommendation model is updated each iteration), quantization, and asynchronous checkpointing. CPR [45], following SCAR [57]'s paradigm, proposes partial recovery, which restores the training state only for the failed node. These techniques are orthogonal to PCcheck.

Transferring Data from GPU to Storage. GPUDirect storage [4] allows NVIDIA GPUs to access PCIe storage devices without involving the CPU. FlashNeuron [19] uses GPUDirect to offload a part of the training state to an NVMe device, enabling training larger models with larger batch sizes than what can fit on a single GPU. Unified Virtual Memory [11] allows applications to access data both on the GPU and CPU, regardless of whether they run on GPU or CPU. Several works [47, 55] have extended UVM to include persistent memory devices. GPM [55] extends UVM to include the PMEM region, to allow for fine-grained persistence of GPU-accelerated applications. DRAGON [47] uses UVM with PMEM to increase the amount of available memory. Despite UVM's advantages in providing an improved programming experience through unified memory access, our experiments suggest that having an intermediate buffer in DRAM for copying data is beneficial for the checkpoint throughput.

7 Conclusion

PCcheck is a framework for orchestrating multiple checkpoints on persistent storage during DNN training. While prior DNN checkpointing systems support one checkpoint at a time and hence stall training at high checkpoint frequencies, PCcheck avoids these stalls by efficiently managing multiple concurrent checkpoints. Thus, PCcheck enables fast, frequent checkpointing, which is critical for dealing with common preemptions and failures in large-scale DNN hardware clusters as well as for fine-grained model training monitoring. PCcheck achieves up to 2.86× higher DNN training goodput in preemptible cloud VM environments compared to state-of-the-art checkpointing systems.

8 Acknowledgements

We thank the anonymous reviewers from ASPLOS'25 and our shepherd, Sam Ainsworth, for their valuable feedback. We also thank the anonymous reviewers from ASPLOS'24 and ATC'24 for their valuable comments on earlier versions of the paper. We thank Xindi Zuo for her technical contribution early stages of the project and her work on non-volatile memory support. We thank Daniel Schwyn, Dimitris Koutsoukos, and Dario Korolija for their technical advice.

References

[1] A2 machine series, google cloud. https://cloud.google.com/compute/docs/gpus#a100-gpus. Accessed: 2024-10-24.

[2] Gcp network bandwidth. https://cloud.google.com/compute/docs/network-bandwidth. Accessed: 2023-07-23.

[3] Google cloud persistent disk types. https://cloud.google.com/compute/docs/disks#disk-types. Accessed: 2024-10-24.

[4] Gpudirect storage: A direct path between storage and gpu memory. https://developer.nvidia.com/blog/gpudirect-storage/. Accessed: 2023-08-05.

[5] How to overlap data transfers in cuda c/c++. https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/. Accessed: 2024-06-20.

[6] Hugging face. https://huggingface.co/. Accessed: 2023-07-23.

[7] Ncads h100 v5-series. https://learn.microsoft.com/en-us/azure/virtual-machines/ncads-h100-v5. Accessed: 2024-06-20.

[8] Nvidia deep learning examples. https://github.com/NVIDIA/DeepLearningExamples. Accessed: 2023-07-23.

[9] The stanford question answering dataset. https://rajpurkar.github.io/SQuAD-explorer/. Accessed: 2024-10-24.

[10] Torchvision. https://github.com/pytorch/vision. Accessed: 2023-07-23.

[11] Unified memory for cuda beginners. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/. Accessed: 2023-08-06.

[12] Wikitext. https://developer.ibm.com/exchanges/data/all/wikitext-103/. Accessed: 2024-10-24.

[13] Discontinuation of intel optane. https://www.intel.com/content/www/us/en/support/articles/000024320/memory-and-storage.html, 2022. Accessed: 2023-08-10.

[14] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, USA, 2016. USENIX Association.

[15] Arize AI. The ml observability platform for practitioners. https://arize.com/, 2022. Accessed: 2023-08-10.

[16] Joel André, Foteini Strati, and Ana Klimovic. Exploring learning rate scaling rules for distributed ml training on transient resources. In *Proceedings of the 3rd International Workshop on Distributed Machine Learning*, DistributedML '22, page 1–8, New York, NY, USA, 2022. Association for Computing Machinery.

[17] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: Scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 472–487, New York, NY, USA, 2022. Association for Computing Machinery.

[18] AWS. Aws spot instance interruption notices. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-instance-termination-notices.html. Accessed: 2024-06-16.

[19] Jonghyun Bae, Jongsung Lee, Yunho Jin, Sam Son, Shine Kim, Hakbeom Jang, Tae Jun Ham, and Jae W. Lee. FlashNeuron: SSD-Enabled Large-Batch training of very deep neural networks. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 387–401. USENIX Association, February 2021.

[20] Stella Biderman, Hailey Schoelkopf, Quentin Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, Aviya Skowron, Lintang Sutawika, and Oskar van der Wal. Pythia: A suite for analyzing large language models across training and scaling, 2023.

[21] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, feb 1985.

[22] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[23] CXL. Compute express link. https://www.computeexpresslink.org/, 2022.

[24] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context, 2019.

[25] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[26] Xiangyu Dong, Yuan Xie, Naveen Muralimanohar, and Norman P. Jouppi. Hybrid checkpointing using emerging nonvolatile memories for future exascale systems. *ACM Trans. Archit. Code Optim.*, 2011.

[27] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, and Angela Fan et al. The llama 3 herd of models, 2024.

[28] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Murali Annavaram, Krishnakumar Nair, and Misha Smelyanskiy. Check-n-run: A checkpointing system for training recommendation models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 929–943, Renton, WA, April 2022. USENIX Association.

[29] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, sep 2002.

[30] Shen Gao, Bingsheng He, and Jianliang Xu. Real-time in-memory checkpointing for future hybrid memory systems. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, page 263–272, New York, NY, USA, 2015. Association for Computing Machinery.

[31] Wei Gao, Zhisheng Ye, Peng Sun, Yonggang Wen, and Tianwei Zhang. Chronus: A novel deadline-aware scheduler for deep learning training jobs. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 609–623, New York, NY, USA, 2021. Association for Computing Machinery.

[32] Google. Google cloud gpus. https://cloud.google.com/compute/docs/gpus. Accessed: 2023-08-06.

[33] Google. Google cloud spot vms. https://cloud.google.com/spot-vms. Accessed: 2023-08-10.

[34] Alex Graves, Abdel-rahman Mohamed, and Geoffrey E. Hinton. Speech recognition with deep recurrent neural networks. *CoRR*, abs/1303.5778, 2013.

[35] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, Boston, MA, February 2019. USENIX Association.

[36] Tanmaey Gupta, Sanjeev Krishnan, Rituraj Kumar, Abhishek Vijeev, Bhargav Gulavani, Nipun Kwatra, Ramachandran Ramjee, and Muthian Sivathanu. Just-in-time checkpointing: Low cost error recovery from deep learning training failures. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 1110–1125, New York, NY, USA, 2024. Association for Computing Machinery.

[37] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 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*, pages 770–778. IEEE Computer Society, 2016.

[38] Yi He, Mike Hutton, Steven Chan, Robert De Gruijl, Rama Govindaraju, Nishant Patil, and Yanjing Li. Understanding and mitigating hardware failures in deep learning training systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[39] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, Renton, WA, July 2019. USENIX Association.

[40] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, Santa Clara, CA, April 2024. USENIX Association.

[41] Sudarsun Kannan, Naila Farooqui, Ada Gavrilovska, and Karsten Schwan. Heterocheckpoint: Efficient checkpointing for accelerator-based systems. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 738–743, 2014.

[42] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, 1987.

[43] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[44] Youjie Li, Amar Phanishayee, Derek Murray, Jakub Tarnawski, and Nam Sung Kim. Harmony: overcoming the hurdles of gpu memory capacity to train massive dnn models on commodity servers. *Proc. VLDB Endow.*, 15(11):2747–2760, jul 2022.

[45] Kiwan Maeng, Shivam Bharuka, Isabel Gao, Mark Jeffrey, Vikram Saraph, Bor-Yiing Su, Caroline Trippel, Jiyan Yang, Mike Rabbat, Brandon Lucia, and Carole-Jean Wu. Cpr: Understanding and improving failure tolerant training for deep learning recommendation with partial recovery. 11 2020.

[46] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. Rethinking main memory oltp recovery. In *2014 IEEE 30th International Conference on Data Engineering*, pages 604–615, 2014.

[47] Pak Markthub, Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Satoshi Matsuoka. Dragon: Breaking gpu memory capacity limits with direct nvm access. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 414–426, 2018.

[48] Meta. Democratizing access to large-scale language models with opt-175b. https://ai.facebook.com/blog/democratizing-access-to-large-scale-language-models-with-opt-175b/, 2022.

[49] Microsoft. Azure spot virtual machines. https://learn.microsoft.com/en-us/azure/virtual-machines/spot-vms. Accessed: 2023-08-10.

[50] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. Checkfreq: Frequent, fine-grained DNN checkpointing. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 203–216. USENIX Association, February 2021.

[51] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. Looking beyond GPUs for DNN scheduling on Multi-Tenant clusters. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 579–596, Carlsbad, CA, July 2022. USENIX Association.

[52] Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, page 103–112, New York, NY, USA, 2013. Association for Computing Machinery.

[53] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Analysis and exploitation of dynamic pricing in the public cloud for ml training. In *Workshop on Distributed Infrastructure, Systems, Programming, and AI*, 2020.

[54] Bogdan Nicolae, Jiali Li, Justin M. Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. Deepfreeze: Towards scalable asynchronous checkpointing of deep learning models. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 172–181, 2020.

[55] Shweta Pandey, Aditya K Kamath, and Arkaprava Basu. Gpm: Leveraging persistent memory from a gpu. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 142–156, New York, NY, USA, 2022. Association for Computing Machinery.

[56] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. *PyTorch: An Imperative Style, High-Performance Deep Learning Library.* Curran Associates Inc., Red Hook, NY, USA, 2019.

[57] Aurick Qiao, Bryon Aragam, Bingjing Zhang, and Eric Xing. Fault tolerance in iterative-convergent machine learning. volume abs/1810.07354, 2018.

[58] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 1–18. USENIX Association, July 2021.

[59] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. ArXiv, May 2020.

[60] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 3505–3506, New York, NY, USA, 2020. Association for Computing Machinery.

[61] Nathalie Rauschmayr, Vikas Kumar, Rahul Huilgol, Andrea Olgiati, Satadal Bhattacharjee, Nihal Harish, Vandana Kannan, Amol Lele, Anirudh Acharya, Jared Nielsen, Lakshmi Ramakrishnan, Ishan Bhatt, Kohen Chia, Neelesh Dodda, Zhihan Li, Jiacheng Gu, Miyoung Choi, Balajee Nagarajan, Jeffrey Geevarghese, Denis Davydenko, Sifei Li, Lu Huang, Edward Kim, Tyler Hill, and Krishnaram Kenthapadi. Amazon sagemaker debugger: A system for real-time insights into machine learning model training. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 770–782, 2021.

[62] Kun Ren, Thaddeus Diamond, Daniel J. Abadi, and Alexander Thomson. Low-overhead asynchronous checkpointing in main-memory database systems. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1539–1551, New York, NY, USA, 2016. Association for Computing Machinery.

[63] Frank Schneider, Felix Dangel, and Philipp Hennig. Cockpit: A practical debugging tool for the training of deep neural networks, 2021.

[64] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

[65] Rob Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, aug 1985.

[66] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large DNNs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 497–513. USENIX Association, 2023.

[67] Marcel Wagenländer, Luo Mai, Guo Li, and Peter Pietzuch. Spotnik: Designing distributed machine learning for transient cloud resources. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association, July 2020.

[68] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 364–381, New York, NY, USA, 2023. Association for Computing Machinery.

[69] Zhuang Wang, Haibin Lin, Yibo Zhu, and T. S. Eugene Ng. Hi-speed dnn training with espresso: Unleashing the full potential of gradient compression with near-optimal usage strategies. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 867–882, New York, NY, USA, 2023. Association for Computing Machinery.

[70] Weights and Biases. The ai developer platform. https://wandb.ai/site, 2022. Accessed: 2023-08-10.

[71] BigScience Workshop, :, Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, and Jonathan Tow et al. Bloom: A 176b-parameter open-access multilingual language model, 2023.

[72] Carole-Jean Wu, Ramya Raghavendra, Udit Gupta, Bilge Acun, Newsha Ardalani, Kiwan Maeng, Gloria Chang, Fiona Aga Behram, James Huang, Charles Bai, Michael Gschwind, Anurag Gupta, Myle Ott, Anastasia Melnikov, Salvatore Candido, David Brooks, Geeta Chauhan, Benjamin Lee, Hsien-Hsin S. Lee, Bugra Akyildiz, Maximilian Balandat, Joe Spisak, Ravi Jain, Mike Rabbat, and Kim Hazelwood. Sustainable ai: Environmental implications, challenges and opportunities. 2021.

[73] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, 2016.

[74] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association.

[75] Binhang Yuan, Yongjun He, Jared Quincy Davis, Tianyi Zhang, Tri Dao, Beidi Chen, Percy Liang, Christopher Re, and Ce Zhang. Decentralized training of foundation models in heterogeneous environments, 2023.

[76] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona 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. Opt: Open pre-trained transformer language models, 2022.

[77] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 465–477, Broomfield, CO, October 2014. USENIX Association.

A Artifact Appendix

A.1 Abstract

The artifact consists of the source code of PCcheck[4], benchmarks for deployment and evaluation, scripts and instructions for reproducing the key results of the paper, in comparison with related work, microbenchmarks, and sensitivity study.

The evaluation focuses on reproducing key results from the paper:

- **Figure 8:** Effect on training throughput, and comparison with baselines, when varying the checkpoint frequency. To reduce evaluation hours and costs, we focus on one model.
- **Figure 9:** Effect on goodput, and comparison with baselines, when varying the checkpoint frequency, given a preemption trace. To reduce evaluation hours and costs, we focus on one model.
- **Figure 11:** Microbenchmarks, showing the end-to-end time to persist a checkpoint for different baselines.
- **Figure 12:** Sensitivity analysis, varying the number of concurrent checkpoints.

A.2 Artifact check-list (meta-information)

- **Algorithm:** The paper proposes PCcheck, a framework for fine-grained checkpointing of ML applications. Part of PCcheck is an algorithm that enables multiple checkpoints to be on-they-fly concurrently (described in section 4.1).
- **Program:** We evaluate PCcheck using benchmarks from TorchVision [10], the NVIDIA DeepLearning repo [8] and HuggingFace [6]. All benchmarks are public. We provide instructions for setting up in our repo.
- **Compilation:** To compile PCcheck we used GCC-9.4, and CUDA-12.1.
- **Model:** For our evaluation, we used VGG-16 [64] (138M params), Transformer-XL [24] (192M params), BERT [25] (345M params), various sizes of OPT [76] (350M, 1.3B, 2.7B params), and BLOOM [71] (7B params). All models are open-source. We provide instructions to download all models in our repo. As part of the artifact evaluation process, we focus on Transformer and OPT.
- **Data set:** We use SQUAD [9] and WikiText [12]. Both datasets are downloaded automatically with our scripts provided in this link.
- **Run-time environment:** For our evaluation, we used Ubuntu 20.04. We used GCC-9.4, Python 3.9.18, NVIDIA Driver 530.30.02, and CUDA 12.1. We provide instructions here (they need sudo access).
- **Hardware:** We used a2-highgpu-1g VMs from Google Cloud Platform [1]. Each VM has an A100-40GB GPU attached, 1TB of pd-ssd, 12 vCPUs, and 85 GB of DRAM.
- **Execution:** To get more accurate measurements, we recommend solo access to the machine. PCcheck uses thread pinning to specific cores for higher performance. To replicate all of the experiments, 6 hours are needed on average
- **Metrics:** We report training throughput, goodput and relative slowdown. The scripts included in our repo automatically collect these metrics.
- **Output:** For each graph, we generate a `.csv` file containing the results, and we provide the scripts to generate the corresponding plot.
- **Experiments:** Our README contains instructions on how to prepare experiments and reproduce results. We provide instructions on how to install and run a basic test. To reproduce results, a user needs to download the appropriate models and datasets with our setup_models_and_datasets.sh script. Each Figure in the paper corresponds to a script that runs the experiments and generates the plots.
- **How much disk space required (approximately)?:** We recommend at least 1TB of SSD. We used pd-ssd from Google Cloud for our evaluation.
- **How much time is needed to complete experiments (approximately)?:** 6 hours. We provide a breakdown in out README
- **Publicly available?:** The artifact will be publicly available.
- **Code licenses (if publicly available)?:** Our repo is using the MIT License.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.14054822

A.3 Description

A.3.1 How to access. The artifact is available in https://github.com/eth-easl/pccheck and can be cloned from there. The DOI for the artifact is https://doi.org/10.5281/zenodo.14054822.

A.3.2 Hardware dependencies. The artifact has been tested on GCP machines with the following specifications:

- a2-highgpu-1g (12 vCPUs, 85GB of DRAM) with 1 A100-40GB GPU attached.
- 1TB pd-ssd attached.

We have set up a Google Cloud Platform (GCP) Project for VM creation and artifact evaluation. We have created a GCP VM image with the NVIDIA drivers installed, to allow for faster deployment. We encourage the reviewers to create a Google Cloud account (with an anonymous email) and reach out to us to be added to the GCP project for conducting experiments.

A.3.3 Software dependencies. We use Ubuntu 20.04 and the following packages:

- GCC-9.4
- Python 3.9
- NVIDIA Driver 565.57.01
- CUDA 12.1

[4]https://github.com/eth-easl/pccheck

- NVIDIA Apex

Furthermore, we use the following Python packages:

- Torch 2.1
- Deepspeed 0.12.6
- Torchvision 0.14.1
- HF accelerate 0.20.3

We have set up a GCP Image with all the software dependencies pre-installed. We encourage reviewers to deploy and evaluate PCcheck using this image, as described in our README.

A.3.4 Data sets. We use SQUAD and WikiText. Both datasets are downloaded automatically with our scripts provided here

A.3.5 Models. We used VGG-16 [64] (138M params), Transformer-XL [24] (192M params), BERT [25] (345M params), various sizes of OPT [76] (350M, 1.3B, 2.7B params), and BLOOM [71] (7B params). All models are open-source. We provide instructions on how to download all models in our repo.

A.4 Installation

As a first step, the user needs to set up an environment with the necessary dependencies. We provide a GCP image, with all software dependencies preinstalled. We also provide the exact commands in the install_preq_at_vm.sh script on how to install all dependencies from scratch.

Next, the user needs to download the PCcheck repo, and install PCcheck (and build the other baselines) using our install.sh script.

The user can run a simple example by executing our test_simple.sh script. This runs a few training iterations of the VGG-16 model, using PCcheck to checkpoint every 50 iterations.

We provide full instructions in our README.

A.5 Experiment workflow

To replicate the results from our paper, these steps need to be followed:

1. Download the required models and datasets using the setup_models_and_datasets.sh script.
2. For each Figure, we provide scripts to run the appropriate experiments, collect the results, and generate plots. The user needs to run the respective script for each plot.

These steps are documented in our README.

A.6 Evaluation and expected results

We first show how PCcheck performs compared to baselines under various checkpoint frequencies.

- Figure 8 shows the training throughput of PCcheck under various checkpoint frequencies for different models, and compares it with baselines such as CheckFreq [50], GPM [55], and Gemini [68]. To reduce evaluation time and costs, we focus on 8b. Overall, PCcheck allows checkpointing at more frequent intervals compared to the baselines, with minimal impact on throughput.
- Figure 9 shows the goodput for PCcheck and the rest baselines for a given preemption trace. Again, we focus on 9b. Since PCcheck allows more frequent checkpointing than the baselines, it also leads to higher goodput.

We then run some basic microbenchmarks as depicted in Figure 11, where we plot the end-to-end time to copy and persist a checkpoint for various baselines. We omit Gemini due to the complexity of the setup.

Finally, we conduct a sensitivity study, analyzing the factors that affect PCcheck's performance. We focus on Figure 12. This figure depicts the slowdown in training throughput for the VGG-16 model, under various checkpoint frequencies, when varying the number of concurrent checkpoints. We observe that no more than 4 concurrent checkpoints are needed at a time, due to SSD bandwidth saturation.